A Formal Framework for Authentication

Colin Boyd

Communications Research Group
Electrical Engineering Laboratories
University of Manchester, Manchester M13 9PL, UK

Email : colin@uk.ac.man.ee.comms

Abstract

This paper presents an abstract formal framework for authentication using the standardised formal description technique LOTOS. The purpose of this framework is to investigate the abstract definition of authentication in a standardised formal language and to illustrate how to put some recent standardisation activities on a formal basis. Two authentication protocols are specified as examples of how the framework may be used in the specification and analysis of authentication.

Keywords: Authentication protocols, formal models, security standards

1 Introduction

Authentication is now widely recognised as a prerequisite for establishing secure communications in a distributed system. There has been considerable interest in recent years in developing formal models to analyse authentication. By making precise what is required for authentication we can be sure what is achieved when authentication takes place and check whether specific protocols satisfy the definition. The consequences of authentication can further be analysed in a rigorous fashion. Important recent work by Burrows, Abadi and Needham [4] has defined a special logic to analyse authentication protocols. An alternative state-based methodology has been developed by Meadows [11]. Both these models have been used not only to put previously known protocol weaknesses in a formal setting, but also to find previously unrecognised weaknesses.

There are developments inside the International Organisation for Standardisation (ISO) towards providing general frameworks for security, one of these being the Authentication Framework [5]. This is quite a different document from, for example, the CCITT X509 Directory Authentication Framework, [6], which defines specific authentication mechanisms using particular protocols. Instead the new Authentication Framework describes a range of different authentication services and classifies mechanisms for providing these services.

In this paper a formal framework for authentication is developed which encompasses the basic elements described in the Authentication Framework [5] and is sufficiently general to describe a wide range of protocols. Unlike previous approaches, a basic aim is to give an abstract definition of what constitutes authentication of a message. In addition no special notation is introduced, but an existing standardised formal language is used. This allows the authentication process to be defined as an integral part of the complete protocol definition, using a language specially designed for the purpose. There is an official policy within ISO

Y. Deswarte et al. (eds.), *Computer Security - ESORICS 92*

to move towards use of formal definitions in Open Systems standards, although there are few examples to date. This paper, while not constituting a complete formal version of the Authentication Framework [5], illustrates how formal definitions could be incorporated into it. It should also be emphasised, however, that the present approach is limited to deciding whether authentication of specific messages is achieved rather than building up representations of beliefs over a whole protocol as in the models of [4] and [11].

The language used in this paper to describe the framework is the Language for Temporal Ordering Specification (LOTOS) which has been standardised by ISO [9]. There are several benefits which come from this choice of language in this particular case. Firstly it comes with a sound theoretical basis and is ideal in its expressiveness to describe communications protocols. Secondly, many authentication protocols are part of OSI protocols, and LOTOS was specifically designed for the description of these. Indeed some OSI protocols are already specified as a whole in LOTOS [1]. Thirdly, as the framework becomes more mature, tools are available to verify correctness and analyse the framework and the protocols which conform to it [10]. Such reasons have already led to LOTOS being used for security protocol analysis [15].

An introduction to the language LOTOS is given in [2]. The main points of this paper can be understood without specific knowledge of the language. At various points the formal language is annotated with comments in parentheses thus: (*Comment*). The framework attempts to make a generic definition of what is meant by authentication by describing processes which perform checking of the particular parameters required for authentication. These parameters are described via a formal abstract data type which can be made specific (*actualised* in the language of LOTOS) in the case of a particular protocol. Thus the framework does not constitute a specification in itself, but it may be included in specifications of particular protocols, thereby ensuring that a uniform and consistent notion of authentication is used.

2 The Elements of Authentication

There are two properties that may be required of an authentication process. The first relates to showing that the source of a communication is the one claimed. This property is always required and constitutes what we call *simple authentication*. The second relates to showing that the communication was generated "currently", where this term is defined according to the context. *Strong authentication* is defined by the presence of both properties. The property of being "current" is not always required by the party demanding authentication - for example access control to general computer systems is often protected by a password which is used as an unchanging authenticator over a certain period.

Simple authentication corresponds to the classes 0,1 and 2 as described in the Authentication Framework [5] while strong authentication corresponds to classes 3 and 4. Simple authentication can be seen as preventing the threat of *masquerade* and strong authentication as preventing also the threat of *replay* (these terms are defined in the OSI Security Architecture [8]) Note that these requirements may apply equally to authentication of human users, but we will deal in this paper only with electronic communications.

We deduce from these requirements that there are four elements which may be associated with a message used for authentication. We will call messages which include these elements *credentials*.

1) The claimed identity of the source.
2) A secret which must have been used to form the message.
3) A "liveness indicator" or "nonce".
4) The message contents.

