Model-based verification of a security protocol for conditional access to services

G. Leduc, O. Bonaventure, L. Léonard, E. Koerner, C. Pecheur
Université de Liège, Institut d'Electricité Montefiore, B 28, B-4000 Liège 1, Belgium
Phone: + 32 4 3662691 Fax: + 32 4 3662989 E-mail: leduc@montefiore.ulg.ac.be

Abstract

We use the formal language LOTOS to specify and verify the robustness of the Equicrypt protocol under design in the European OKAPI project for conditional access to multimedia services. We state some desired security properties and formalize them. We describe a generic intruder process and its modelling, and show that some properties are falsified in the presence of this intruder. The diagnostic sequences can be used almost directly to exhibit the scenarios of possible attacks on the protocol. Finally, we propose an improvement of the protocol which satisfies our properties.

1. Introduction

The Equicrypt protocol is a conditional access protocol under design in the European ACTS OKAPI project [GBM96]. It allows users to subscribe to multimedia services such as video on demand. Equicrypt is designed to be equitable, meaning that any user or service provider can potentially enter the system provided that it complies with this minimal protocol. This contrasts with proprietary systems which all use different conditional access protocols and thus oblige users to implement almost as many protocols as there are different service providers, which is a severe limitation.

After a brief description of the Equicrypt protocol and its modelling in the formal language LOTOS [ISO 8807, BoB87] at an appropriate abstraction level, we will describe the verification process. We state some desired safety properties and formalize them. Then we describe a generic intruder process and its modelling, and show that some properties are falsified in the presence of this intruder. The diagnostic sequences can be used almost directly to exhibit the scenarios of possible attacks on the protocol. Two of them are presented.

Our approach thus consists of using a generic formal language and its associated verification methods and tools to verify security protocols. In this respect, we give a brief comparison with other works and refer to [Mea95] for a more complete survey on this topic.

Special modal logics have been designed to verify security protocols. The most well-known such logic is the BAN logic [BAN90], which is intentionally limited to authentication properties. Other more expressive logics have been proposed, for example to model knowledge. Such logics have been used successfully to verify several protocols, but have not proved very effective in

some other circumstances. For example, a thorough analysis of the Needham-Schroeder protocol using the BAN logic is given in [BAN90]. However, this analysis did not find the security problem subsequently reported in [Low95], although it was an authentication problem.

Another approach consists of using state-machines. In the Interrogator system [MCF87], the participants are modelled as communicating state-machines and the network is assumed to be under the control of an intruder, which can intercept messages, destroy or modify them, or pass them through unmodified. Given a final state in which the intruder knows something which should be secret, the tool searches exhaustively the state space to locate this state. The idea of introducing an intruder was first proposed in [DEK82, DoY83] in another setting. We also base our method on such an intruder. The NRL Protocol Analyser [KMM94, Mea94] is similar to the Interrogator, but the goal is here to prove the unreachability of some undesirable states. It can deal with infinite-state systems but the search is less automated than in the Interrogator. The common point of the previous works is that some pathological target states have to be defined prior to any search.

With model-checking, the method is different. It consists of specifying the desired properties in some (usually temporal) logic, or as a reference specification. To our knowledge, the first application of model-checking to the verification of security protocols is [Low96] where the Needham-Schroeder authentication protocol was specified in CSP [Hoa 85] and model-checked by the FDR tool. In this work the authentication properties were specified as correspondence properties, like in [WoL93], which require that certain events can take place only if others have taken place previously. Independently of [Low96], we have specified the Equicrypt protocol in LOTOS and used the model-checker of the Eucalyptus toolbox to verify it [LBK+96]. The present paper is an extended version of [LBK+96]. LOTOS had already been used to specify security protocols in [Var89] but no verification was attempted.

The model-based methods are extremely powerful at finding subtle flaws in protocols, but are less adequate to prove correctness when no bug is found. This is because they are applied on simplified, though realistic, models of the systems. On the other hand, theorem provers [Kem89, ChG90, Bol96] can provide such proofs and can also deal more easily with infinite-state systems. However, the proofs are usually less automated, and when no proof has been derived for a given property, it is not easy to know whether the property is wrong or whether the tool simply did not find it. In particular, an attack that falsifies the property is not provided automatically.

2. Presentation of the Equicrypt protocol

The following is a short summary of the Equicrypt protocol [LBQ+96] under design in the OKAPI project [GBM96].

2.1. Structure

The aim of the Equicrypt system is to control the access to multimedia services broadcast on a public channel (the main examples being cable or satellite TV programs or Video On Demand

services). Scrambling is used to make the data usable by authorized users only, using a special decoding device. To avoid requiring different decoders for every accessed service (provider), a unique decoder uses a public-key cryptography protocol to subscribe to and decode different services. An independent entity known as the Trusted Third Party (TTP) acts as a registering authority trusted by both users and providers.

The Equicrypt system (fig. 1) thus involves three kinds of entities:

- the Set Top Units (STU) of users,
- the Service Providers (SPv),
- the Trusted Third Party (TTP).

The communications between SPvs and STUs use an insecure (broadcast) channel, whereas their communications with the TTP use secure channels. The environment of this system is composed of video servers producing video images (through the `source` interface) and users watching these images (through the `tv` interface) and subscribing to services (through `cmd` interface). Every Set Top Unit contains an Access Control Unit (ACU) which is basically a smart card that executes the security functions. In fact, it is the ACU that acts on the user's behalf in the Equicrypt system, and contains the critical security information.

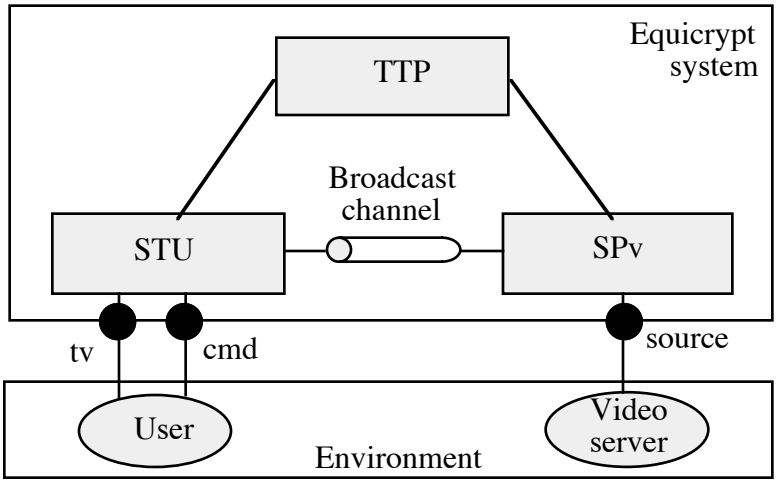

Fig. 1: The overall system

2.2. Operations

Subscribing to services via TTPs occurs in three main phases:

- **ACU certification:** a Certification Authority checks that the ACU performs sanely (e.g. will not trick the TTP or reveal concealed information) and assigns it a hard-wired certification number.
- **ACU registration:** an ACU gets registered by a TTP. It uses a zero-knowledge protocol [GQ88] to prove the validity of its registration identifier w.r.t. its certificate without disclosing the latter.

- **Service subscription:** an ACU asks for access to a service. The SPv gets the ACU's registration information from the TTP. If this succeeds, the SPv sends to the ACU a key allowing it to descramble the desired service. Cryptography is used to protect against eavesdroppers.

