Formal Analysis of CWA 14890-1

Ashar Javed

Hamburg University of Technology (TUHH)
Hamburg, Germany
ashar.javed@tu-harburg.de

Abstract. Formal analysis is of importance in order to increase confidence that the protocol satisfies its security requirements. In particular, the results obtained from the formal analysis of the smart card security protocols when smart cards are used as a specific type of Secure Signature Creation Devices (SSCDs) are presented. SSCDs are developed to support the EU-directive on electronic signatures. In this paper, we focus on security properties, called the *authentication* and *secrecy*. The device authentication protocols mentioned in CWA 14890-1 are modeled using the high-level protocol specification language HLPSL and verified with the help of AVISPA tool. Our formal analysis does not reveal any weaknesses of the CWA 14890-1 protocol suite.

Keywords: smart cards, CWA 14890-1 authentication protocols, formal analysis, SSCDs, AVISPA

1 Introduction

CEN Workshop Agreement CWA 14890-1 [1] describes the European standardization activities and solutions for smart cards as a specific type of a Secure Signature Creation Device (SSCD). SSCDs means configured software or hardware which is used to manipulate the Signature Creation Data (SCD) [2]. SCD is unique data, such as codes or private cryptographic keys, which are used by the signatory to create an electronic signature. SSCDs are developed to support the EU-directive on electronic signatures. It builds on ISO/IEC 7816-4 [3]. The key issue of CWA 14890-1 is to enable interoperability, so that smart cards from different manufacturers can interact with different kind of signature creation applications [1]. CWA 14890-1 describes the following device authentication protocols:

- An Asymmetric Session Key Agreement Protocol with Privacy Protection
- An Asymmetric Session Key Transport Protocol based on RSA
- Symmetric Authentication Protocol

Formal analysis is of importance in order to increase the assurance that the protocol satisfies its security requirements. In this sense, the Automated Validation of Internet Security Protocols and Applications (AVISPA) and the Security Protocol Animator for AVISPA (SPAN) [6] tools have been used to validate device authentication protocols mentioned in CWA 14890-1.

Contribution. Our main contribution is to provide the first comprehensive formal analysis of CWA 14890-1. We explain formalization of the protocols in AVISPA's high-level protocol specification language HLPSL [7], and describe approach to verifying that the device authentication protocols in CWA 14890-1 does indeed satisfy the security requirements and are safe w.r.t. finite number of sessions in our analysis runs. Even though the CWA 14890-1 specs amount to over 150 pages, our formal models are 16 pages of HLPSL code. Of course, our model abstracts from some of the low-level details that are in the 150-odd pages CWA 14890-1 specs, e.g., bit-level selection of data but such abstraction seems crucial to keep an overview and understand the CWA 14890-1 standard as a whole.

Synopsis. The paper is organised as follows. In Section 2 a brief overview of AVISPA tool. Section 3 comments on smart cards. Section 4 is devoted to the description and specifications of an asymmetric session key agreement protocol. In Section 5, we describe the asymmetric key transport protocol. Section 6 comments on symmetric authentication protocol. Finally in Section 7 we give our conclusions.

2 The AVISPA Tool

The AVISPA tool is used for the Automated Verification of Internet Security Protocols and Applications [4,5]. High-Level Protocol Specification Language (HLPSL) is used to interact with the AVISPA tool. In HLPSL, protocol designer specify a security protocol along with the security requirements that should be met. HLPSL is role based formal specification language. The AVISPA tool automatically translates (via HLPSL2IF Translator) a security protocol into an equivalent description written in the rewriting-based formalism known as Intermediate Format IF [8].

The current version of the tool integrates four back-ends: the On-the-fly Model-Checker OFMC [9], the Constraint-Logic-based Attack Searcher CL-AtSe [10], the SAT-based Model-Checker SATMC [11], and the Tree Automata tool based on Automatic Approximations for the Analysis of Security Protocols analyzer TA4SP [12]. All the back-ends of the tool analyze protocols under the assumptions of perfect cryptography and that the protocol messages are exchanged over a network that is under the control of a Dolev-Yao intruder [13].