The second element constitutes what is called *Claim Authentication Information* in [5]. We have already noted that all these components are not always required. Furthermore, some may be defined implicitly. Thus parts 3) and 4) will be empty for simple authentication. Also the way that the components relate to the credentials is abstract, so that we do not mean to convey that these elements are necessarily explicitly listed in the credentials. For example, the claimed identity of the source may have been established in an earlier message. The secret may be an explicit password or a key used to encrypt the message. The nonce may be the current time on a suitable clock, a counter that is incremented at each message, or a challenge previously sent.

Part 4) of the credentials may be absent if the message is intended merely to assure authentication of the other end of a secure connection, or, on the other hand, the contents may be the sole reason for sending the message. This point illustrates that this abstract framework does not differentiate between "party-to-party authentication" and "data authentication" a distinction frequently made. We note that all the mechanisms defined in the Authentication Framework [5] to provide these two services are applicable to both.

An informal authentication model presented in [5] divides the authentication process into the following phases.

1) installation phase
2) change-authenication-information phase
3) distribution phase
4) acquisition phase
5) transfer phase
6) verification phase
7) disable phase
8) re-enable phase
9) deinstallation phase

Of these phases only the transfer and verification phases are part of every authentication exchange and these are the ones that are covered in the framework. It is mentioned at the end of this paper how some of the other phases could be included in an extended framework.

3 Simple Authentication Model

In accordance with the terminology defined in the Authentication Framework [5], and elsewhere, users involved in an authentication protocol are called *principals*. These may include authentication servers of different kinds as well as entities requiring or providing authentication. The data type Principal is defined as a renaming of the standard LOTOS type Set. For use later a particular principal name called nullid is introduced.

```
type        Principal     is     Set     renamed by

sortnames   Principal     for    Element
            PrincipalSet  for    Set

opns        nullid : -> Principal

endtype
```

The next data type defines how secret keys are identified with principal identities. Sets of (identity,key) pairs, or *keymaps*, are recorded by authenticating processes for each principal. These constitute what is called *Verification Authentication Information* in [5]. Given an identity and a certain keymap, a key is defined if the identity is in the domain of the keymap.

```
type key_id_pair is Principal,Set renamed by

sortnames     keymap for Set
              kidpair for Element

formalsorts   secret

opns          makekid : Principal,secret -> kidpair     (* A kid pair is formed from
                                                          a principal and a secret*)
              key : Principal,keymap -> secret          (* The secret associated
                                                          with a principal may be
                                                          extracted from a keymap *)
              dom : keymap -> PrincipalSet              (* Each keymap has a set
                                                          of principals it knows about *)

eqns          forall id:Principal, s:secret, A:keymap

ofsort Bool   dom({}) eq {} = true;
              id NotIn dom(A) => dom(insert(makekid(id,s),A) eq
              insert(id,dom(A))) = true;
              id NotIn dom(A) => key(id,insert(makekid(id,s),A)) eq s = true;
endtype
```

The significance of the "formalsorts" is that they may be made concrete with a particular structure in the case of particular protocols. For example the sort "secret" may become a bit string if the secret is a binary key for a cryptosystem, or it may be a character string if it is a simple password. The type defining the formal simple credentials includes the above type and also defines a secret value for all values of simple credentials.

```
type   FormalSimpleCreds   is      key_id_pair

       formalsorts         simplecreds
       opns                sec : simplecreds -> secret

endtype
```

We now define the process that performs simple authentication. It is parametrised by a particular keymap. When it receives a principal name and a simple credential it checks that the secret associated with the credential is the same as that associated with the identity in the keymap. If this is not the case, or if the identity is not in the keymap, then the authentication fails. The values to be checked are received through the communication gate auth and the Boolean results are sent to the same gate.

```
process SimpleAuth [auth] (Known:keymap) : noexit :=

auth ?claimant:Principal ?cred:simplecreds;      (*Receive claimant and credentials *)
[claimant NotIn dom(Known)] -> auth!false; SimpleAuth[auth](Known)
                                (* If claimant is not known then start again *)
[]                                      (* Choice symbol*)
[claimant IsIn dom(Known)] ->           (* If claimant is known then continue*)
(
        [key(claimant,Known) eq sec(cred)] -> auth!true;
SimpleAuth[auth](Known)
        []
        [key(claimant,Known) neq sec(cred)] -> auth!false;
SimpleAuth[auth](Known)
)

endproc
```

3.1 Definition of Simple Authentication in LOTOS models

Before going on to give some examples we will now outline how we see the framework being used and what its purpose is. We assume that the authentication protocol under scrutiny is formally specified in LOTOS and that each principal is a separate process in the protocol. We will then say that a particular set of credentials are simply authentic if they are written in a standard format (an actualisation of the sort SimpleCreds) and are checked successfully by the authenticating process SimpleAuth. In every authentication protocol credentials are received by the principal doing the authenticating. Normally the checking of the credentials is not specified as part of the protocol - it is precisely the purpose of the authenticating process to make the checking explicit. A more precise formulation of how this is achieved is as follows.

