COWS specification of an XDM-based protocol for disconnected clinics in an healthcare scenario

Technical Report

Author: Massimiliano Masi, Rosario Pugliese, Francesco Tiezzi

Date: April 2, 2010
Institute: Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

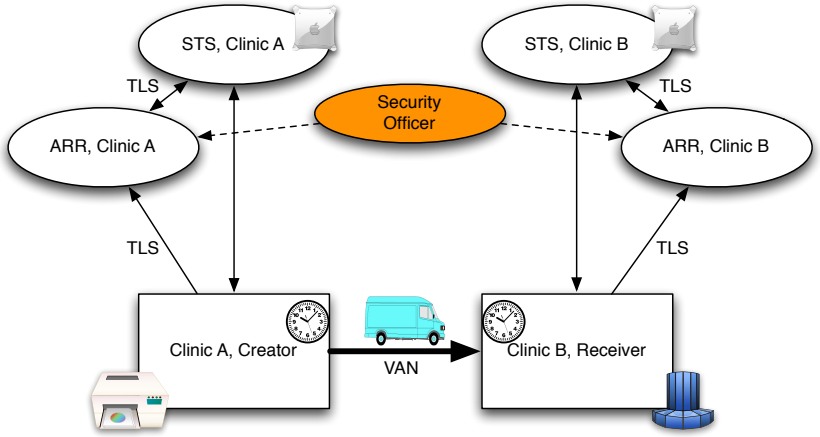

Figure 1: A typical scenario

1 An healthcare scenario with disconnected clinics

We tackle the problem of sharing patients healthcare data among clinics without any connection to the Internet. This is a frequent problem in developing countries where no network infrastructure is given by the institutions. A clinic located in a desert that can be reached only by means of hundreds of kilometers of tracks sand, or a caravan that travels along townships with first-aid equipment are scenarios in the scope of our work.

The scenario considered here is illustrated in Figure 1. Some patients are receiving healthcare treatment in clinic *A*, located in a region without neither Internet connection, satellite, nor GSM coverage. A governmental agency, clinic *B*, is providing a storage of patients' healthcare records, based on standard profiles. To provide optimum care for patients, clinic *A* must register all patients' healthcare documents in the central storage. The XDM specification defines a way for clinic *A* to write the documents in a certain format within portable media (e.g. CD, DVD, USB drive) that patients can carry whenever they travel over the region of residence. The records produced can also be sent to other clinics (e.g. a regional hospital) creating a connected graph of clinics. Because of lack of network connection, the documents registered in the portable media are sent using a car-transportation system. At clinic *B*, authorised personnel can read the portable media and import the documents in the local hospital information system.

We propose an XDM-based protocol for facing the severe security and safety problems on the treatment of patients healthcare data that the scenario outlined above presents. For example, a patient could forge his own portable media including new prescriptions for drugs or an intruder could easily have access to the data by hijacking the carrier on its way (or the carrier itself could act as an intruder).

Of course, we also want to preserve interoperability. Therefore, we only rely on standards and specifications usually used in EHR settings coming e.g. from HL7, IHE and W3C. In this way, e.g., we cannot prevent the attacks by the intruder, but we can let another actor, that we call *security officer*[1], to take charge of tracing if one of the attacks mentioned above was performed.

2 CMC specification of the scenario

We report here the 'machine readable' syntax of COWS accepted by CMC and the complete specification of the considered healthcare scenario, together with the SocL formulation of the properties we have checked, written using such syntax.

[1]In the real world, the security officer can be seen as a persona that, on duty of the government, supervises the correct information flow inside the clinics.

s	::=		(services)
		`nil`	(empty activity)
	|	`kill(k)`	(kill)
	|	`u.u'!` <*args*>	(invoke)
	|	`p.o?` <*params*> . s	(receive)
	|	$s_1 + \ldots + s_n$	(receive-guarded choice)
	|	$s_1 \mid s_2$	(parallel composition)
	|	$\{\, s \,\}$	(protection)
	|	$[\mathtt{n}\sharp]\, s$	(name delimitation)
	|	$[\mathtt{k}]\, s$	(kill delimitation)
	|	$[\mathtt{X}]\, s$	(variable delimitation)
	|	$* s$	(replication)
	|	`A`(*aparams*)	(call)
	|	`let A`(*fparams*) $= s_1$ `...` `B`(*fparams*) $= s_2$ `in` s' `end`	(let construct)
e	::=	`X` | `v` | $e_1 + e_2$ | e_1 `le` e_2 | ...	(expressions)
args	::=	e | *args*, *args*	(invoke arguments)
params	::=	`X` | `v`	(receive parameters)
	|	*params*, *params*	
fparams	::=	`X` | `n` | `k`	(formal parameters)
	|	*fparams*, *fparams*	
aparams	::=	`X` | `v` | `k`	(actual parameters)
	|	*aparams*, *aparams*	