ACU certification is an administrative procedure that does not concern us. We shall specify certified ACUs, and describe ACU registration and service subscription, as well as the broadcasting of the service in itself.

2.3. Modelling of encryption operations

The TTP algorithms involve several encryption operations, for which we give an abstract view only. Each scheme uses peer encryption and decryption keys K_E and K_D and functions $E(\_, \_)$ and $D(\_, \_)$ such that $D(K_D, E(K_E, m)) = m$ for any message m. In the simple cases such as secret key cryptography, $K_E = K_D$ is the shared secret key. In public key cryptography, K_E is public and K_D is private for encryption (or vice-versa for authenticity).

2.4. Description of operations

All the messages have the following structure:

Number: Source → Destination: Message_id ⟨parameters_list⟩

Therefore we omit the source and destination in parameters_list.

We also use the more compact notation $\{m\}_K$ to denote the message m encrypted with the key K, that is $\{m\}_K = E(K, m)$.

Identification and keys:

During certification, a user's ACU that has the serial identification number A is assigned a certificate C_a correlated to (a hash value of) its identity A, in such a way that they operate as peer encryption/decryption keys, that is $D(A, \{m\}_{C_a}) = m$. Then the ACU generates peer keys K_a^p and K_a^s, respectively public and private (secret), and chooses an identification number A' which can be seen as an alias of A for privacy purposes. K_a^s will remain internal to the user's ACU.

A service provider B has peer public and private keys K_b^p and K_b^s and advertises K_b^p with proposed services. For each provided service S, it also keeps a key K_S.

Let N_{ta} be a nonce[1] (called challenge) generated by the TTP T to authenticate user A; and let N_{ab} be another nonce (called ticket) generated by user A to authenticate the service provider B.

[1] A nonce is a random number generated with the purpose of being used once (i.e. in at most one run of the protocol).

Registration:

1: $A \rightarrow T$:Register request $\langle A', K_a^p \rangle$

If T accepts the request, it generates and sends a random challenge N_{ta}.

2: $T \rightarrow A$:Register challenge $\langle \{N_{ta}\}_{K_a^p} \rangle$

3: $A \rightarrow T$:Register response $\langle \{N_{ta}\}_{C_a} \rangle$

T checks $D(A, \{N_{ta}\}_{C_a}) = N_{ta}$. If so, A is registered with its public key K_a^p and its alias A', that is T stores the tuple $\langle A, A', K_a^p \rangle$ in its directory.

3': $T \rightarrow A$:Register ack

The real algorithm also involves a random number introduced by the ACU for preventing the TTP from guessing C_a. We ignore this in our model.

Subscription:

4: ? $\rightarrow B$: Subscribe request $\langle A', S, \{N_{ab}\}_{K_b^p} \rangle$

The ? indicates that the source (user Id) is not part of the message.

B gets $N_{ab} = D(K_b^s, \{N_{ab}\}_{K_b^p})$. If this nonce N_{ab} has been used in a previous subscription, B ignores the request, otherwise the protocol continues.

5: $B \rightarrow T$:Check request $\langle A' \rangle$

6^+: $T \rightarrow B$:Check answer $\langle A', \text{true}, K_a^p \rangle$

6^-: $T \rightarrow B$:Check answer $\langle A', \text{false}, - \rangle$

A negative answer is provided to B if A is not registered or blacklisted. If the answer is positive, then B sends A the message 7^+ containing the service key K_S, otherwise B sends message 7^-.

7^+: $B \rightarrow$ All: Subscribe answer $\langle S, \text{true}, \{N_{ab}, K_S\}_{K_a^p} \rangle$

Only A can get $\langle N_{ab}, K_S \rangle = D(K_a^s, \{N_{ab}, K_S\}_{K_a^p})$. A then checks N_{ab} and, if correct, registers K_S and sends the acknowledgement message 8. Note that the ACU certification ensures that the service key K_S will remain concealed in the ACU and thus never be sent to other users.

7^-: $B \rightarrow$ All: Subscribe answer $\langle S, \text{false}, \{N_{ab}, -\}_{K_a^p} \rangle$

8: ? $\rightarrow B$: Subscribe ack $\langle \{A', S\}_{K_a^s} \rangle$

It contains A's signature. B checks whether $D(K_a^p, \{A', S\}_{K_a^s}) = \langle A', S \rangle$ and keeps $\{A', S\}_{K_a^s}$ as a proof of delivery.

Broadcasting:

The broadcasting of the video service can then be a sequence of messages of the form:

repeat select a new control word CW;
send CW encrypted with K_S (message 9 below);
send a couple of images scrambled with CW (message 10 below);
until the service is over.

9: $B \rightarrow$ All: Send control word $\langle S, \{CW\}_{K_S} \rangle$
ACUs knowing K_S get $CW = D(K_S, \{CW\}_{K_S})$ and keep it in the (decoder of the) hosting set top box.

10: $B \rightarrow$ All: Send image $\langle S, \{\text{image}\}_{CW} \rangle$
The (decoders of the) set top boxes knowing the CW descramble the image as $D(CW, \{\text{image}\}_{CW})$.

In practice, the control words have to be sent a bit more ahead of the scrambled images, because the decryption of $\{CW\}_{K_S}$ takes some time, but we do not consider this issue here.

3. Formal specification

The specification has been written in LOTOS, which is a standardized formal description language suitable for the description of distributed systems. It is made up of two components:

- A process algebra, mostly inspired by CCS [Mil 89] and CSP [Hoa 85], with a structured operational semantics. It describes the behaviour of processes and their interactions. LOTOS has a rich set of operators (multiway synchronization and abstraction like in CSP, disabling, …), and an explicit internal action like in CCS. LOTOS is briefly introduced in the appendix.
- An algebraic datatype language, ACT ONE [EM85]. A type is defined by its signature (sorts + operations on the sorts) and by equations to give a meaning to the operations.

3.1. Behaviour

The LOTOS specification models both the Equicrypt system and the environment that it interacts with as two processes `EquicryptSystem` and `Environment` (fig. 1). The `EquicryptSystem` is composed of four main LOTOS processes, modelling its three main components (TTP, SPvs and STUs) and the common broadcast channel.

The service provider is split into a process that handles control words and scrambles images, and a process handling the subscriptions (fig. 2).

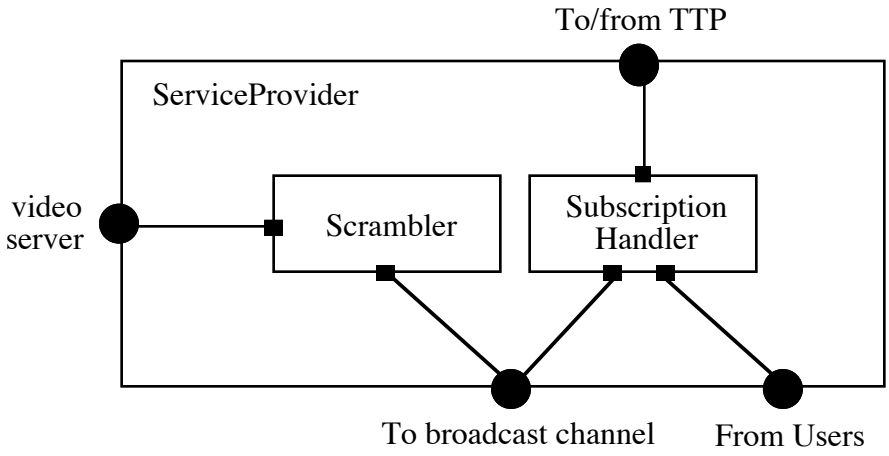

Fig. 2: The structure of the Service Provider

The set top unit contains a descrambler and the ACU, itself decomposed into a process handling registration, one handling subscription to new services and one decoding control words for subscribed services (fig. 3). New service keys are passed through gate `skey`, new subscriptions are notified to the descrambler via `nsvc` and `dcw` is used by the descrambler to ask for decoding of newly received control words.