Definition

Suppose a process A is defined as part of a LOTOS specification. Suppose also that there is some other process A1 such that A may be written as

```
Process A

A 1 [auth,...](...)|[auth]|SimpleAuth[auth](Known)

endproc
```

Let v be of type SimpleCreds, where SimpleCreds is an actualisation of the formalsort FormalSimpleCreds, and id be of type Principal. Suppose the process A1 includes the following events.

Process A1
..
..
auth!v !id
auth?ind:Bool
..
..
endproc

If the value of ind is "true" then we will say that the value v is *simply authentic* for A. Thus the aim is not to prove anything about the LOTOS specification of the protocol, but rather to format the specification in such a way that the credentials can be checked in a standard way by the authenticating process.

If credentials are taken to be authentic this will always be with respect to an assumption about the relationship between credentials and the secret and also, in the case of strong authentication, an assumption about the validity of the nonce. For example, in the above the assumption is that the value v could not have been formed without knowledge of the value of key(id,Known). The assumption associated with each authentication will often be a cryptographic assumption if encryption is used. It is essential to know what cryptographic assumptions are required in order that the cryptographic mechanisms employed are strong enough but not unnecessarily so (see [3]).

It may seem that the definition of (simple) authentication is unnecessarily restrictive and also does not allow for a systematic way to show when a set of credentials is **not** authentic. The examples show that a wide range of protocols can be accommodated into the definition and also that protocol weaknesses can be found through attempts to satisfy the definitions.

4 Examples of Simple Authentication

4.1 Simple Password Checking

Let us illustrate the idea of how the simple authentication framework may be used in a particular instance. We consider the checking of a simple password that is stored by the authenticating party. Such a protocol is also defined as Simple Authentication in the CCITT X509 Directory Authentication Framework [6].

Before defining the processes we need to actualise the formal types by replacing the formalsorts defined in the type FormalSimpleCreds. We will assume the existence of two sorts called charstring and bitstring throughout this paper, whose formal definitions are omitted.

type Password **is** FormalSimpleCreds **actualised by** charstring

sorts charstring **for** secret
charstring **for** simplecreds

opns length : charstring -> Nat
Minlength -> Nat

eqns **forall** pw: charstring, cred: charstring

ofsort Bool length(pw) le Minlength = false;
ofsort simplecreds sec(cred) = cred;

endtype

Here we have illustrated how additional constraints may be added for a particular protocol by adding the condition that passwords have a minimum length.

Now we define a process which receives a password and an identity and passes these to the authenticating process. This process has two event gates called receive and auth. Through gate receive it communicates with the party requiring authentication and through auth it communicates with the authenticating process.

Process B [receive,auth]: **exit**(Bool) :=

receive ?s:charstring ?id:Principal;
auth !id !s;
auth ?ind:Bool;
exit(ind)

endproc

This process accepts a password and identity pair and passes these to the authenticating process in the form of a simple credential to check that the pair is correct. It then receives a Boolean indicator to tell it what the result is. The two processes run in parallel and synchronise on events at the gate auth.

Process SimpleAuthProtocol [receive,auth] : (Bool) :=

B[receive,auth] |[auth]| SimpleAuth[auth](Known)

endproc

The password will be accepted as authentic by the process if the Boolean indicator is "true". The assumption associated with the protocol is that the credentials could not have been formed without knowledge of the secret key of the user. In this case the assumption is obvious since the secret password itself is sent. It is not specified what action is taken as a result of the authentication. In a particular implementation the keymap Known will be initialised to contain those (user,key) pairs valid in the system.

In a concrete implementation the two processes B and SimpleAuth may be located together, as when a password is presented to a computer system for authentication. On the other hand, the SimpleAuth process could be a completely separate entity, as is the case in the X509 protocol where the authenticating process represents the directory which will check correctness of a password on behalf of a principal.

4.2 Hashed Password Checking

The second example is a slightly more complicated case where the password is hashed using a one-way function and that value is used to check against a stored value. This procedure is commonly used in order that compromise of the stored password file should not allow user impersonation (see [14]).

type HashedPassword **is** FormalSimpleCreds **actualised by** charstring, bitstring

sorts charstring **for** secret
bitstring **for** simplecreds

opns hash : charstring -> bitstring

eqns **forall** cred:bitstring, s: charstring
ofsort bool sec(cred) eq s = cred eq hash(s);

endtype

The type HashedPassword associates with each simple credential a bitstring. This bitstring is a hashed value of the secret password. The equation states that if a particular secret hashes to the value of a credential then it must have been used to form the credential. The process B is similar to that in example 4.1. The difference is that the credentials output only have the identity and the hashvalue defined.

Process B [receive,auth] : **exit**(Bool) :=

receive ?s:charstring ?id:Principal;
auth !id !hash(s);
auth ?ind:Bool;
exit(ind)

endproc

As above the process SimpleAuth runs in parallel with process B.