Table 1: Syntax of COWS accepted by CMC

2.1 Syntax accepted by CMC

The syntax of COWS accepted by CMC is presented in Table 1. *Killer labels* (ranged over by `k`, `k'`, ...) start with lower case letters and can only be used as argument of kill activities. *Variables* (ranged over by `X`, `Y`, ...) start with capital letters. *Service identifiers* (ranged over by `A`, `A'`, ...) start with capital letters and each of them has a fixed non-negative arity. *Names* (ranged over by `n`, `m`,...,`p`,`p'`,...,`o`,`o'`, ...) start with lower case letters. *Values* (ranged over by `v`, `v'`, ...) are integer numbers, booleans, or names. *Identifiers* (ranged over by `u`, `u'`, ...) are variables or names. The arguments of a receive-guarded choice must be receive activities. The expression operators + and `le` are defined as follows: if both e_1 and e_2 are evaluated as integer numbers then the evaluation of $e_1 + e_2$ returns the integer number corresponding to their sum, otherwise it returns the name corresponding to their concatenation; if both e_1 and e_2 are evaluated as values then the evaluation of e_1 `le` e_2 returns the boolean `true` if the value corresponding to e_1 is not greater than the value corresponding to e_2, otherwise it returns the boolean `false`.

The let construct permits to define services in a modular style, thus facilitating re-use of the same 'service code'. `let A`(*fparams*) $= s$ ... `in` s' `end` behaves like s', where calls to `A` can occur. A service *call* `A`(*aparams*) occurring in the body s' of a `let A`(*fparams*) $= s$ ... `in` s' `end` behaves like the service obtained from s by replacing the formal parameters *fparams* with the corresponding actual parameters *aparams*.

The syntax of SocL accepted by CMC slightly differs from that presented in [Fantechi, Gnesi, Lapadula, Mazzanti, Pugliese, Tiezzi: A model checking approach for verifying COWS specifications. In

FASE, LNCS 4961, 230-245, 2008] mainly for what concern the notation to indicate correlation variables. In fact, given a variable `var`, its binding occurrence (i.e. `var` in SocL) is written `$var`, while its free occurrences are written `%var`.