Finally, the trusted third party is split into two processes dealing resp. with users and providers (fig. 4). Information about registered users is passed through gate `tup` for consultation by providers.

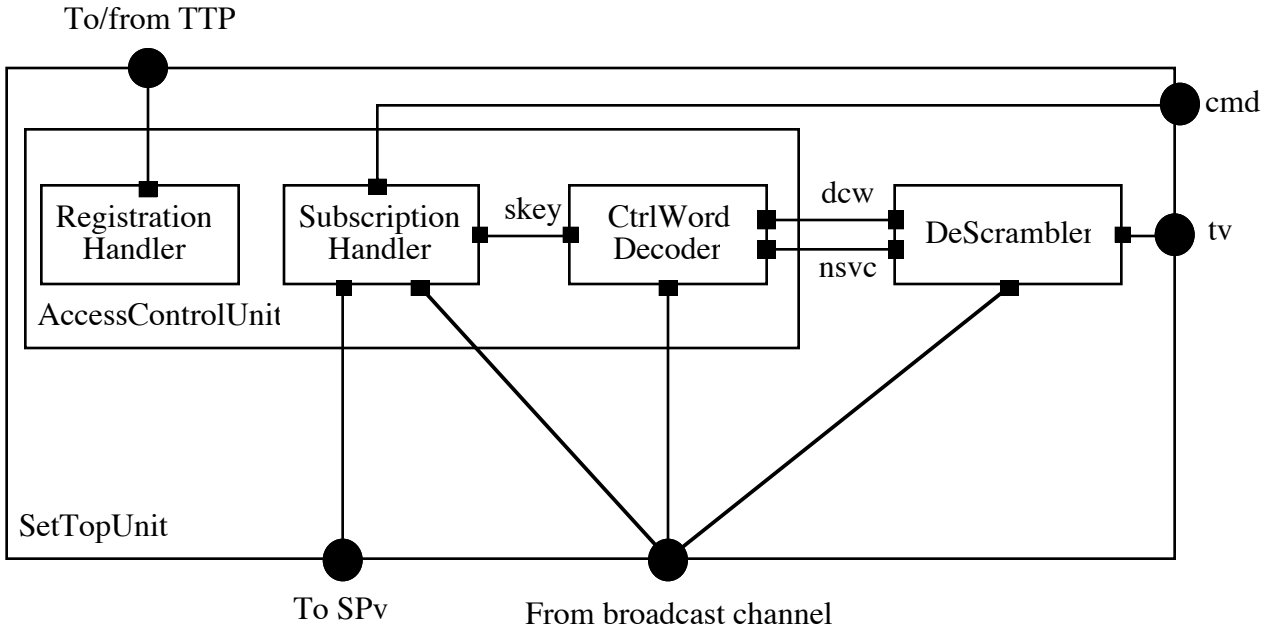

Fig. 3: The structure of the Set Top Unit of the User

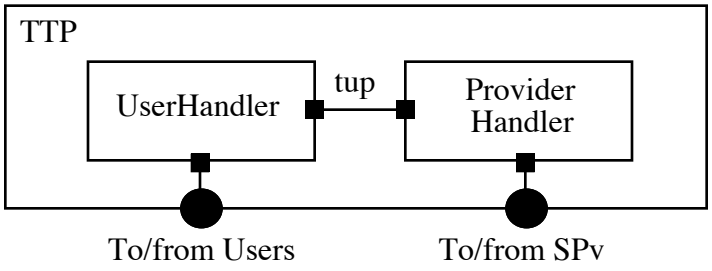

Fig. 4: The structure of the TTP

3.2. Data types

This specification has been written using data type language extensions, as offered by the APERO tools [Pec96] included in the Eucalyptus toolbox [Gar96] (fig. 5). The original text has to be processed by the APERO translator to get a valid LOTOS specification. This provides for a smaller and more readable specification and for some level of immunity w.r.t. underlying processing tools. Some types have been written from scratch, though, hence it was necessary to take tool restrictions explicitly into account. The other parts of the toolset will be explained later.

The abstract data types are composed of:

- *Base values*: identifiers, keys, transmitted data, etc., described as explicit enumerations.
- *Interaction primitives*: one APERO `recordtype` for each primitive. Gate multiplexing is used to model several similar communication channels as a single gate.
- *Tables*: needed for storing registered information in several processes, and defined using the APERO `tabletype` extension.
- *Encryption and decryption*: functions for each sort of encrypted value. These are modelled as abstract operations that are the reverse of each other. This allowed us to avoid modelling the actual algorithms such as RSA [RSA78]. Decryption with a bad key is handled explicitly and produces a distinguished value `xyzJunk` for each sort `xyz`, as described in ACT ONE below:

```
sorts Msg, EMsg, EK, DK          (* Message, encrypted message, encryption and
                                    decryption keys *)
opns  Match : EK, DK -> Bool     (* Define a binary predicate Match pairing those keys *)
E : EK, Msg -> EMsg              (* Define operations E and D for encryption/decryption *)
D : DK, EMsg -> Msg              (*  of some sort Msg with those keys *)
eqns forall m : Msg, ek : EK, dk : DK
ofsort  Msg
Match(ek, dk) => D(dk, E(ek, m)) = m ;      (* Decryption is the inverse of encryption
                                               when the keys match *)
Not Match(ek, dk) => D(dk, E(ek, m)) = msgJunk;
                                            (* otherwise, we get a junk message *)
```

3.3. Preliminary assessment of the specification

To gain confidence into the specification, it has been simulated with the XEludo tool [STS94] from the Eucalyptus toolbox in step-by-step non-symbolic execution mode. All the entities were present, in particular two service providers and two set top units. The broadcast channel was

modelled as an unbounded queue. Several early bugs were detected in our specification. The simulation was judged satisfactory, when it was possible to subscribe the two users simultaneously to both services and the images were descrambled correctly. This first phase of the work took one-man month and the LOTOS specification was about 1300 lines, including comments.