Process SimpleAuthProtocol [receive,auth : (Bool) :=

B[receive,auth] ||[auth]|| SimpleAuth[auth](Known)

endproc

Using the definition of the type HashedPassword it can be verified that the SimpleAuth process will return a Boolean value "true" only if the value of the credentials received is equal to the hashed value of the key of the claimant identity received. In this case the assumption is that the hashed value sent with the credentials could not have been formed without knowledge of the password.

5 Strong Authentication

The strong credentials are based around the idea of encryption and it is assumed that all strong credentials contain two elements; the nonce which is used to ensure that the credentials are current, and the contents which may be used for different purposes in different protocols. We introduce a special null value for both of these, which are used if a particular instance of credentials have them missing.

```
type Nonce is Set renamed by

sortnames   nonce for     Element
            Nonceset      for     Set
opns        nullnonce:  -> nonce

endtype
```

With each credential there are two decryption operations - one which reveals the nonce and one which reveals the contents. In practice these will normally be the same decryption followed by extracting the relevant parts using formatting information. The keymap used for Simple Authentication also plays a central role here.

```
type FormalStrongCreds is key_id_pair, Nonce

formalsorts   contents, strongcreds

opns          nullconts: -> contents
              decryptnonce: strongcreds, secret -> nonce
              decryptcont : strongcreds,secret -> contents
endtype
```

We now define the process which checks that the message is strongly authentic. As for the SimpleAuth process, it is parametrised by a keymap. The process has two functions. Firstly it accepts identity and strong credential pairs, and checks whether the nonce obtained from deciphering the credentials is current, using the key belonging to the identity in the keymap. If so the credentials are strongly authentic and the process returns a Boolean value "true". If not, or if the identity is not in the domain of the keymap, it returns a Boolean value "false". The second function is to update the parameters regarding what nonces are acceptable or what keys are known for each principal. It either receives a set of nonces which replace the previous current set, or it receives a key/identity pair which is added to the key map. If a key/identity pair is already known for a given identity then the pair is rejected.

```
process StrongAuth [auth] (Current:NonceSet, Known:keymap) : noexit :=

auth ?claimant:Principal ?cred:strongcreds;
[claimant NotIn dom(Known)] -> auth!false; StrongAuth[auth](Current,Known)
[claimant IsIn dom(Known)] ->
        (
        [decryptnonce(cred,key(id,Known))  IsIn Current] -> auth !true;
        StrongAuth[auth](Current,Known)
        []
```

```
        [decryptnonce(cred,key(id,Known)) NotIn Current] -> auth !false;
        StrongAuth[auth](Current,Known)
        )
[]
auth ?New : Nonceset; StrongAuth[auth](New,Known)
[]
auth ?id:Principal ?s:secret;
        (
        [id IsIn dom(Known)] -> StrongAuth[auth](Current,Known)
                []
        [Id NotIn dom(Known)] ->
                        StrongAuth[auth](Current,insert(makekid(id,s),Known))
        )
```

endproc

The parameter Current is defined as a set of nonces. This allows it to represent a whole set of times, for example when a window of times is allowed for a timestamp. On the other hand, in a challenge-response exchange, the Current set may have only one element.

We define strong authentication of messages in LOTOS specifications in an entirely analogous way to what was done for simple authentication in 3.1. There are now two assumptions that must accompany any claim that a message is strongly authentic. Firstly that the credentials could not have been formed without knowledge of the secret key of the principal. Secondly that a message that has a current nonce cannot have been replayed from a previous message.

6 Examples of Strong Authentication

We give two examples of authentication protocols. The first is the well known Needham and Schroeder protocol [13] which is specified in a detailed manner, illustrating how different formats of credentials can be included. The second example is from the CCITT Directory Systems standard [6] which illustrates use of timestamps as nonces, and is treated only in outline.

6.1 Needham-Schroeder Protocol

This protocol was proposed in 1978 and is very significant in the history of such protocols. It has long been known that there is a weakness in the Needham-Schroeder protocol in relation to the re-use of old keys. This is a result of the responding principal having no opportunity to receive a nonce from the authentication server.

In view of the importance and interest in this protocol we will give a more or less complete specification. This illustrates how different processes can use the strong authenticating process by initialising it in different ways. We will see that the specifications of the two principals A and B are in this case quite different. Furthermore, in order that principal B can strongly authenticate its messages a dubious assumption is necessary. Thus, in common with other analyses ([4],[12]) the framework is shown to detect such a weakness.

The protocol is illustrated in figure 1 and takes place between two entities A and B who wish to agree on a common session key K(A:B) . They make use of an authentication server S that initially shares keys K(A:S) and K(B:S) with A and B respectively. We first give an informal description. The notation $\{X\}_K$ denotes the message X encrypted with key K and commas denote concatenation of messages.

The successful protocol consists of five messages as follows. Here M and N_A are nonces generated by A and N_B is a nonce generated by B.