2.2 CMC specification and abstractions

We report here the complete specification of the healthcare scenario, written in the syntax accepted by CMC, together with four sets of abstraction rules and SocL formulae, one for each considered attack.

```
let

Cipher(enc_KbPub_Req,enc_KbPub_Resp,dec_KbSec_Req,dec_KbSec_Resp,enc_KaPub_Req,
       enc_KaPub_Resp,dec_KaSec_Req,dec_KaSec_Resp) =
            * [Client][Datum]
              cipher.enc_KbPub_Req?<Client,Datum>.
                [fresh#] ( Client.enc_KbPub_Resp!<Datum,fresh>
                           | *[Client2] cipher.dec_KbSec_Req?<Client2,fresh>.
                                        Client2.dec_KbSec_Resp!<fresh,Datum> )
            |
            * [Client][Datum]
              cipher.enc_KaPub_Req?<Client,Datum>.
                [fresh#] ( Client.enc_KaPub_Resp!<Datum,fresh>
                           | *[Client2] cipher.dec_KaSec_Req?<Client2,fresh>.
                                        Client2.dec_KaSec_Resp!<fresh,Datum> )


Signer(sign_KaSec_Req,sign_KaSec_Resp,verify_KaPub_Req,verify_KaPub_Resp,
       sign_KstsaSec_Req,sign_KstsaSec_Resp,verify_KstsaPub_Req,verify_KstsaPub_Resp,
       sign_KstsbSec_Req,sign_KstsbSec_Resp,verify_KstsbPub_Req,verify_KstsbPub_Resp,
       sign_KiSec_Req,sign_KiSec_Resp,verify_KiPub_Req,verify_KiPub_Resp) =
            * [Requester][RequesterName][Hash]
              signer.sign_KaSec_Req?<Requester,RequesterName,Hash>.
                [fresh#] ( Requester.sign_KaSec_Resp!<Hash,fresh>
                           | * [Verifier][X]
                               ( signer.verify_KaPub_Req?<Verifier,RequesterName,X>.
                                     ( sys.attack3Detected!<invalidSignature>
                                       | sys.attack3Detected?<invalidSignature>.nil)
                                 +
                                 signer.verify_KaPub_Req?<Verifier,RequesterName,fresh>.
                                     Verifier.verify_KaPub_Resp!<fresh,Hash>
                               )
                         )
            |
            * [Requester][RequesterName][Hash]
              signer.sign_KstsaSec_Req?<Requester,RequesterName,Hash>.
                [fresh#] ( Requester.sign_KstsaSec_Resp!<Hash,fresh>
                           | * [Verifier][X]
                               ( signer.verify_KstsaPub_Req?<Verifier,RequesterName,X>.
                                     ( sys.attack3Detected!<invalidSignature>
                                       | sys.attack3Detected?<invalidSignature>.nil)
                                 +
                                 signer.verify_KstsaPub_Req?<Verifier,RequesterName,fresh>.
                                     Verifier.verify_KstsaPub_Resp!<fresh,Hash>
                               )
                         )
            |
            * [Requester][RequesterName][Hash]
              signer.sign_KstsbSec_Req?<Requester,RequesterName,Hash>.
                [fresh#] ( Requester.sign_KstsbSec_Resp!<Hash,fresh>
                           | * [Verifier][X]
                               ( signer.verify_KstsbPub_Req?<Verifier,RequesterName,X>.
                                     ( sys.attack3Detected!<invalidSignature>
                                       | sys.attack3Detected?<invalidSignature>.nil)
                                 +
                                 signer.verify_KstsbPub_Req?<Verifier,RequesterName,fresh>.
                                     Verifier.verify_KstsbPub_Resp!<fresh,Hash>
                               )
                         )
            |
            * [Requester][RequesterName][Hash]
```

```
              signer.sign_KiSec_Req?<Requester,RequesterName,Hash>.
                [fresh#] ( Requester.sign_KiSec_Resp!<Hash,fresh>
                           | * [Verifier][X]
                               ( signer.verify_KiPub_Req?<Verifier,RequesterName,X>.
                                     ( sys.attack3Detected!<invalidSignature>
                                       | sys.attack3Detected?<invalidSignature>.nil)
                                 +
                                 signer.verify_KiPub_Req?<Verifier,RequesterName,fresh>.
                                     Verifier.verify_KiPub_Resp!<fresh,Hash>
                               )
                         )


Sha(hashReq,hashResp) =
        ( * [Client][Datum1][Datum2]
             hFunc.hashReq?<Client,Datum1,Datum2>.
                Client.hashResp!<Datum1,Datum2,Datum1+Datum2>
          | * [Client][Datum1][Datum2][Datum3][Datum4][Datum5][Datum6]
              hFunc.hashReq?<Client,Datum1,Datum2,Datum3,Datum4,Datum5,Datum6>.
                 [conc#]
                 ( hFunc.conc!<Datum1+Datum2>
                   | [X1] hFunc.conc?<X1>.
                     ( hFunc.conc!<X1+Datum3>
                       | [X2] hFunc.conc?<X2>.
                         ( hFunc.conc!<X2+Datum4>
                           | [X3] hFunc.conc?<X3>.
                             ( hFunc.conc!<X3+Datum5>
                               | [X4] hFunc.conc?<X4>.
                                 ( hFunc.conc!<X4+Datum6>
                                   | [X5] hFunc.conc?<X5>.
                                     Client.hashResp!<Datum1,Datum2,Datum3,Datum4,Datum5,Datum6,X5>
                                 )
                             )
                         )
                     )


                 )
          )


Clock(clock,start,inc) = [c#]([Y] clock.get?<Y>.
                                  [n#]( c.init!<start+inc,n>
                                        | c.initAck?<n>. clock.ret!<Y,start> )
                              | * [X][Z]