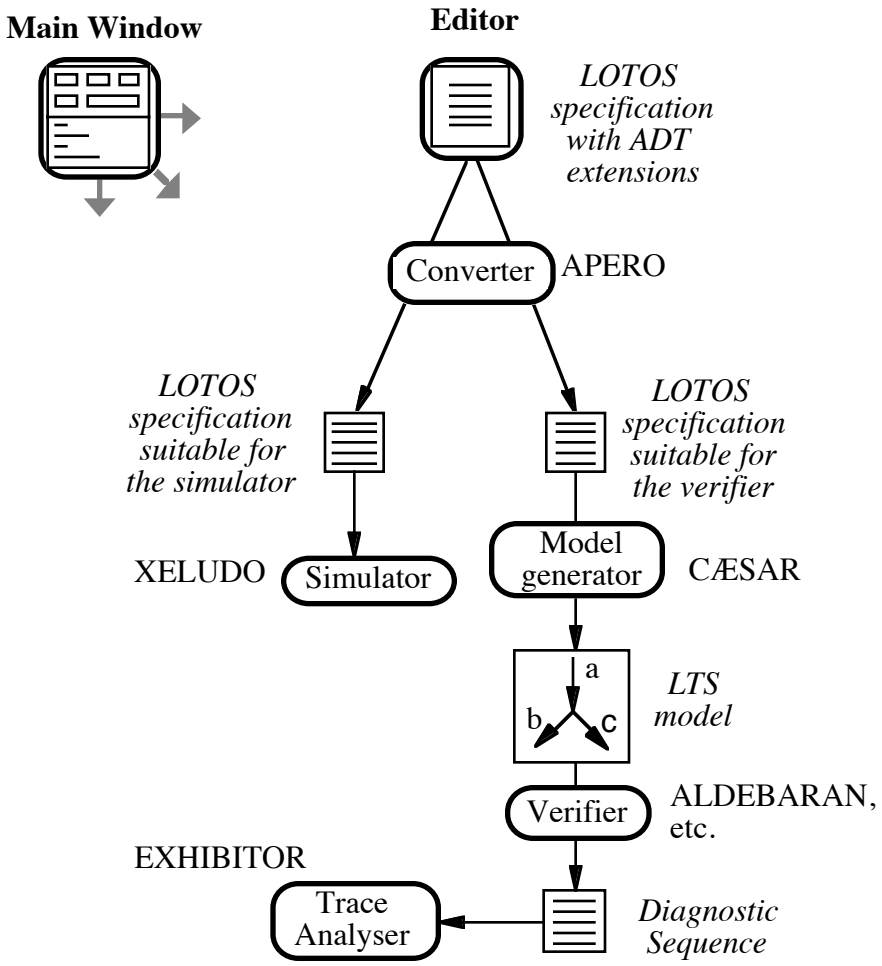

Figure 5: the Eucalyptus toolbox

4. Safety properties to be verified

The Equicrypt document itself did not mention specific properties to be satisfied. So, we had to think about a set of properties that was felt to capture the security of the system. We decided to focus mainly on authentication properties and came up with the following list of safety properties:

- P_1: Authentication of the user by the TTP during registration: when the TTP registers a user *A*, this user *A* must have started a registration procedure with the TTP.
- P_2: Authentication of the TTP by the user during registration: when the user *A* believes it is registered by the TTP, the TTP must have started a registration procedure with this user *A*.
- P_3: Authentication of the service provider by the user during subscription: when the user *A* believes it has successfully subscribed to a service provider *B*, the service provider *B* must have started a subscription procedure with *A'* (*A'* being the alias of *A*).
- P_4: Authentication of the user by the service provider during subscription: when the service provider *B* accepts the subscription of *A'*, the user *A* (that has alias *A'*) must have started a subscription procedure with *B*.
- P_5: Authentication of the TTP by the service provider: when the service provider *B* accepts the subscription of *A'*, the TTP must have given guarantees regarding the registration of user *A* (that has alias *A'*).
- P_6: When the service provider *B* receives a correctly signed subscription acknowledgement from *A'*, the user *A* (that has alias *A'*) must have started a subscription procedure with *B*.
- P_7: When a user successfully subscribes, it must have successfully registered.

For properties P_1 to P_5, the dual properties $P_1^{'}$ to $P_5^{'}$ should also hold. We give $P_1^{'}$ as an example:

- $P_1^{'}$: When the TTP decides *not* to register a user *A*, this user *A* must have started a registration procedure with the TTP.

Finally, we have no authentication property of the service provider *B* by the TTP, because there is no need for such an authentication, since the key requested by *B* is public.

5. Verification of the protocol

5.1. Model of an intruder

We want to model an intruder as a process that can mimic any attack a real-world intruder can perform. Thus our intruder process shall be able to:

- Eavesdrop on and/or intercept any message exchanged among the entities;
- Decrypt (parts of) messages that are encrypted with her own public key, and store them;
- Introduce fake messages in the system (a fake message can be an old message which is replayed or a new message built up from components of old messages, including components that she was unable to decrypt).

The intruder behaves in such a way that neither the receiver of a fake message, nor the sender of an intercepted message can notice the intrusion. In fact the intruder merely replaces the channel linking users and providers in the model.

In this paper, the intruder will only act on the insecure channel between the users and the providers, because the proposed Equicrypt protocol relies heavily on the hypothesis that the

communication channels with the TTP are secure. The registration phase of the protocol has been verified separately [GeL97a, GeL97b] with an insecure channel between the user and the TTP.

The intruder is parameterized with some initial knowledge, which should be realistic and give her enough power to act as a user when she communicates with a service provider, and act as a provider when she communicates with a user. In particular, we do *not* exclude the following two cases: (1) the intruder is a certified user, or (2) is (known to users as) a service provider. However, if the intruder is a certified user, we still consider that her ACU behaves as a certified ACU. For example, the ACU cannot disseminate a stored service key to other users. Therefore the initial knowledge of the intruder is as follows:

I, I': The identification and alias of the intruder.
K_i^p , K_i^s : The public and private (secret) keys of I.
N_{ia}, N_{ib}: Nonces that I can use in fake messages sent to A and B.
C_i: The certificate of I.
Ki_S: A service key I can use in fake messages sent to A.
K_a^p , K_b^p : The public keys of user A and service provider B.

In addition, the intruder also knows the user aliases and the service identifiers used in the specification, either because they are well-known, or because they have been observed in earlier runs of the protocol. The user identifiers are not needed for the verification of the subscription phase of the protocol and are therefore omitted from the intruder's knowledge.

During a protocol run, the intruder can increase her knowledge base. This is modelled by additional parameters such as sets of (parts of) encrypted messages that she has been unable to decrypt, and sets of (parts of) plaintext messages that she has been able to decrypt (or were sent in plaintext).

By having a single nonce to talk to A and a single nonce to talk to B, our intruder can only take part in a single run of the protocol. Therefore, any form of attack that relies on several attempts cannot be exhibited by this intruder.

Furthermore, we assume that our intruder cannot break the public key cryptosystem by getting the message in clear from the encrypted message and the public encryption key, or forging a signed message from the message in clear and the public decryption key. In our model, this would mean for example guessing m from $\{m\}_K$ and K. Note that LOTOS[1] easily provides processes that transgress this rule, and thus break any cryptosystem:

```
process GuessMsg (emsg:EMsg, ek:EK) : exit(Msg) :=
   (* 'emsg' is an encrypted message, and 'ek' is the encryption key *)
choice msg:Msg □ [E(ek, msg) = emsg] → exit(msg)
(* pick an arbitrary message 'msg' such that its encryption with 'ek'
   gives 'emsg' and return it *)
endproc
```

[1] A brief introduction of the LOTOS syntax is given in the appendix.

When building, for verification purposes, an intruder process that tries to break the access control mechanisms, care must be taken to avoid these kinds of unrealistic behaviours. In practice one can interpret them as trying all possible encodings until the correct one is found, which is precisely the computationally unfeasible operation on which security relies. For the same reason, we assume that the intruder cannot guess nonces.