OFMC and CL-AtSe are dedicated to the *refutation* of protocols and focus on a finite number of sessions. The analysis with the SATMC and TA4SP back-ends were always 'Inconclusive' in our experiments. Both back-ends do not support `modulus` and `xor` operators, thats why analysis was always 'Inconclusive'. The device authentication protocol mentioned in CWA 14890-1 make use of `modulus` and `xor` operators.

Space does not permit discussion on back-ends. We refer to the AVISPA's user manual (google avispa-project) for deeper concerns about AVISPA. *From now on we will only consider OFMC and CL-AtSe back-ends for the analysis of protocols with AVISPA tool.*

3 Smart Card Commands

Following smart card commands specified in ISO/IEC 7816-4 [3] suffice to cover all smart cards related authentication protocols.

1. *Manage Security Environment (MSE)* command informs the smart card about the encryption algorithms, signature algorithms, hash algorithms and keys to be used in the subsequent command sequence.
2. The *Get Challenge* command requests challenge (e.g., random number) from the smart card.
3. The *Get Data / Read Binary* command is used for the retrieval of data object(s) e.g., serial number.
4. *Perform Security Operation (PSO)* command is used to compute hash and digital signature. The data to be hashed or the final digest respectivly are sent to the card with the *PSO*.
5. The *Internal Authenticate* is used when a smart card has to authenticate itself, the terminal sends an *Internal Authenticate* Command.
6. When a terminal has to authenticate itself, the smart card expects an *External Authenticate* command containing an authentication token.
7. Finally the *Mutual Authenticate* command combines *Internal Authenticate* and *External Authenticate* into one command.

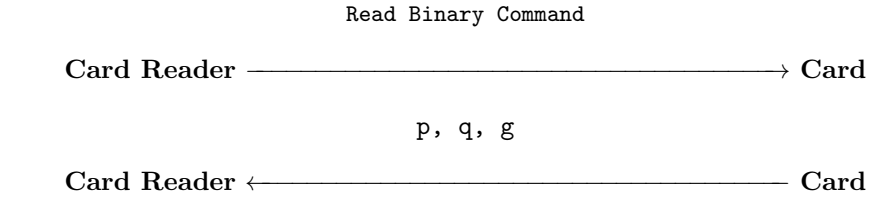
4 An Asymmetric Session Key Agreement Protocol with Privacy Protection

Key agreement is the process of establishing a shared secret key between two entities A and B in such a way that neither of them can predetermine the value of the shared secret key. [1].

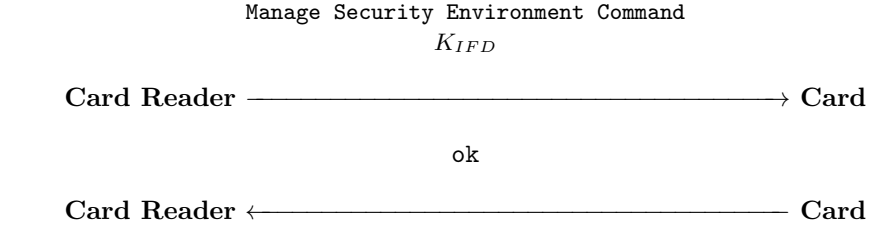
In an asymmetric session key agreement protocol with privacy protection, to avoid the card disclosing private information, such as identity, a secure channel session is established before any other operation. To do so, the protocol starts with an *unauthenticated Diffie-Hellman key exchange* [15] and then authenticates the Interface Device (IFD) before the Integrated Circuit Card (ICC). In device authentication protocol card reader (i.e., IFD) can authenticate itself to smart card (i.e., ICC) without having to know smart card's identity. The following section shows the general flow of device authentication protocol.