                                  c.init?<X,Z>.
                                    ( c.initAck!<Z>
                                      | [Y] clock.get?<Y>.
                                            [n#]( c.init!<X+inc,n>
                                                  | c.initAck?<n>. clock.ret!<Y,X>
                                                )
                                    )
                            )


Lifo(q) = [m#][h#]
          (*[YV1][YV2][YV3][YV4][YV5][YR][YE][YZ]
            (q.push?<YV1,YV2,YV3,YV4,YV5,YZ>.[Z](m.op?<Z>.
                 [c#](h.op!<YV1,YV2,YV3,YV4,YV5,Z,c>|m.op!<c>|YZ.op!<>))
             + q.pop?<YR,YE>.[Z](m.op?<Z>.[YV1][YV2][YV3][YV4][YV5][YT]
                                      h.op?<YV1,YV2,YV3,YV4,YV5,YT,Z>.
                                         (m.op!<YT>| YR.op!<YV1,YV2,YV3,YV4,YV5>)
                                +m.op?<empty>.(m.op!<empty>|YE.op!<>) ) )
           |m.op!<empty>)


ARR(p,q,write) =
              (Lifo(q)
               |
               -- Functionality for creator
               * [Ts1][Ctx][A][B]
                 p.write?<Ts1,Ctx,A,B>.
```

```
                [Ts2][SignedDoc]
                p.write?<Ts1,Ctx,A,B,Ts2,SignedDoc>.
                 [n#]
                 ( -- Write the data in the Lifo queue of the ARR of the creator
                   q.push!<Ctx,A,B,Ts2,SignedDoc,n> | n.op?<>. A.ack!<> )
              |
              -- Functionality for receiver
              * [Ts2][Ts3][A][Ctx][SignedDoc]
                p.write?<Ts2,Ts3,A,Ctx,SignedDoc>.
                 [n#]
                 ( -- -- Write the data in the Lifo queue of the ARR of the creator
                   q.push!<Ctx,A,p,Ts2,SignedDoc,n>
                   |
                   -- If another 'write' request with the same context arrives,
                   -- than the context has been reused by an intruder
                   [X1][X2][X3][X4]
                   p.write?<X1,X2,X3,Ctx,X4>.
                     (sys.attack4Detected!<Ctx>
                      | sys.attack4Detected?<Ctx>.nil )
                 )
               )


STS(sts,stsName,client,getToken,retToken,validateToken,validateResp,write,clock,
    encode,encodeResp,sign,signResp,verifyReq,verifyResp,hashReq,hashResp) =
  * [A][B][User][Ts1]
    sts.getToken?<A,B,User,Ts1>.
    -- Create a unique context
    [conc#](sts.conc!<ctx+A>
            | [X1] sts.conc?<X1>.
              ( sts.conc!<X1+B>
                | [X2] sts.conc?<X2>.
                  ( sts.conc!<X2+Ts1>
                    | [Ctx] sts.conc?<Ctx>.
                      [ts#]
                      ( client.write!<Ts1,Ctx,A,B>
                        | clock.get!<ts>
                        | [SamlTs] clock.ret?<ts,SamlTs>.
                          ( -- Encrypt Ctx with B's public key
                            cipher.encode!<sts,Ctx>
                            | [EncCtx] sts.encodeResp?<Ctx,EncCtx>.
                              ( -- Create a token SAML
                                hFunc.hashReq!<sts,A,stsName,SamlTs,User,B,EncCtx>
                                | [SAMLhash] sts.hashResp?<A,stsName,SamlTs,User,B,EncCtx,SAMLhash>.
                                  ( -- Sign the hash code with stsA's private key
                                    signer.sign!<sts,stsName,SAMLhash>
                                    | [Signature] sts.signResp?<SAMLhash,Signature>.
                                      -- Return the token
                                      client.retToken!<Ctx,A,stsName,SamlTs,User,B,EncCtx,Signature>
                                  )
                              )
                          )
                      )
                  )
              )
          )
    |
    * [MsgID][Ts][A][StsNameA][SamlTs][User][B][EncCtx][Signature]
      sts.validateToken?<MsgID,Ts,A,StsNameA,SamlTs,User,B,EncCtx,Signature>.
      ( -- Decipher the signature with stsA's public key
        signer.verifyReq!<sts,StsNameA,Signature>
        | [DecipheredHash]
          sts.verifyResp?<Signature,DecipheredHash>.
          ( -- Calculate the token hash
            hFunc.hashReq!<sts,A,StsNameA,SamlTs,User,B,EncCtx>
            | [Hash] sts.hashResp?<A,StsNameA,SamlTs,User,B,EncCtx,Hash>.
              -- Compare the two hashes, i.e. the new one with the
              -- dechipered one
              [comp#]
              ( sts.comp!<Hash>
                | [X][msgId#]
                  ( sts.comp?<X>.
```

```
                    -- The two hashes do not coincide (broken signature):
                    -- send a negative response
                    client.validateResp!<MsgID,msgId,invalid>
                  + sts.comp?<DecipheredHash>.
                    -- The two hashes coincide: send a positive response
                    client.validateResp!<MsgID,msgId,valid>
                )
            )
        )
    )


Creator(a,stsA,user,doc,user2,doc2,clock,getToken,retToken,write,sign,signResp,hashReq,hashResp) =
  [ts#]
  ( clock.get!<ts>
    | [Ts1] clock.ret?<ts,Ts1>.
      ( stsA.getToken!<a,b,user,Ts1>
        | [Ctx][StsName][SamlTs][EncCtx][Signature]
        a.retToken?<Ctx,a,StsName,SamlTs,user,b,EncCtx,Signature>.
        ( clock.get!<ts>
          | [Ts2] clock.ret?<ts,Ts2>.
            ( -- Create the hash of doc
              hFunc.hashReq!<a,user,doc>
              | [Dochash] a.hashResp?<user,doc,Dochash>.
                ( -- Sign the hash code: we use "a" also as  creator name
                  signer.sign!<a,a,Dochash>
                  | [SignedDoc] a.signResp?<Dochash,SignedDoc>.
                    [msgId1#]
                    ( -- Updates the audit of the transaction
                      a.write!<Ts1,Ctx,a,b,Ts2,SignedDoc>