The structure of the LOTOS intruder process is shown below. Note that we have just provided the parts dealing with the interceptions of subscribe requests (message 4) sent by users and the generations of fake subscribe requests sent to service providers. The "…" notation is not part of the LOTOS syntax; it replaces some parameters that we have omitted here for clarity. We have also removed the typing information for the same reason.

```
process Intruder [U,P] (lnr,lsv,lti,letk,lsek,svcdk,...) :noexit :=
  (* U and P are interfaces with users and providers respectively;
     lnr is the list of known user aliases (including the intruder's alias);
     lsv is the list of known service identifiers;
     lti is the list of known tickets (i.e. nonces that she has decrypted,
       initialized with her own nonces);
     letk is the list of known encrypted tickets (i.e. the nonces that she
       cannot decrypt);
     lsek is the list of known public keys of service providers (including
       hers to allow her to act as a service provider);
     svcdk is her private key
     ... *)

U ?subr:SubscribeRequest;  (* intercept a subscription request *)
([D(svcdk,ETicket(subr)) eq TicketJunk] →
      (* if the ticket cannot be decrypted then *)

    Intruder [U,P] (lnr,lsv,lti,Add(ETicket(subr),letk),lsek,svcdk,...)
      (* add this encrypted ticket to the list of encrypted tickets *)
 □
 [not(D(svcdk,ETicket(subr)) eq TicketJunk)] →
      (* if the ticket can be decrypted then *)

    Intruder [U,P] (lnr,lsv,Add(D(svcdk,ETicket(subr)),lti),letk,lsek,svcdk,
                    ...)
      (* add the decrypted ticket to the list of known tickets *)
)
□ (* or *)
(choice un,pv,sv,sek,ti □
      [(un IsIn lnr) and (sv IsIn lsv) and
       ((E(sek,ti) IsIn letk) or ((sek IsIn lsek) and (ti IsIn lti)))] →
     (* pick any user alias (say 'un') in lnr, any service provider id (say
        'pv'), any service id (say 'sv') in lsv, any encrypted ticket (say
        'E(sek,ti)') either in letk or by encrypting any ticket (say 'ti')
        from lti with any key (say 'sek') from lsek *)
   P !SubReq(un,pv,sv,E(sek,ti));
      (* send the fake message just built *)
   Intruder [U,P] (lnr, lsv, lti, letk, lsek, svcdk,...)
      (* don't change the data base *)
)
```

```
□ (* or *)
... (* ditto with all other messages *)
endproc (* Intruder *)
```

5.2. Formalizing the properties

To model the properties presented in section 4, it is necessary to observe some states of all the entities (user, provider and TTP). To achieve this, we will add special events to be executed in those states. Here are some of them used in the subscription procedure. Others are used in the registration procedure:

`U_start_sub!A!B!S`	The user *A* starts a subscription procedure with the service provider *B* requesting service *S*. This event is executed by the user *A* just before it sends message 4 to *B*.
`U_sub!A!B!S`	The user *A* believes it has successfully subscribed to service *S* provided by *B*. This event is executed by the user *A* just after it has received a positive message 7 from *B* with the expected nonce.
`U_Rsub!A!B!S`	The user *A* believes it has not successfully subscribed to service *S* provided by *B*. This event is executed by the user *A* just after it has received a negative message 7 from *B*, or a positive message 7 with a wrong nonce.
`P_start_sub!A'!B!S`	The service provider *B* has started a subscription procedure with *A*' requesting service *S*. This event is executed by the service provider *B* just after it has received message 4 from *A*'.
`P_sub!A'!B!S`	The service provider *B* accepts the subscription of *A*' regarding service *S*. This event is executed by the service provider *B* just before it sends a positive message 7 to *A*'.
`P_Rsub!A'!B!S`	The service provider *B* refuses the subscription of *A*' regarding service *S*. This event is executed by the service provider *B* just before it sends a negative message 7 to *A*'.
`P_sub_ack!A'!B!S`	The service provider *B* receives a correctly signed subscription acknowledgement from *A*' regarding service *S*. This event is executed by the service provider *B* just after it receives a correct message 8 from *A*'.
`T_sub!A!T`	The TTP *T* gives registration guarantees for user *A*. This event is executed by the TTP *T* just before it sends a positive message 6'.
`T_Rsub!A!T`	The TTP *T* does not give registration guarantees for user *A*. This event is executed by the TTP *T* just before it sends a negative message 6'.

The properties can now be expressed solely in terms of these events. We give properties P_3 and P_6 as examples below:

P_3: For all $A \neq I$, A', $B \neq I$ and S such that A' is the alias of user A, `U_sub!A!B!S` must be preceded by `P_start_sub!A'!B!S`.

P_6: For all $A \neq I$, A', $B \neq I$ and S such that A' is the alias of user A, `P_sub_ack!A'!B!S` must be preceded by `U_start_sub!A!B!S`.

These properties are similar to the correspondence properties of [WoL93].

A property such as "`U_sub!A!B!S` must be preceded by `P_start_sub!A'!B!S`" can be easily modelled by the graph, called a Labelled Transition System (LTS), depicted on figure 6, which could also be modelled by the following two-state LOTOS process:

```
process Precedence[U_sub,P_start_sub] (A,A',B,S): noexit :=
P_start_sub!A'!B!S; run[U_sub,P_start_sub] (A,A',B,S)

where process run[U_sub,P_start_sub]: noexit :=
      P_start_sub!A'!B!S; run[U_sub,P_start_sub] (A,A',B,S)
      []
      U_sub!A!B!S; run[U_sub,P_start_sub] (A,A',B,S)
endproc (* run *)

endproc (* Precedence *)
```

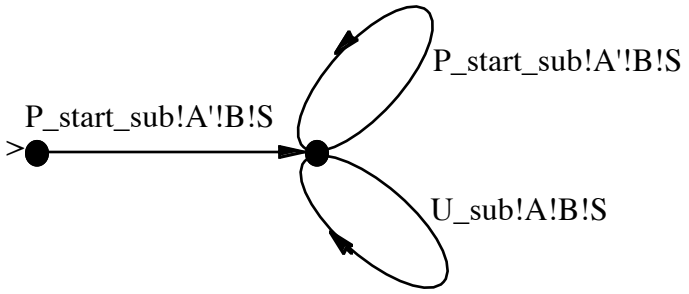

Figure 6: LTS modelling "`U_sub!A!B!S` must be preceded by `P_start_sub!A'!B!S`"

To model a property like P_3 in LOTOS, it suffices to interleave as many instances of this `Precedence` process as there are valid combinations of A, A', B and S in the specification (in particular such that A' is the alias of A). If there are n possible combinations, this leads to an LTS composed of 2^n states and $3n \times 2^{n-1}$ transitions.

5.3 The verification

We have used the CADP package [FGK+96] included in the Eucalyptus toolbox to carry out the verification (figure 5). The first step consists of using the Cæsar tool to generate an LTS from the LOTOS specification. To be able to generate a *finite-state* LTS of *reasonable size*, some simplifications were required. For example, to verify the properties listed in section 4, the parts of the protocols dealing with the scrambling and descrambling of the images have been left out. Also, an environment has been added to restrict the behaviour to a single run of the protocols (one registration and one subscription). Finally, the broadcast channel has been bounded. Several variants of this simplified specification with an intruder have been written. For example: a specification with two service providers offering distinct as well as identical services. Let us consider one such configuration and call it `spec`.

The second step consists of using the Aldebaran tool to minimize the resulting graph. The first minimization is always done modulo the strong bisimulation equivalence, which preserves all the properties of the graph. We call the reduced LTS `spec.bsim`. Our security properties being all safety properties, the minimization can be further improved modulo the safety equivalence [BFG+91], which preserves all the properties expressible in Branching time Safety Logic (BSL). The resulting graph is called `spec.safety`.