4.1 Authentication Steps

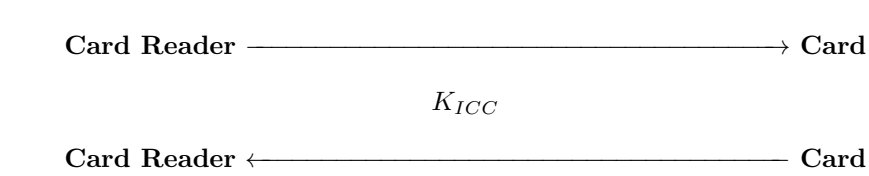
Step 1. The reader starts the protocol by sending the *Read Binary* command. With the help of *Read Binary* command IFD reads the public key quantities from the file. For instance, in a Diffie Hellman Key exchange scheme the public key quantities would be the public parameters p, q and g. The public key quantities reveal information about the authentication mechanism. As long as these quantities are used in many ICCs, the identity of an ICC cannot be determined from this information.



Step 2. After receiving the public parameters, card reader chooses random number a with $1 \leq \mathtt{a} \leq \mathtt{q}$ - 1, computes a key token $K_{IFD} = g^a$ mod p and sends the key token to smart card with the help of *Manage Security Environment* command in order to establish a secure session. 'OK' as response from smart card after receiving the key token i.e., K_{IFD}.



Step 3. The opposite key token/portion K_{ICC} is returned in the response of a *Get Data* command. Upon receiving the *Get Data* command smart card computes $K_{ICC} = g^b$ mod p and transmits key token K_{ICC} to card reader.



Step 4. At this point, neither card reader nor smart card has revealed his identity. The reader and the card have completed simple unauthenticated Diffie-Hellman key agreement. Neither side has been authenticated yet, but by using the common secret i.e., K_{IFDICC}, they now derive an encryption key Kenc and a MAC key Kmac that will be used to protect the remainder of the authentication protocol from casual eavesdroppers. Both keys are 112-bit 3DES keys. Note that there could still be a man-in-the-middle at this point in the protocol. If we model authentication (as an experiment) with the help of witness and request goal facts (see section 4.5) at this point then AVISPA also finds man-in-the-middle. It does not make sense to model authentication at this point of the protocol. The reason is that man in the middle attacker could have made contact with the IFD by himself – he did not even need to be in the middle as the protocol involves an anonymous DH-Key Exchange. Both the reader and the card calculate:

$$\text{HASH1} = \text{HMAC}[K_{IFDICC}]\ (1)$$
$$\text{HASH2} = \text{HMAC}[K_{IFDICC}]\ (\text{HASH1} \parallel 2)$$

112 bits are selected from HASH1 to produce `Kenc`, and 112 bits are selected from HASH2 to produce `Kmac`. After generation of the encryption and MAC keys reader sends the *Manage Security Environment* command. The MSE command sets the key reference of the public key of trusted certification authority to be used for the verification of the IFD's authentication certificate. 'OK' as response from the smart card because we assume that key of trusted certification authority is present in card.

`Manage Security Environment Command`

Card Reader ⟶ **Card**

`ok`

Card Reader ⟵ **Card**

Step 5. Upon receiving 'OK', reader now sends its certificate to the card by encrypting it with `Kenc`. The MAC of the encrypted certifcate is also send to the card with the help of *Perform Security Operation* command. The card will verify the certificate with the help of public key of certification authority. After verification card will send 'OK' as response.

`Perform Security Operation Command`
`({Reader.PKifd}_Kenc).Hash(Kmac, ({Reader.PKifd}_Kenc))`

Card Reader ⟶ **Card**

`ok`

Card Reader ⟵ **Card**

Step 6. Upon receiving 'OK' from the smart card, the reader requests a 64 bits (8 bytes) random number (`RND_ICC`) from the smart card with a *Get Challenge* command in order to prove its authenticity dynamically. Upon receiving *Get Challenge*, the smart card generates a 64 bits random number used as nonce. The smart card stores random number in its internal memory and replies with random number to the reader.

`Get Challenge Command`

Card Reader ⟶ **Card**

`RND_ICC`

Card Reader ⟵ **Card**

Step 7. The IFD computes a signature on the concatenation of the challenge (`PRND2`) with its own key token (K_{IFD}) information. The signature is a signature with message recovery, so all parameters in the signature can be considered to be recoverable. The Diffie-Hellman key parameters (`DH.P`) are part of the signature in order to provide authenticity of the parameters. The MAC of the

encrypted signature is also transmitted to the card with the help of *External Authenticate* command. Card now verifies the MAC, decrypts, and verifies the signature using private key generated in step 4. At the conclusion of this step, card has authenticated reader and knows that K_{IFD} and K_{ICC} are fresh and authentic. `6A` and `BC` are hexadecimal numbers. After verification card sends 'OK' as response.