1) A sends to S : A,B,N_A

2) S sends to A : $\{N_A,B,K(A:B),\{K(A:B),A\}_{K(B:S)}\}_{K(A:S)}$

3) A sends to B : $\{K(A:B),A\}_{K(B:S)}$

4) B sends to A : $\{N_B\}_{K(A:B)}$

5) A sends to B : $\{N_B -1\}_{K(A:B)}$

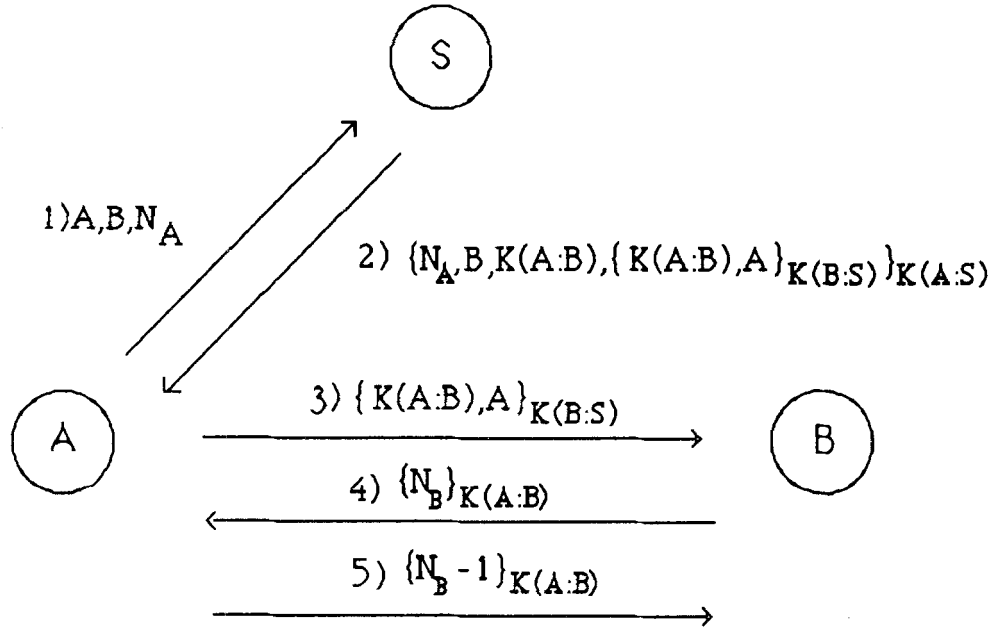

Fig. 1. Needham and Schroeder Protocol

There are a number of different types of message passed in this protocol which are all captured in the following type definition. The operation makecred specifies the format of credentials made by the server. The operation makecheck specifies the rather simpler credentials formed by A and B to check that they hold the same key.

type NScred **is** FormalStrongCreds **actualised by**
encryptionkey,NaturalNumber, bitstring

sortnames	encryptionkey	**for**	secret
	nat	**for**	nonce
	bitstring	**for**	strongcreds
	bitstring	**for**	contents

opns getkey: contents -> encryptionkey
getcred: contents -> bitstring
getid: contents -> Principal
makecred : keymap,Principal,Principal,nonce,encryptionkey -> contents
makecheck: nonce, encryptionkey -> bitstring

eqns **forall** A,B:Principal, n:nonce, k:encryptionkey, km:keymap

ofsort encryptionkey
getkey(decryptcont(makecred(km,A,B,n,k),key(A,km))) = k;
getkey(decryptcont(getcred(decryptcont(makecred(km,A,B,n,k),key(A,km))),
key(B,km)))= k;

ofsort nonce
decryptnonce(makecred(km,A,B,n,k),key(A,km)) = n;
decryptnonce(getcred(decryptcont(makecred(km,A,B,n,k),key(A,km))),key(B,km))
= nullnonce;
decryptnonce(makecheck(n,k),k) = n;

ofsort Principal
getid(decryptcont(makecred(km,A,B,n,k),key(A,km))) = B;
getid(decryptcont(getcred(decryptcont(makecred(km,A,B,n,k),key(A,km))),
key(B,km))) = A;

endtype

The equations say that when the credentials are decrypted by the first principal the key obtained from the contents is k and the nonce obtained is n. In addition the content obtained by the first principal can be used to get the credentials for the second principal. When these are decrypted by the second principal the key obtained is again k. However the nonce obtained by the second principal is the nullnonce. This is because there is no element in the credentials (message 3 in figure 1) which can be used as a nonce by B. All these equations are in terms of the keymap of the process forming the credentials.

The schematic picture in Figure 2 shows the communications gates for the processes. Note that although A and B appear to communicate with the same StrongAuth process this does not imply that in implementation it will be a shared server.