                      | -- Wait an ack from ARR:
                        -- this interaction is not present in the protocol but
                        -- it is reasonable to assume this acknowledgement
                        -- since the communication takes place along a TLS channel.
                        -- This avoids that the officer reads the ARR (and the van leaves)
                        -- between a "write" and a "push"
                        a.ack?<>.
                        ( -- Send the document over portable media to B via van
                          b.van!<a,b,msgId1,Ts2,a,user,doc,SignedDoc,a,StsName,
                                SamlTs,user,b,EncCtx,Signature>
                          |
                          -----------------
                          -- 2nd message --
                          -----------------
                          [ts#] ( clock.get!<ts>
                                  | [Ts1] clock.ret?<ts,Ts1>.
                                    ( stsA.getToken!<a,b,user2,Ts1>
                                      | [Ctx][StsName][SamlTs][EncCtx][Signature]
                                        a.retToken?<Ctx,a,StsName,SamlTs,user2,b,EncCtx,Signature>.
                                        ( clock.get!<ts>
                                          | [Ts2] clock.ret?<ts,Ts2>.
                                            ( -- Create the hash of doc2
                                              hFunc.hashReq!<a,user2,doc2>
                                              | [Dochash] a.hashResp?<user2,doc2,Dochash>.
                                                ( -- Sign the hash code: we use "a" also as  creator name
                                                  signer.sign!<a,a,Dochash>
                                                  | [SignedDoc] a.signResp?<Dochash,SignedDoc>.
                                                    [msgId1#]
                                                    ( -- Updates the audit of the transaction
                                                      a.write!<Ts1,Ctx,a,b,Ts2,SignedDoc>
                                                      | a.ack?<>.
                                                        ( -- Send the document doc2 over portable
                                                          -- media to B via van
                                                          b.van!<a,b,msgId1,Ts2,a,user2,doc2,SignedDoc,a,
                                                                 StsName,SamlTs,user2,b,EncCtx,Signature>
                                                          |
                                                          ----------------------
                                                          -- Further messages --
                                                          ----------------------
                                                          -- The specification of further messages is not
                                                          -- relevant for our analysis
                                                          nil
```

```
                                                    )
                                                  )
                                                )
                                              )
                                            )
                                          )
                                        )
                                      )
                                    )
                                  )
                                )
                              )
                            )
                          )
                        )
                      )
                    )
                  )
                )
              )
            )
          )
        )
      )
    )
  )


ClinicA(a,q,stsA,stsNameA,user,doc,user2,doc2,sign_KaSec_Req,sign_KaSec_Resp,
        sign_KstsaSec_Req,sign_KstsaSec_Resp,verify_KstsbPub_Req,verify_KstsbPub_Resp,
        enc_KbPub_Req,enc_KbPub_Resp,hashReq,hashResp) =
  [clockA#][getToken#][retToken#][validateToken#][validateResp#][write#]
  ( Clock(clockA,1,1)
    | ARR(a,q,write)
    | Creator(a,stsA,user,doc,user2,doc2,clockA,getToken,retToken,write,sign_KaSec_Req,
              sign_KaSec_Resp,hashReq,hashResp)
    | STS(stsA,stsNameA,a,getToken,retToken,validateToken,validateResp,write,clockA,
          enc_KbPub_Req,enc_KbPub_Resp,sign_KstsaSec_Req,sign_KstsaSec_Resp,verify_KstsbPub_Req,
          verify_KstsbPub_Resp,hashReq,hashResp)
  )


Receiver(clock,validateToken,validateResp,write,decode,decodeResp,verifyReq,verifyResp,hashReq,hashResp) =
  * [A][MsgId1][Ts2][AfromDoc][User][Doc][SignedDoc][AfromToken][StsNameA][SamlTs][UserFromToken][BfromToken]
    [EncCtx][Signature]
    b.van?<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,StsNameA,SamlTs,UserFromToken,BfromToken,
           EncCtx,Signature>.
     [ts#]
     ( clock.get!<ts>
       | [Ts3] clock.ret?<ts,Ts3>.
                 [comp#][X][Y]
                 ( -- Check if the A's certificate in the document signature equals the certificate
                   -- within the token (HoK) and check if the user value in the document signature
                   -- equals the user within the token
                   b.comp!<AfromDoc,User>
                   | b.comp?<X,Y>.
                          ( sys.attack2Detected!<User,UserFromToken>
                            | sys.attack2Detected?<User,UserFromToken>.nil )
                     + b.comp?<AfromToken,UserFromToken>.
                       [MsgID]
                       ( -- Validate the assertion to its STS