`External Authenticate Command`

`SIG = 3DESEncrypt Key (6A || PRND2 || hash[PRND2 || KIFD || SN_IFD || RND_ICC || KICC || DH.P] || BC)`

Card Reader ⟶ **Card**

`ok`

Card Reader ⟵ **Card**

Step 8. Upon receiving 'OK', reader sends *Read Binary* Command in order to get the ICC's encrypted certificate plus MAC of encrypted certificate. However at this point, while card knows there is no man-in-the-middle because card checked the signature from reader, reader does not know whom he is talking to, and hence is unsure if there may be a man-in-the-middle attack, thats why he demanded the certificate.

`Read Binary Command`

Card Reader ⟶ **Card**

`({Card.PKicc}_Kenc).Hash(Kmac, ({Card.PKicc}_Kenc))`

Card Reader ⟵ **Card**

Step 9. Before processing the *Internal Authenticate* command the ICC's private authentication key must be set by the *Manage Security Environment* command. The MSE command updates the current security environment.

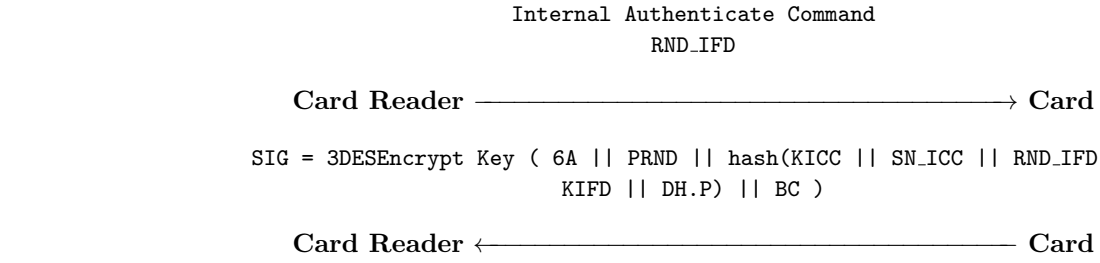
`Manage Security Environment Command`

Card Reader ⟶ **Card**

`ok`

Card Reader ⟵ **Card**

Step 10. The IFD performs an *Internal Authenticate* command. The challenge (`PRND`) sent to the ICC with this command is `RND_IFD`. The ICC then computes the signature over the challenge and the key token K_{IFD}, K_{ICC} and returns it to the IFD encrypted with secure messaging. The signature is a signature with message recovery, so all parameters in the signature can be considered to be recoverable. Reader can now verify the MAC and decrypt signature. The IFD verifies the certificate using the public key of the trusted certification authority.



4.2 HLPSL Specifications

In this section we show the fragments of HLPSL specifications of protocol. Our intended target is to model the protocol as close as possible to the protocol description given in 4.1. The protocol behavior could be modeled, except next issues / limitations. The detailed HLPSL model can be found in [18]. We refer to the AVISPA's user manual (google avispa-project) for deeper concerns about HLPSL.

Restrictions Applied:

- Our HLPSL model abstracts from some of the low-level details that are in the protocol specifications, e.g., bit-level selection of data. Such abstraction seems crucial to keep an overview and understand the protocol as a whole. During the generation of encryption and MAC keys, protocol make use of bit-level selection of data. This could not be mapped in HLPSL specifications.
- Protocol includes provisions for the optional exchange of chain of public-key certificates. This is not included in the model.
- We assume that appropriate public key of the trusted certification authority is present in IFD and in ICC.