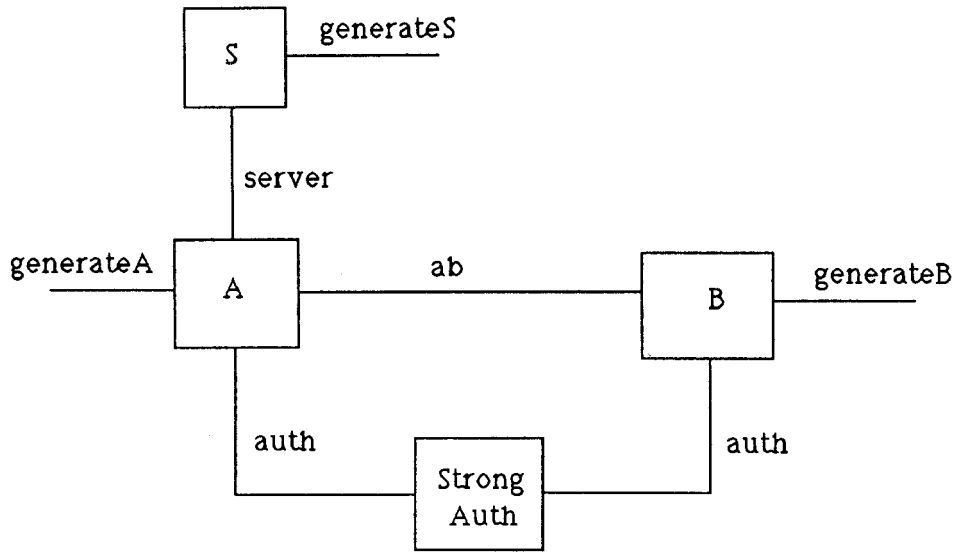

Fig. 2. Schematic process picture for Needham and Schroeder protocol

The process A has four gates. The gates server and ab are used to communicate with the other two principals, S and B respectively. The gate generateA is used to receive input such as which user to communicate with and values for nonces. The details of how these are generated are outside the scope of the protocol definition. The final gate auth is used to communicate with the authenticating process.

```
Process A[generateA,server,auth,ab]: exit :=

generateA  ?B:principal, ?nA:nonce;
(
        server !A !B !nA;                       (*Send message 1)
        server ?cred1:bitstring; exit           (*Receive message 2)
|||
        auth !insert(nA,{}); exit               (*Send nonce to auth *)
)
>>
let id=getid(decryptcont(cred1,S)) in
(
[id eq B] -> exit
[]                                              (*Check id in cred1 is same as one
sent *)
[id neq B] -> A[generateA,server,auth,ab]
)
>>
```

```
auth !S !cred1;                                    (*Send cred1 for checking *)
auth? ind:Bool;                                    (*Receive result *)
(
[ind=true] -> exit                                 (*If OK continue *)
[ind=false] -> A[generateA,server,auth,ab]         (* If bad start again *)
)
>>
(
let credB:bitstring = getcred(decryptcont(cred1,S)) in
        ab !credB;                                 (*Send message 3*)
        ab ?cred2 : bitstring;                     (* Receive message 4*)
exit(any secret)
|||
let s:secret = getkey(decryptcont(cred1,S)) in
        auth !B !s; exit(s)                        (*Update keymap with new key for B *)
)
>> accept s:secret in
(
let cred3:bitstring = makecheck(succ(decryptnonce(cred2,s)),s)  in
ab !cred3;                                         (* Send message 5 *)
Session[...](...)
)
```

endproc

The successful termination of process A starts the process Session which is outside the scope of the authentication protocol. We have changed the protocol very slightly so that A returns $N_B + 1$ = succ(NB) in message 5 rather than $N_B - 1$, simply to save defining a new operation on integers. From this basic process we can define a new process which combines the authenticating process initialised for A.

```
Process newA[generateA,server,auth,ab] : exit :=

let Known:keymap = insert (makekid(S,K(A:S)),{}) in

A[generateA,server,auth,ab]|[auth]| StrongAuth[auth](A,{},Known)
```

endproc

In this parallel combination the process StrongAuth is initialised so that the set of current nonces is empty and the value K(A:S) is the known key for principal S. Process A is initialised with the same key value for S.

The process B has one gate less than A since it does not communicate with the server.

```
Process B[generateB,auth,ab]: exit :=

ab? cred:bitstring                                 (* Receive message 3 *)
auth !S !cred;                                     (*Send message to auth *)
auth ?ind1:Bool;                                   (*Receive result *)
```

```
(
[ind1 = true] -> exit                                   (*If OK continue *)
[ind1=false] -> B[generateB,auth,ab]                    (*If bad start again *)
)
>>
let cred2: bitstring = makecheck(nB,getkey(decryptcont(cred,S))) in
ab !cred2;                                              (* send message 4*)
(
ab ?cred3:bitstring; exit                               (*Receive message 5 *)
|||
auth !insert(succ(nA),{}); exit                         (*Send nonce value to auth *)
)
>>
auth ! cred3;                                           (* Send message to auth *)
auth ?ind2: Bool;                                       (* Receive result *)
(
[ind2=true] -> exit
[ind2=false] -> B[generateB,auth,ab]
)
>>
let s:secret = getkey(decryptcont(cred,key(S,Known))),
id:Principal=getid(decryptcont(cred,key(S,Known))) in
        auth !id !s;                                    (*Update keymap with new key for A *)
Session[...](...)
```

endproc

Again, Process B is combined with the authenticating process but with different initial values.