                         stsB.validateToken!<MsgId1,Ts2,AfromToken,StsNameA,SamlTs,UserFromToken,BfromToken,
                                             EncCtx,Signature>
                         | b.validateResp?<MsgId1,MsgID,invalid>.
                               sys.attackDetected!<invalidToken>
                           + b.validateResp?<MsgId1,MsgID,valid>.
                                ( -- Check the signature of the document
                                  signer.verifyReq!<b,AfromDoc,SignedDoc>
                                  | [DecipheredHash]
                                     b.verifyResp?<SignedDoc,DecipheredHash>.
                                     ( -- Calculate the hash
                                       hFunc.hashReq!<b,User,Doc>
                                       | [Hash] b.hashResp?<User,Doc,Hash>.
                                             [comp#] -- Compare the two hashes
                                             ( b.comp!<Hash>
                                               | [X][msgId#]
                                                 ( b.comp?<X>.
                                                      -- The two hashes do not coincide (broken signature)
                                                      ( sys.attackDetected!<invalidSignature>
                                                        | sys.attackDetected?<invalidSignature>.nil )
                                                   + b.comp?<DecipheredHash>.
                                                      -- The two hashes coincide
                                                      ( -- Decode ctx with B's secret key
```

```
                                                    cipher.decode!<b,EncCtx>
                                                    | [Ctx]
                                                       b.decodeResp?<EncCtx,Ctx>.
                                                       -- Write to ARR
                                                       b.write!<Ts2,Ts3,A,Ctx,SignedDoc>
                                                 )
                                              )
                                           )
                                        )
                                     )
                                  )
                               )
     )


ClinicB(qb,dec_KbSec_Req,dec_KbSec_Resp,enc_KaPub_Req,enc_KaPub_Resp,sign_KstsbSec_Req,
        sign_KstsbSec_Resp,verify_KstsaPub_Req,verify_KstsaPub_Resp,verify_KaPub_Req,
        verify_KaPub_Resp,hashReq,hashResp) =
  [clockB#][getToken#][retToken#][validateToken#][validateResp#][write#]
  ( Clock(clockB,1,1)
    | ARR(b,qb,write)
    | Receiver(clockB,validateToken,validateResp,write,dec_KbSec_Req,dec_KbSec_Resp,
               verify_KaPub_Req,verify_KaPub_Resp,hashReq,hashResp)
    | STS(stsB,stsNameB,b,getToken,retToken,validateToken,validateResp,write,clockB,
          enc_KaPub_Req,enc_KaPub_Resp,sign_KstsbSec_Req,sign_KstsbSec_Resp,
          verify_KstsaPub_Req,verify_KstsaPub_Resp,hashReq,hashResp)
  )


Officer(qa,qb)=
   officier.start!<>
   |
   -- The officer is activated by a signal "activate"
   * [k] officier.activate?<>.
          -- Only one instance of the officer can run at the same time
          officier.start?<>.
           -- Firstly, the officer gets the value maxt1 from ARR_a
           [r#][e#][n#]
           (qa.pop!<r,e>
            | [Ctx1][A1][B1][MaxT1][SignedDoc1]
              ( e.op?<>.
                   -- ARR_a is empty: officer does nothing
                   nil
                +
                r.op?<Ctx1,A1,B1,MaxT1,SignedDoc1>.
                    ( -- The officer has saved MaxT1 and
                      -- then puts the extracted tuple in the queue
                      qa.push!<Ctx1,A1,B1,MaxT1,SignedDoc1,n>
                      |
                      n.op?<>.
                      -- Then, the officer reads ARR_b (within clinicB)
                      -- NB: it is worth noticing that the officer
                      -- should copy the queue and act on its copy,
                      -- while here, in order to reduce the model state space,
                      -- the officer extracts all tuples from the queue of ARR_b.
                      -- This means that at the end of its execution,
                      -- the queue will be empty. This does not affect our
                      -- analysis, since we consider here just one attack and,
                      -- hence, just an officer instance in action.
                      [loop#](loop.op!<>
                               | *loop.op?<>.
                                          (qb.pop!<r,e>
                                           | [Ctx][A][B][Ts][SignedDoc]
                                             ( e.op?<>.
                                                 -- Exit from the loop
                                                 off.end!<>
                                               +
                                               r.op?<Ctx,A,B,Ts,SignedDoc>.
                                                 ( off.compare!<Ts le MaxT1>
                                                   | off.compare?<true>.
                                                        ( -- Add the tuple to "logB" (which is a
                                                          -- parallel composition of input actions)
```

```
                                                    [Xack] off.logB?<Ctx,SignedDoc,Xack>.
                                                           off.Xack!<>
                                                    | -- Start a new step of the loop