4.3 Roles

In order to describe the protocol we should specify the actions of each kind of participant, i.e., the basic roles. Roles are independent processes: they have a name, receive information by parameters and contain local declarations. Basic roles are played by an agent whose name is received as parameter. The actions of a basic role are transitions, describing changes in their state depending on events or facts. To describe protocol in HLPSL we introduce two basic roles: ifd and icc. We now present the declaration of basic roles and their (typed) parameters in HLPSL:

```
role ifd (  Reader, Card: agent, PKifd : public_key, Hash, SHA1 : hash_func,
            SND, RCV: channel (dy) ) played_by Reader def=
  local
        State, C1, C2      : nat,
        PKicc : public_key,
        KICC, KIFD : symmetric_key,
```

```
        ...
role icc (  Reader, Card: agent, PKicc : public_key, Hash, SHA1 : hash_func,
            SND, RCV: channel (dy)) played_by Card def=
  local
        State, C1, C2 : nat,
        PKifd : public_key,
        KICC, KIFD : symmetric_key,
        ...
```

Here we have: `Reader`, `Card` are agents playing roles ifd and icc respectively. `Hash`, `SHA1` are hash functions used in calculating MAC and in session key generation respectively. `PKifd`, `PKicc` are public keys of Reader and Card respectively. `SND`, `RCV` are channels for sending and receiving messages. In HLPSL variable names start with capital letters; constants, keywords[1] and data types[2] start with lower-case letters.

4.4 Composed Roles

Composed roles instantiate one or more basic roles, "**gluing**" them together so that they execute together, usually in parallel (with interleaving semantics) [7]. Composed roles have no transition section rather a composition section in which the basic roles are instantiated. The composition is presented in HLPSL.

```
role session( Reader, Card: agent,
              Hash, SHA1 : hash_func,
              PKifd, PKicc : public_key )
def=
  local SIFD, RIFD, SICC, RICC: channel (dy)
  composition
        ifd(Reader,Card,PKifd,Hash,SHA1,SIFD,RIFD)
     /\ icc(Reader,Card,PKicc,Hash,SHA1,SICC,RICC)
end role
```

Symbol $\wedge$ denotes here a parallel execution. A transition is a rule that can be fired if the left-hand side is satisfied i.e., before symbol `=|>`.

4.5 Role IFD

For reasons of space we show the transitions of the role ifd that are more important as compared to the other one because these transitions contain the sending of the key token of the reader i.e., K_{IFD}, signed certificate along with the *External Authenticate* Command.

```
2.  State   = 2 /\ RCV(G'.P'.Q') =|>
%% Manage Security Environment Command. The detailed syntax of MSE command
%%can be found in CWA 14890-1. A is a random number that lies between 1 and q-1
 State' := 4 /\ A' := new()
%% KIFD' is a Key Token of Reader used in generation of Mutual Key i.e., KIFDICC
```

[1] role, played_by and def = are keywords in HLPSL.

[2] agent, public_key, hash_func and channel are data types in HLPSL. dy is the attribute of channel type and it represents Dolev-Yao.

```
/\ KIFD' := exp(G',A')
/\ SND(two_two.four_one.a_6.l_a_6.KIFD')
8.  State   = 8 /\ RCV(ok) =|>
%% Perform Security Operation Command. The detailed syntax of MSE command
%%can be found in CWA 14890-1. {Reader.PKifd}_Kenc is a certificate encrypted
%%with encryption key i.e., Kenc & contains the agent name
%%of reader & its public key. Reader now transmits it together with its MAC to Card.
State' := 10 /\
SND (two_a.zero_zero.a_e.(({Reader.PKifd}_Kenc). Hash(Kmac,({Reader.PKifd}_Kenc))))
12. State   = 12  /\ RCV(RND_ICC') =|>
%% EXTERNAL AUTHENTICATE COMMAND
State' :=  14 /\ PRND' := new()
%% SIG = 3DESEncrypt Key (6A || PRND2 || h[PRND2 || KIFD
%% || SN.IFD || RND.ICC || KICC || DH.P || BC)
/\     SND(eight_two.zero_zero.zero_zero.{({six_a.PRND'.KIFD.sn_ifd.
       RND_ICC'.KICC.(G.P.Q).b_c}_inv(PKifd))}_Kenc.
       Hash(Kmac,{({six_a.PRND'.KIFD.sn_ifd.
       RND_ICC'.KICC.(G.P.Q).b_c}_inv(PKifd))}_Kenc ))
/\ witness(Reader,Card,ifd_icc_run_id_for_authentication_of_ifd,RND_ICC')
```