Not all the observable actions are relevant to verify the properties. In particular, our properties only rely on some special actions that have been described in section 5.2. Every property was separately modelled as a reference LTS generated from a simple LOTOS process and containing these special actions only (e.g. the LTS modelling property P_i is called `Pi.safety`), and the `spec.safety` LTS was checked against such a property by verifying the safety preorder relation [BFG+91] between `spec.safety` and `Pi.safety`, while hiding irrelevant actions. Formally, the safety preorder ($\leq_s$) is the preorder that generates the safety equivalence ($\sim_s$), and is nothing else than the weak simulation preorder:

Consider a LTS = $\langle S, A, T, s_0 \rangle$ where S is the set of states, A the alphabet of actions (with i denoting the internal action), T the set of transitions and s_0 the initial state.

A relation $R \subseteq S \times S$ is a weak simulation iff $\forall\ \langle B_1, B_2 \rangle \in R$,$\forall\ a \in A$:

$$\text{if } B_1 \stackrel{i^* a}{\Rightarrow} B_1' , \quad \text{then } \exists\ B_2' \text{ such that } B_2 \stackrel{i^* a}{\Rightarrow} B_2' \text{ and } \langle B_1' , B_2' \rangle \in R$$

$\text{Sys}_1 = \langle S_1, A, T_1, s_{0_1} \rangle$ can be simulated by $\text{Sys}_2 = \langle S_2, A, T_2, s_{0_2} \rangle$, denoted $\text{Sys}_1 \leq_s \text{Sys}_2$, iff there exists a weak simulation relation $R \subseteq S_1 \times S_2$, such that $\langle s_{0_1}, s_{0_2} \rangle \in R$

Informally, "behaviour $\leq_s$ property" means that the behaviour (exhibited by the system) is allowed (i.e. can be simulated) by the (safety) property.

Two systems P_1 and P_2 are safety equivalent iff $P_1 \leq_s P_2$ and $P_2 \leq_s P_1$.

When a property is not verified, Aldebaran produces a diagnostic sequence. However, this sequence is usually of little help as such, because it only refers to the few non hidden actions that were kept for their relevance to express the properties. We call it the abstract diagnostic sequence. To circumvent this difficulty and get a detailed sequence (i.e. with all actions visible) that can clearly identify the scenario of the intruder's attack, we have encoded this abstract diagnostic sequence in a format suitable for input to the Exhibitor tool, which was then instructed to find the detailed sequence (allowed by the specification) matching the abstract one.

For a typical configuration with one user, two service providers (B offering service S, and C offering services S and S'), one TTP and an intruder offering both services S and S', the generated LTS was composed of 786,681 states and 4,161,795 transitions. It took 20 hours of CPU time on a Sun Ultra-2 workstation running Solaris 2.5 with 800 Mbytes of RAM. After minimization with the strong bisimulation, the LTS still had 69,754 states and 520,633 transitions. The minimization was carried out in 20 minutes of CPU time on the same workstation. The minimization by the safety equivalence failed due to the size of the LTS, but fortunately, it was possible to check all the properties on this graph.

This second phase of the work that consisted of finding a verification approach, adding an intruder process, specifying the properties and verifying them, took about 2 man-months.

5.4 Results of the verification

Given that our intruder can only act on the channels linking users and providers, it turns out that properties P_1, P_2 and P_5 are satisfied. A deep analysis of the registration phase of the protocol is beyond the scope of this paper and can be found in [GL97b]. On the other hand, properties P_3, P_4 and P_6 are not verified in the presence of our intruder. The following two scenarios translated from those provided by Exhibitor will illustrate why these properties are falsified. Finally, P_7 is verified.

First scenario

When checking property P_6, Aldebaran discovered it was not satisfied and reported a (abstract) diagnostic sequence of 37 steps mainly composed of hidden actions and leading to a state where `P_sub_ack!A'!C!S` was enabled in the system but not allowed by the safety property. This abstract sequence was then translated into a precise search pattern given to the Exhibitor tool that found several matching sequences in `spec.bsim` where all the actions are visible. The smallest sequence found was composed of 31 transitions. The part of it modelling exchanges between users and service providers (thus omitting the exchanges with the TTP for clarity) is explained below and depicted on figure 7.

For this attack to succeed, the intruder does not need to be registered; she just needs to know the public keys of two service providers *B* and *C*. We consider a user *A* subscribing successfully to service *S* provided by server *B*. This subscription runs *implicitly* in parallel with the attack described below. Now suppose that another service provider *C* also offers this service *S*, and suppose an intruder is aware of that (for example, the intruder might be *C* itself, or *C*'s accomplice). The story goes as follows: the intruder can copy the subscription request sent by *A* to *B*, namely the message:

4: ? → *B*: Subscribe request $\langle A', S, \{N_{ab}\}_{K_b^p} \rangle$

Then, the intruder sends a fake message to service provider *C*, which is basically the above message where the nonce has been changed and encrypted with *C*'s public key:

4: ? → *C*: Subscribe request $\langle A', S, \{N_{ic}\}_{K_c^p} \rangle$

C then gets N_{ic} and starts its subscription procedure with *A*', namely it executes `P_start_sub!A'!C!S`, and exchanges messages 5 and 6^+ with the TTP. This successful exchange leads *C* to commit on a subscription with *A*': *C* executes `P_sub!A'!C!S`. Then *C* sends message:

7: *C* → All: Subscribe answer $\langle S, \text{true}, \{N_{ic}, K_S'\}_{K_a^p} \rangle$

I intercepts this message and sends a fake subscribe acknowledgement *containing A's signature* to *C*, which is a copy of the corresponding subscribe acknowledgement that *A* sent earlier to *B*, namely:

8: ? → *C*: Subscribe ack $\langle \{A', S\}_{K_a^s} \rangle$

Finally C executes `P_sub_ack!A'!C!S.`

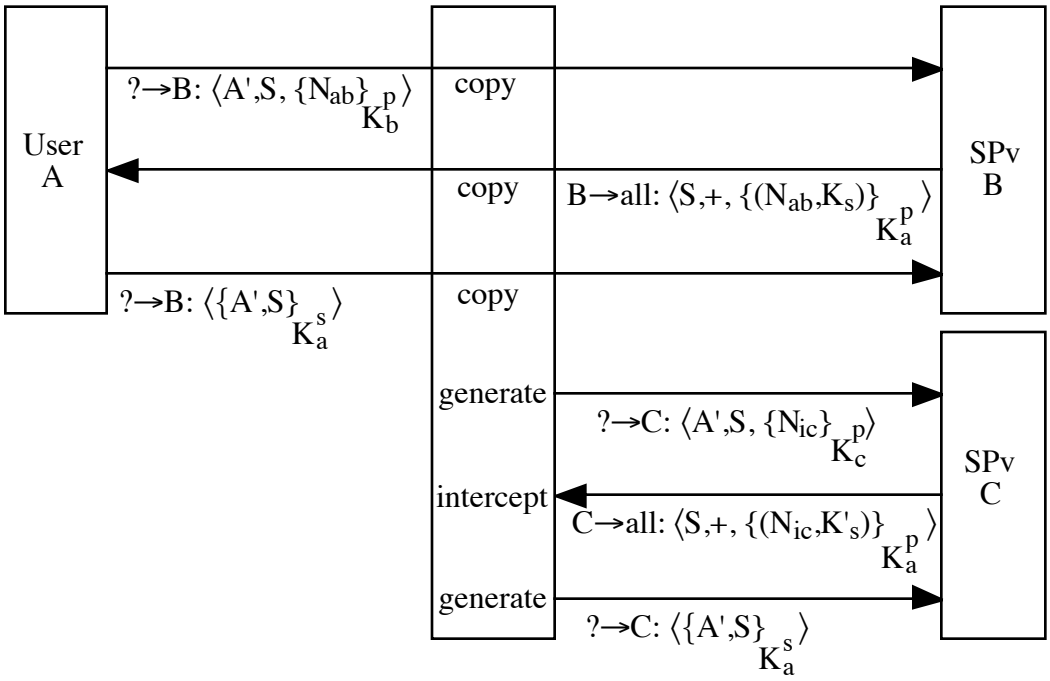

Figure 7: First attack

Clearly property P_4 and P_6 are not fulfilled, because events `P_sub!A'!C!S` and `P_sub_ack!A'!C!S` are not preceded by the event `U_start_sub!A!C!S`. The consequence is that the service provider *C* can claim money from *A* for the subscription, because it possesses a signed subscribe acknowledgement from *A* for this service *S*.

A first idea was to encrypt *S* in messages 4 (the subscribe request) and 7 (the subscribe answer), but this does not make sense for well-known service identifiers, which can always be tried by the intruder. A more effective way to make this attack impossible consists of adding the name of the service provider in the signed part of message 8 (the subscribe acknowledgement). However, this would not be enough to satisfy P_4. The only way to fulfil P_4 is actually to sign message 4 (the subscribe request).

Second scenario

Among the attacks we have found, we present another one which falsifies P_3 (fig. 8): *A* starts a subscription procedure with *B*, namely it executes `U_start_sub!A!B!S`. Then *A* sends the message:

4: ? → *B*: Subscribe request $\langle A', S, \{N_{ab}\}_{K_b^p} \rangle$

I intercepts it, replaces the service identifier by *S*' and sends the fake message:

4: ? → *B*: Subscribe request $\langle A', S', \{N_{ab}\}_{K_b^p} \rangle$

B then gets N_{ab} and starts its subscription procedure with *A'*: namely it executes `P_start_sub!A'!B!S'`, and exchanges messages 5 and 6+ with the TTP. This successful exchange leads *B* to commit on a subscription with *A'*: *B* executes `P_sub!A'!B!S'`. Then *B* sends this message:

7: *B* → All: Subscribe answer $\langle S', \text{true}, \{N_{ab}, K_S'\}_{K_a^p} \rangle$

I intercepts it, replaces the service identifier by *S* and sends the fake message:

7: *B* → All: Subscribe answer $\langle S, \text{true}, \{N_{ab}, K_S'\}_{K_a^p} \rangle$

Finally *A* decrypts this message, finds its nonce N_{ab} and therefore commits to subscription by executing `U_sub!A!B!S`.

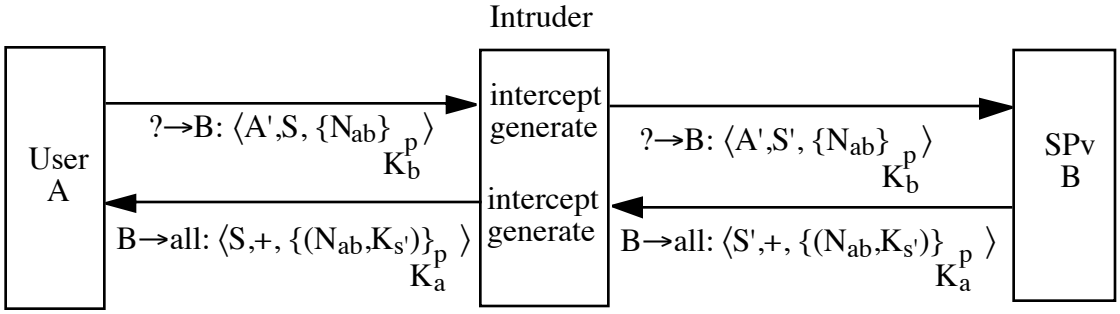

Figure 8: Second attack

Clearly the property P_3 is not fulfilled because event `U_sub!A'!B!S` is not preceded by the event `P_start_sub!A!B!S`, but by `P_start_sub!A!B!S'`. The consequence is that the user has received a service key which cannot be used to decrypt service *S*, but will send an acknowledgement.

Here again, the signature of message 4 (the subscribe request) suffices to make this attack impossible.

A third attack similar to the one found recently in the Needham-Schroeder authentication algorithm [Low95, Bol96, Low96] has been reported in [LBK+96].

The overall resources to complete the specification, the verification and the correction of the subscription part of the Equicrypt protocol are estimated to be around 4 man-months, not including reports and published papers.

6. Conclusion and future work

We have had the opportunity to apply LOTOS to a protocol that was still under design, and on which our work has had a real impact. The revision of the subscription procedure by signing the subscription request turns out to fulfil all our properties. We hope that this non trivial case study

illustrates the power of our approach, which is further detailed in [GL97a]. The registration phase of the protocol has also been studied and fixed in [GL97b].

We have shown how attacks can be discovered by adding an intruder process, which is simple and can mimic all behavioural real-world attacks that are neither cryptographic nor repetitive attacks. We have shown how security properties can be modelled and checked as safety properties by the safety preorder relation.

Our method is very effective in finding subtle bugs, but it should be clear that the correctness guarantees we can provide when no bug is found are within the scope of our hypotheses. We cannot prove the correctness of the protocol in general, but only of some reasonably complex configurations, where the numbers of users and of service providers are bounded, and where the ranges of possible values for variables are kept finite.

There exist ways to extend the method. In a simpler case [Low96], an additional induction proof has been provided to extend the correctness guarantee to an arbitrary number of involved entities. Another possible approach, proposed recently in [Bol97], is based on an abstraction function and automates the computation of a correct (finite) abstract model of the system. Finally, we do not ensure any sort of completeness of our security properties. We have focused on authentication properties and some others but, for example, we have not considered non-repudiation properties. Methods to automate the definition of security properties would therefore be desirable. Some work in this direction is proposed in [AbG97].

Acknowledgements

This work has been partially supported by the Commission of the European Union (DG XIII) under the ACTS AC051 project OKAPI: “Open Kernel for Access to Protected Interoperable Interactive Services”. The development of the Apero tool and the user interface of the Eucalyptus toolset have been partially supported by the Commission of the European Union (DG III) under project ISC-CAN-65 Eucalyptus-2: “A European/Canadian LOTOS Protocol Tool Set”. Finally we are grateful to the anonymous referees for their valuable comments.

References

[AbG97] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols - The Spi Calculus. *In: Proc. of the 4th ACM Conference on Computer and Communications Security*, 1997.

[BFG+91] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez and J. Sifakis. Safety for Branching Time Semantics. *In: 18th ICALP*, Berlin, July 1991. Springer-Verlag.

[BoB87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14 (1) 25-59 (1987).

[Bol96] D. Bolignano. Formal verification of cryptographic protocols. *In: Proc. of the 3rd ACM Conference on Computer and Communication Security,* 1996.

[Bol97] D. Bolignano. Towards a Mechanization of Cryptographic Protocol Verification. *In: Proc. of CAV 97, LNCS 1254*, Springer-Verlag, 1997.