                                                    loop.op!<> )
                                                + off.compare?<false>.
                                                    -- Start a new step of the loop
                                                    loop.op!<>)
                                        )
                                    )
                            )
                        |
                        -- At the end of the loop above, the officer compares
                        -- the tuples in queue "qa" with those in "logB"
                        off.end?<>.
                            -- Scan ARR_a
                            -- NB: as above, the officer extracts all tuples
                            -- from the queue of ARR_A.
                            [loop#](loop.op!<>
                                    | *loop.op?<>.
                                        (qa.pop!<r,e>
                                         | [Ctx][A][B][Ts][SignedDoc]
                                           ( e.op?<>.
                                               -- Exit from the cycle
                                               ( {officier.start!<>} | kill(k) )
                                             +
                                             r.op?<Ctx,A,B,Ts,SignedDoc>.
                                               ( off.compare!<Ts le MaxT1>
                                                 | off.compare?<true>.
                                                   [n#]
                                                   ( -- Check if the tuple is in logB
                                                     off.logB!<Ctx,SignedDoc,n>
                                                     |
                                                     [X1][X2][X3]
                                                     ( off.logB?<X1,X2,X3>.
                                                         -- Message suppressed
                                                         ( sys.attack1Detected!<messageSuppressed,SignedDoc>
                                                           | sys.attack1Detected?<messageSuppressed,SignedDoc>.
                                                             loop.op!<>)
                                                       +
                                                       off.n?<>.
                                                         -- Tuple found: it is ok, so continue the loop
                                                         loop.op!<>
                                                     )
                                                   )
                                                   + off.compare?<false>.
                                                      -- Ignore the tuple because it has been inserted
                                                      -- after that the officer has started to read
                                                      loop.op!<>)
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                    )
                )
            )


-- This Intruder suppresses the messages sent by van
Intruder1() =
  [A][MsgId1][Ts2][AfromDoc][User][Doc][SignedDoc][AfromToken][StsNameA]
  [SamlTs][UserFromToken][BfromToken][EncCtx][Signature]
  b.van?<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,StsNameA,
         SamlTs,UserFromToken,BfromToken,EncCtx,Signature>.
     ( sys.attack1!<messageSuppressed,SignedDoc>
       | sys.attack1?<messageSuppressed,SignedDoc>.
         -- To ensure that the officer performs his checks after that the
         -- attack has been performed, we activate here the officer.
         -- From now on, the officer can go to clinics A and B at any time
         officier.activate!<>
     )
```

```
-- This intruder models an healthcare professional sitting on clinic A
-- (e.g. a nurse) that wants to access restricted resources by
-- reusing an already issued SAML assertion
Intruder2(a,hashReq,hashResp,sign,signResp) =
  -- Suppress the correct message
  [A][MsgId1][Ts2][AfromDoc][User][Doc][SignedDoc][AfromToken][StsNameA]
  [SamlTs][UserFromToken][BfromToken][EncCtx][Signature]
  b.van?<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,StsNameA,
         SamlTs,UserFromToken,BfromToken,EncCtx,Signature>.
     ( sys.attack2!<>
       | sys.attack2?<>.
           ( -- Create the hash of the fake document
             hFunc.hashReq!<a,nurse,fakeDoc>
             | [FakeDochash] a.hashResp?<nurse,fakeDoc,FakeDochash>.
                 ( -- Sign the hash code: "a" is used also as creator name
                   signer.sign!<A,A,FakeDochash>
                   | [FakeSignedDoc] a.signResp?<FakeDochash,FakeSignedDoc>.
                         -- Send the fake message
                         b.van!<A,b,msgIdFake,300,AfromDoc,nurse,fakeDoc,
                               FakeSignedDoc,AfromToken,StsNameA,SamlTs,
                               UserFromToken,BfromToken,EncCtx,Signature>
                 )
           )
     )


-- This intruder is similar to Intruder2, but it reuses the SAML assertion
-- by using the correct username (i.e. 'user1')
Intruder3(hashReq,hashResp,sign,signResp) =
  [A][MsgId1][Ts2][AfromDoc][User][Doc][SignedDoc][AfromToken][StsNameA]
  [SamlTs][UserFromToken][BfromToken][EncCtx][Signature]
  b.van?<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,StsNameA,
         SamlTs,UserFromToken,BfromToken,EncCtx,Signature>.
     ( sys.attack3!<>
       | sys.attack3?<>.
           ( -- Create the hash of the fake doc
             hFunc.hashReq!<i,User,fakeDoc>
             | [FakeDochash] i.hashResp?<User,fakeDoc,FakeDochash>.