This first transition in the above code fragment is called `'2.'`, though the names of the transitions serve merely to distinguish them from one another. It specifies that if the value of State is equal to 2 and a message (Diffie-Hellman Public Quantities) is received on channel `RCV`, then a transition fires which sets the new value of State to 4 and sends the *Manage Security Environment Command* on channel `SND`. HLPSL uses dot operator to denote the concatenation of messages as you see in the `RCV` channel above. Comments in HLPSL begin with the % symbol and continue to the end of the line. In any transition, the old value and the new value of a variable are syntactically distinguished: the prime symbol ' has to be attached to the name of a variable for considering its new value. Prime notation stems from the temporal logic TLA [17], upon which HLPSL is based. It is important to realise that the value of the variable will not be changed until the current transition is complete. So, the right-hand-side tells us that the value of the State variable, after transition `'2.'` fires, will be 4.

The second transition in HLPSL model above states that: if the value of variable State equals to 8 and we receive on channel `RCV ok` then reader sends the *Perform Security Operation Command* in which reader sends its certificate (Identity plus public key) encrypted with encryption key i.e., `Kenc`. The MAC of the encrypted certificate is also sent so that card can check the integrity with the help of MAC key (i.e., `Kmac`) also available at the card side". HLPSL uses the same notation for the encryption and decryption i.e., {`Message`}_`Key`.

Another transition is: The IFD computes a signature on the concatenation of the challenge with its own key token (K_{IFD}) information. The signature is a signature with message recovery, so all parameters in the signature can be considered to be recoverable. The Diffie-Hellman key parameters are part of the signature in order to provide authenticity of the parameters. The MAC of the encrypted signature is also transmitted to the card with the help of *External Authenticate Command*. Note that for the analysis tool as well as for the modeling in the tool's language, a signature is equivalent to an encryption with a private key i.e., {`Message`}_`inv(PublicKey)`.

4.6 Modelling Authentication in HLPSL

Authentication, security property, is modelled by means of several goal predicates in HLPSL: witness(agent,agent,protocol_id,message), request(agent,agent,protoc ol_id,message) and wrequest(agent,agent,protocol_id,message). The `witness` and `request` are goal facts related to authentication. The last line of the transition in above section 4.5 named `'12.'` is an authentication related event `witness`. Here it should be read as follows: means that honest agent Reader wants to execute the protocol with agent Card by using (RND_ICC') as value for the authentication identifier ifd_icc_run_id_for_authentication_of_ifd. Goal facts `witness` and `request` are used to check that a principal is right in believing that its intended peer is present in the current session, has reached a certain state, and agrees on a certain value, which typically is fresh. They have identical third parameter and it should be declared as a constant of type `protocol_id` in the top-level role. The label ifd_icc_run_id_for_authentication_of_ifd (of type `protocol_id`) is used to identify the goal. The third parameter is used to associate the `witness` and `request` predicates with each other and to refer to them in the goal section. There is also `wrequest` which corresponds to weak authentication (also called non-injective agreement according to Lowe's paper [16]). No replay protection is imposed if one uses `wrequest`.

Goal fact `request` means that agent Card accepts the value (TEMP_RND_ICC') and now relies on the guarantee that agent Reader exists and agrees with him on the value (TEMP_RND_ICC') for the authentication identifier ifd_icc_run_id_for_a uthentication_of_ifd. Goal fact `request` is used for strong authentication (also called injective agreement according to Lowe's paper [16]). In general, the authenticated issues a `witness` fact as soon as possible in his protocol execution, i.e., as soon as he has known the name of the authenticator and the payload (the data that shall be agreed upon) message. The authenticator issues in his last rule a `request` fact, i.e., when he has executed the protocol to the end during the session from his point of view. Also authentication goals have a direction, namely from an **authenticator to an authenticated**, in the sense that the authenticator convinces himself of the identity of the authenticated.