[BAN90] M. Burrows, M. Abadi and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8, 1990.

[ChG90] P. Chen and V. Gligor. On the Formal Specification and Verification of a Multiparty Session Protocol. In: Proc. of the *IEEE Symposium on Research in Security and Privacy*, 1990.

[DEK82] D. Dolev, S. Even, and R. Karp. On the Security of Ping-Pong Protocols. *Information and Control*, pp. 57-68, 1982.

[DoY83] D. Dolev, and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198-208, March 1983.

[EM85] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification 1, Equations and Initial Semantics. In: W. Brauer, B. Rozenberg, A. Salomaa, eds., *EATCS , Monographs on Theoretical Computer Science*, Springer Verlag, 1985.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. *In: R. Alur and T. Henzinger, eds, Proc. of the 8th Conference on Computer-Aided Verification* (New Brunswick, New Jersey, USA), Aug. 1996.

[Gar96] H. Garavel. An overview of the Eucalyptus Toolbox. *In: Proc. of the COST247 workshop* (Maribor, Slovenia), June 1996.

[GBM96] J. Guimaraes, J.-M. Boucqueau and B. Macq. OKAPI: a Kernel for Access Control to Multimedia Services based on Trusted Third Parties. *In: Proc. of ECMAST 96 – European Conference on Multimedia Applications, Services and Techniques* (Louvain-la-Neuve, Belgium), pp. 783-798, May 1996.

[GL 97a] F. Germeau, G. Leduc. Model-based Design and Verification of Security Protocols using LOTOS. Proc. of the DIMACS Workshop on Design and Formal Verification of Security Protocols, Rutgers University, NJ, USA, Sept. 97, 22 p.

[GL97b] F. Germeau, G. Leduc. A computer-aided design of a secure registration protocol. Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE/PSTV' 97, Chapman & Hall, London (1997), 145-160.

[GQ88] L. Guillou, J.-J. Quisquater. A Practical Zero-knowledge Protocol Fitted to Security Microprocessor Minimizing both Transmission and Memory. *In. Proc. of Eurocrypt'88*, Springer-Verlag, LNCS 330, 123-128.

[Hoa 85] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall International, 1985.

[ISO8807] ISO/IEC. Information Processing Systems – Open Systems Interconnection – LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, February 1989.

[Kem89] R. Kemmerer. Using Formal Methods to Analyse Encryption Protocols. IEEE *Journal on Selected Areas in Communications*, 7(4):448-457, 1989.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. Three Systems for Cryptographic Protocol Analysis. *Journal of Cryptology*, 7(2):14-18, 1989.

[LBQ+96] S. Lacroix, J.-M. Boucqueau, J.-J. Quisquater and B. Macq. Providing Equitable Conditional Access by Use of Trusted Third Parties. *In: Proc. of ECMAST 96 – European Conference on Multimedia Applications, Services and Techniques* (Louvain-la-Neuve, Belgium), pp. 763-782, May 1996.

[LBK+96] G. Leduc, O. Bonaventure, E. Koerner, L. Léonard, C. Pecheur, D. Zanetti. Specification and verification of a TTP protocol for the conditional access to services. In: *Proc. of 12th J. Cartier Workshop on "Formal Methods and their Applications: Telecommunications, VLSI and Real-Time Computerized Control System"*, Montreal, Canada, 2-4 Oct. 96.

[Low95] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol, *Information Processing Letters*, 56:131-133, 1995.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In: T. Margaria and B. Steffen (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1055, Springer-Verlag, 1996.

[MCF87] J. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13(2), 1987.

[Mea92] C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1992.

[Mea 94] C. Meadows. The NRL Protocol Analyser: An Overview. *Journal of Logic Programming* 1994:19, 20:1-679.

[Mea95] C. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In: Proc. of Asiacrypt 94, LNCS 917, 1995, pp. 133-150.

[Mil 89] R. Milner. Communication and Concurrency. Prentice-Hall Intern., London, 1989.

[Pec96] C. Pecheur. Improving the Specification of Data Types in LOTOS. *Doctoral dissertation, University of Liège*, July 1996.

[RSA78] R. Rivest, A. Shamir and L. Adleman. On a Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communication of the ACM*, vol. 21, pp. 120-126, Feb. 1978.

[STS94] B. Stepien, J. Tourrilhes and J. Sincennes. ELUDO: The University of Ottawa Toolkit. *Technical Report, University of Ottawa, 1994*. Obtainable by FTP at `lotos.csi.uottawa.ca`.

[Var89] V. Varadharajan. Use of Formal Technique in the Specification of Authentication Protocols. *Computer Standards and Interfaces*, 9:203-215, 1989.

[WoL93] Woo and S. Lam. A Semantic Model for Authentication Protocols. *In : Proc. of IEEE Symposium on Research in Security and Privacy*, 1993.

Appendix: Overview of the LOTOS operators

- **stop** is an inactive (deadlocked) process.
- $g o_1 ... o_n [SP]; P$ (action-prefixing) is a process that first performs an (observable) action on gate g and then behaves like P. The tuple $o_1 ... o_n$ determines the data exchanged during the synchronisation: either data sent, by $!tx$, or data (of sort s) received, by $?x:s$. The variables declared in $o_1 ... o_n$ to receive data can appear in the selection predicate (i.e. the boolean expression) *SP*. Data can be received only if they verify *SP*.
- **i**; P is a process that first performs an internal action and then behaves like P.
- **exit**$(e_1,...e_n)$ is a process that successfully terminates. It performs an action on gate δ and then turns into **stop**. The tuple $e_1,...e_n$ determines the data transmitted to the subsequent process (see the enabling operator).

- $P_1 \Box P_2$ (choice) is a process that can behave either like P_1 or like P_2. The choice is resolved by the first process which performs an action. Notice that internal actions also resolve the choice.
- P_1 |[Γ]| P_2 is the parallel composition of P_1 and P_2 with synchronisation on the gates in γ.
- **hide** γ **in** P hides actions of P occurring at gates present in the set γ, i.e. renames them **i**.
- P_1 » **accept** $x_1{:}s_1,\ldots x_n{:}s_n$ **in** P_2 (enabling) is the sequential composition of P_1 and P_2, i.e. P_2 can start when P_1 has terminated successfully. A process terminating successfully can transmit data to its successor: the tuple $e_1,\ldots e_n$ associated with **exit** determines the data values transmitted and **accept** $x_1{:}s_1,\ldots x_n{:}s_n$ **in** specifies the data P_2 expects to receive.
- P_1 [> P_2 (disabling) allows P_2 to disable P_1 provided P_1 has not terminated successfully.
- [SP]→ P (guard) behaves like P if the guard SP is true and like **stop** otherwise.
- **let** $x_1{=}tx_1,\ldots x_n{=}tx_n$ **in** P (instantiation) instantiates the free variables $x_1\ldots x_n$ in P.
- **choice** $x_1{:}s_1,\ldots x_n{:}s_n \Box P$ (choice over values). Assuming P depends on the variables $x_1\ldots x_n$, (of sorts $s_1\ldots s_n$), **choice** $x_1{:}s_1,\ldots x_n{:}s_n \Box P$ offers a choice between the processes $P(tx_1\ldots tx_n)$ for all the combinations of values $(tx_1\ldots tx_n)$ of sorts $(s_1\ldots s_n)$. For example, **choice** $x{:}Nat \Box P(x)$ means $P(0) \Box P(1) \Box P(2) \Box \ldots$