                 ( -- Sign the hash code: we use "I" as creator name
                   signer.sign!<i,i,FakeDochash>
                   | [FakeSignedDoc] i.signResp?<FakeDochash,FakeSignedDoc>.
                         -- Send the fake message
                         b.van!<i,b,msgIdFake,300,AfromDoc,User,fakeDoc,
                                FakeSignedDoc,AfromToken,StsNameA,SamlTs,
                                UserFromToken,BfromToken,EncCtx,Signature>
                 )
           )
     )


-- This intruder sends multiple time the same packet
-- for obtaining multiple times the same resource
Intruder4() =
  [A][MsgId1][Ts2][AfromDoc][User][Doc][SignedDoc][AfromToken][StsNameA]
  [SamlTs][UserFromToken][BfromToken][EncCtx][Signature]
  b.van?<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,StsNameA,
         SamlTs,UserFromToken,BfromToken,EncCtx,Signature>.
     ( sys.attack4!<>
       | sys.attack4?<>.
           ( -- Send multiple copies of the same message
             b.van!<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,
                    StsNameA,SamlTs,UserFromToken,BfromToken,EncCtx,Signature>
             |
             b.van!<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,
                    StsNameA,SamlTs,UserFromToken,BfromToken,EncCtx,Signature>
           )
     )



in
```

```
[enc_KbPub_Req#][enc_KbPub_Resp#][dec_KbSec_Req#][dec_KbSec_Resp#][enc_KaPub_Req#]
[enc_KaPub_Resp#][dec_KaSec_Req#][dec_KaSec_Resp#][sign_KaSec_Req#][sign_KaSec_Resp#]
[verify_KaPub_Req#][verify_KaPub_Resp#][sign_KstsaSec_Req#][sign_KstsaSec_Resp#]
[verify_KstsaPub_Req#][verify_KstsaPub_Resp#][sign_KstsbSec_Req#][sign_KstsbSec_Resp#]
[verify_KstsbPub_Req#][verify_KstsbPub_Resp#][sign_KiSec_Req#][sign_KiSec_Resp#]
[verify_KiPub_Req#][verify_KiPub_Resp#][hashReq#][hashResp#][qa1#][qb#]
( Cipher(enc_KbPub_Req,enc_KbPub_Resp,dec_KbSec_Req,dec_KbSec_Resp,enc_KaPub_Req,enc_KaPub_Resp,
         dec_KaSec_Req,dec_KaSec_Resp)
  | Signer(sign_KaSec_Req,sign_KaSec_Resp,verify_KaPub_Req,verify_KaPub_Resp,sign_KstsaSec_Req,
           sign_KstsaSec_Resp,verify_KstsaPub_Req,verify_KstsaPub_Resp,sign_KstsbSec_Req,
           sign_KstsbSec_Resp,verify_KstsbPub_Req,verify_KstsbPub_Resp,sign_KiSec_Req,sign_KiSec_Resp,
           verify_KiPub_Req,verify_KiPub_Resp)
  | Sha(hashReq,hashResp)
  | ClinicA(a1,qa1,stsa1,stsNameA1,user1,doc1,user2,doc2,sign_KaSec_Req,sign_KaSec_Resp,
            sign_KstsaSec_Req,sign_KstsaSec_Resp,verify_KstsaPub_Req,verify_KstsaPub_Resp,
            enc_KbPub_Req,enc_KbPub_Resp,hashReq,hashResp)
  | ClinicB(qb,dec_KbSec_Req,dec_KbSec_Resp,enc_KaPub_Req,enc_KaPub_Resp,sign_KstsbSec_Req,
            sign_KstsbSec_Resp,verify_KstsaPub_Req,verify_KstsaPub_Resp,verify_KaPub_Req,
            verify_KaPub_Resp,hashReq,hashResp)
  -- For attack 1
  -- | Officer(qa1,qb)
  --
  -- For attack 2
  -- | Intruder2(a1,hashReq,hashResp,sign_KaSec_Req,sign_KaSec_Resp)
  --
  -- For attack 3
  -- | Intruder3(hashReq,hashResp,sign_KiSec_Req,sign_KiSec_Resp)
)
-- For attack 1
-- | Intruder1()
--
-- For attack 4
--| Intruder4()

end


--------------
-- Attack 1 --
--------------
-- Abstractions{
--  Action sys.attack1<messageSuppressed,$signedDoc> -> attack1Performed(message,$signedDoc)
--  Action sys.attack1Detected<messageSuppressed,$signedDoc>
--         -> attack1DetectedByOfficer(message,$signedDoc)
-- }
--
-- Eventually an attack is performed
-- (this avoids that the second formula below is TRUE because there is no attack)
-- EF {attack1Performed(message,$doc)} true
-- Result: TRUE
-- (states generated= 906, computations fragments generated= 904)
--
-- In any way an attack is performed, in any case it will be eventually detected
-- AG [attack1Performed(message,$doc)] AF {attack1DetectedByOfficer(message,%doc)} true
-- Result: TRUE
-- (states generated= 20538, computations fragments generated= 50614)



--------------
-- Attack 2 --
--------------
-- Abstractions{
--  Action sys.attack2<> -> attack2Performed(AssertionReused)
--  Action sys.attack2Detected<$user,$userFromToken>
--         -> attack2DetectedByB(invalidUser,$user,differsFrom,$userFromToken)
-- }
--
-- EF {attack2Performed(AssertionReused)} true
-- Result: TRUE
-- (states generated= 906, computations fragments generated= 904)
--
```

```
--
-- AG [attack2Performed(AssertionReused)]
--    AF {attack2DetectedByB(invalidUser,$user1,differsFrom,$user2)} true
-- Result: TRUE
-- (states generated= 15543, computations fragments generated= 38134)




--------------
-- Attack 3 --
--------------
-- Abstractions{
--  Action sys.attack3<> -> attack3Performed(invalidSignature)
--  Action sys.attack3Detected<invalidSignature> -> attack3DetectedByB(invalidSignature)
-- }
--
-- EF {attack3Performed(invalidSignature)} true
-- Result: TRUE
-- (states generated= 906, computations fragments generated= 904)
--
--
-- AG [attack3Performed(invalidSignature)] AF {attack3DetectedByB(invalidSignature)} true
-- Result: TRUE
-- (states generated= 26287, computations fragments generated= 70366)



--------------
-- Attack 4 --
--------------
-- Abstractions{
--  Action sys.attack4<> -> attack4Performed(bounceAttack)
--  Action sys.attack4Detected<$ctx> -> attack4DetectedByB(invalidContext,$ctx)
-- }
--
-- EF {attack4Performed(bounceAttack)} true
-- Result: TRUE
-- (states generated= 906, computations fragments generated= 904)
--
--
-- AG [attack4Performed(bounceAttack)] AF {attack4DetectedByB(invalidContext,$ctx)} true
--
--- To verify this formula, it is suggested to reduce the model state space
--- by removing from 'Creator' the creation and sending of the second message
--
-- Result: TRUE
-- (states generated= 1506, computations fragments generated= 4426)
```