Process newB[generateB,auth,ab]

let Known:keymap = insert (makekid(S,K(B:S)),{}) **in**

B[generateB,auth,ab]|[auth]| StrongAuth[auth](B,nullnonce,Known)

endproc

The most significant difference, as compared with the initial values used for A, is the nullnonce value used to initialise the nonce value. This is necessary in order that the initial credentials received by B are to be correctly authenticated. This dubious assumption corresponds to that made by Burrows, Abadi and Needham [4] in their analysis of this protocol and is a consequence of the weakness in the protocol mentioned above. Without making this assumption the authenticating process would return a value of "false" for the credentials received by B from S.

The server process is very simple as it does not perform any checking except to ensure that it has keys for the two principals requested. If so it simply generates credentials on demand. The keys known by the server can be initialised using the parameter Known.

```
Process Server[server,generateS](Known : keymap): noexit :=

server ?A:Principal ?B:Principal ?nA:nat;
(
[A IsIn dom(Known)  and B IsIn dom(Known)  ] -> exit
[]
[A  NotIn dom(Known) or B  NotIn dom(Known)] ->
Server[server,generateS](Known)
)
>>
generateS ?K:encryptionkey;
let cred1 = makecred(Known,A,B,nA,K) in
server !cred1;
Server[server,generateS]

endproc
```

We can finally combine all the processes defined so far into a complete specification of the Needham-Schroeder protocol.

```
specification Needham-Schroeder_Protocol [generateA,generateB,
server,ab,auth] : noexit

library
        Boolean,Set,NaturalNumber
endlib

behaviour
Server[server,generateS]|[server]|newA[generateA,server,auth,ab] |[ab]|
newB[generateB,auth,ab]

endspec
```

What we have done here is to specify the protocol in a particular way, so that all decisions about authentication of messages are made by the StrongAuth process, initialised in a way appropriate to that process which is demanding the authentication.

There are a number of features of the protocol that appear in the formal specification that are not present in the mere definition of the messages passed between principals. The most obvious of these are in the authentication process itself. Another is the checking by process A that the identity received in the credentials from the server in message 2 is the same as that sent by A to the server. As was pointed out in [13] this is essential to the security of the protocol but it illustrates that there is more to this authentication protocol than just checking that messages were timely and came from the claimed source - it is also necessary to examine whether the meaning ascribed to messages (in this case that the key sent is good for use between A and B) is actually conveyed in the text sent.

6.2 X509 Oneway Protocol

For our final example we use one of the strong authentication procedures defined in the Directory System Authentication Framework, X509 section 9.2. For simplicity we choose the one-way authentication protocol.

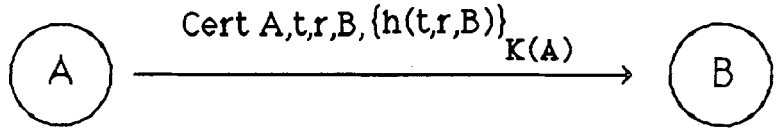

Fig. 3. X509 Oneway Protocol

This is a public key protocol and involves sending a certified copy of the sender's public key, Cert A, along with the message. However, for the sake of simplicity we will assume that the recipient is already in possession of the sender's public key. In fact we will omit all the details of checking the credentials in this example since our current interest is in illustrating use of the timestamp as nonce. Apart from the certificate the message includes a timestamp, t, a random number, r, the recipient's identity, and a signed hashed version of these. As usual we first actualise the formal sorts defined.

```
type X509token is FormalStrongCreds actualised by NaturalNumber,
bitstring

sorts       bitstring    for    secret
            bitstring    for    strongcreds
            nat          for    nonce
            bitstring    for    contents
endtype
```

The process B simply receives the strong credentials and sends them to the StrongAuth process for checking. If they are correct then it proceeds to some undefined process Session. If not it starts again. B may also receive an updated time window from the clock, which is passed on to the authenticating process.

```
process B[receive,auth,update] : noexit :=
(
receive ?cred:strongcreds?id:Principal;          (*Receive message*)
auth !id !cred;                                   (*Send to authenticating process*)
auth ?ind:Bool;                                   (*Receive decision*)
        (
        [ind=true] -> exit;
        []
        [ind=false] -> B[receive,auth,update]
        )
>> Session[...](...)
)
[]
```

```
(
update ?CurrentTimeWindow : NonceSet;
auth ! CurrentTimeWindow ;
B[receive,auth,update]
)
endproc
```

The process defining the clock is as follows.

```
process Clock[update]:noexit :=

update !CurrentTimeWindow; Clock[update]      (*Send current time to
                                               authenticating process*)
endproc
```

The clock operates in parallel with B and periodically updates the authenticating process with the current time window. The nonce sent with the message has a timestamp which is checked to be in the current window.

```
process B1

B[receive,auth,update] |[auth]| StrongAuth[auth]  |[update]| Clock[update]

endproc
```

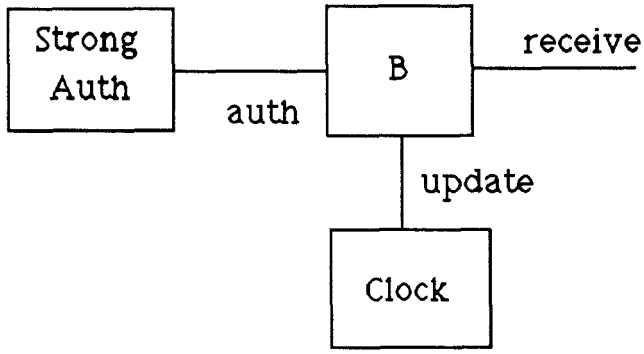

Fig. 4. Schematic process picture for B in X509 Oneway protocol

We deduce again that the message received by B is fully authenticated if the Boolean indicator received from the authenticating process is true. Again there is the assumption that the received credentials could not have been formed without knowledge of the associated secret and use of a valid time stamp.

7 Further Developments

There are various obvious ways in which the framework could be extended. At present only authentication using encryption is included in the strong authentication process. When secrecy is not required, a oneway function can also be used, in which case there is no

decryption function defined. It would be relatively simple to extend the authentication process to include this case. This would then cover all the authentication types defined in the Authentication Framework [5]. As a further extension, all the phases in the informal model from the Authentication Framework [5], as mentioned in section 2, could be included. Many of them are fairly trivial to add, for example the deinstallation phase simply consists of removing a known user from the keymap.

A more radical extension would be to add the properties of the encryption algorithms into the framework so as to describe the properties required before accepting that a message was formed using the secret associated with the claimed source. Another direction is to include the notion of trust. By specifying a subset of principals as a trusted set, messages authenticated from such sources can be acted upon differently, for example to update the keymap. Finally it can be considered whether the other aspects of security such as confidentiality and access control can be modelled in a complementary framework, as is being done informally within ISO [7].

Acknowledgements

I am very grateful to Stewart Black, formerly of Hewlett Packard Laboratories, and Sukhvinder Aujla of British Telecom, for critical comments on various versions of this paper. This work is supported by SERC Research Grant GR/G19787

References

1. I.Adjubi, G.Scollo & M.van Sinderen, Formal Description of the OSI Session Layer: Introduction, in The Formal Description Technique LOTOS, P.H.J.van Eijk, C.A.Vissers, M.Diaz (Eds.), North-Holland 1989.

2. T.Bolognesi & E.Brinksma, Introduction to the ISO Specification Language LOTOS, Computer Networks and ISDN Systems, Vol 14, No 1, 1987, pp 25-59.

3. C.A.Boyd, Hidden Assumptions in Cryptographic Protocols, Proceedings of IEE, Part E, Vol 137, No 6, November 1990, pp433-436.

4. M.Burrows, M.Abadi and R.Needham, A Logic of Authentication, Proceedings of the Royal Society, Series A, Volume 426, Number 1871, December 1989, pp233-271.

5. ISO DP 10181-2, Security Frameworks for Open Systems - Part 2: Authentication Framework.

6. CCITT X509, The Directory - Authentication Framework, November 1987.

7. ISO JTC1/SC21 N6765, Guide to Open System Security, February 1992.

8. ISO 7498/2 Open Systems Interconnection Model Security Architecture

9. ISO 8807 : LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, 1988.

10 A.K.Marshall, Introduction to LOTOS Tools, in The Formal Description Technique LOTOS, P.H.J.van Eijk, C.A.Vissers, M.Diaz (Eds.), North-Holland 1989.

11. C.Meadows, A System for the Specification and Analysis of Key Management Protocols, Proceedings of the 1991 IEEE Computer Society Symposium on Security and Privacy, pp. 182-195, IEEE Computer Society Press, 1991.

12. J.K.Millen, S.C.Clark & S.B.Freedman, The Interrogator: Protocol Security Analysis, IEEE Transactions on Software Engineering, SE-13, No. 2, February 1987.

13. R.M.Needham & M.D.Schroeder, Using Encryption for Authentication in Large Networks of Computers, Communications of the ACM, 21,12, December 1978, pp.993-999.

14. G.B.Purdy, A High Security Log-in Procedure, Communications of the ACM, 17,8,August 1974, pp.442-445.

15. V.Varadharajan & S.Black, Formal Specification of a Secure Distributed Message System, Proceedings of 12th National Computer Security Centre Conference, Baltimore, 1989.