4.7 Role ICC

Now we show role icc's transition responsible for the receiving of the key token and signature from the card reader:

```
3.  State  = 3 /\ RCV(two_two.four_one.a_6.l_a_6.KIFD')  =|>
    State':= 5 /\ SND(ok)
13. State  = 13  /\
RCV(eight_two.zero_zero.zero_zero.{({six_a.PRND'.KIFD'.
    TEMP_SN_IFD'.TEMP_RND_ICC'.KICC'.
    (G'.P'.Q').b_c}_PKifd)}_Kenc.Hash(Kmac,{({six_a.PRND'.
    KIFD'.TEMP_SN_IFD'.TEMP_RND_ICC'.KICC'.
   (G'.P'.Q').b_c}_PKifd)}_Kenc ))
%%  Verify the random number
/\ RND_ICC = TEMP_RND_ICC'
%% Card has now authenticated Reader and knows that KIFD and KICC are fresh and authentic.
/\ KIFD = KIFD'
```

```
/\ KICC = KICC'
=|>
State':= 15  /\ SND (ok)
/\ request(Card,Reader,ifd_icc_run_id_for_authentication_of_ifd,TEMP_RND_ICC')
```

The first transition in the above code fragment of role icc shows the reception of the key token of the IFD i.e., K_{IFD} used in session key generation. After reception of the key token card sends ok.

The second transition shows the reception of the signature. It also shows that at this point of time agent card is verifying the random number that he generated earlier against the random number received as a part of signature. = is comparison operator in HLPSL. Card also checks the authenticity of the key tokens. After checking the authenticity of key tokens, card issued authentication related event `request` described in section 4.6.

4.8 Role Environment

A top-level role is always defined. This role contains global constants and a composition of one or more sessions, where the intruder may play some roles as a legitimate user. Here we also define an initial intruder knowledge set using `intruder_knowledge` token. Initially the intruder knows all agents' names along with their public keys (`Pkifd` and `Pkicc`), hash functions (`h` and `sha1`) and his public and private keys i.e., `ki` and `inv(ki)` respectively. The constant `i` is used to refer to the intruder. One should introduce a goal section to define security goals and to look for an attack. The authentication properties to be checked are listed in the goal section. The `authentication_on` keyword specify authentication goal with replay protection i.e., freshness of the agreement (or session) between the two, and directly corresponds to Lowe's injective agreement [16].

```
goal
authentication_on ifd_icc_run_id_for_authentication_of_ifd
authentication_on icc_ifd_run_id_for_authentication_of_icc
end goal
environment()
```

4.9 Automatic Analysis of the Protocol

The analysis with the AVISPA tool is performed on the following parallel sessions scenarios of the protocol.

Man-in-the-Middle Attack Scenario
In our first experiment, we consider a configuration: One session between agents reader and card and one session between card and reader in order to check for *man-in-the-middle* attack.

```
session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
```

Replay Attack Scenario
In our second experiment, we consider a configuration: One normal session between agents reader and card and in order to check for *replay attacks* we repeat the normal session.

```
session(reader,card,h,sha1,pkifd,pkicc)
∧ session(reader,card,h,sha1,pkifd,pkicc)
```

Impersonating Attack Scenarios
In our third experiment, the analysis is also performed on the set of configurations where an intruder (represented by `i`) is playing the role of legitimate agent(s) in order to attack the protocol. In AVISPA we can define an explicit intruder knowledge set using `intruder_knowledge` token. In the following analysis scenario intruder is *impersonating* `card`.

```
session(reader,card,h,sha1,pkifd,pkicc)
  ∧ session(reader,i,h,sha1,pkifd,ki)
```

In our forth experiment, the analysis is performed on parallel sessions where intruder is *impersonating* `reader` in order to attack the protocol.

```
session(reader,card,h,sha1,pkifd,pkicc)
   ∧ session(i,card,h,sha1,ki,pkicc)
```

Experiments
From now on we tested different parallel session scenarios until the results were exhaustive. In our fifth experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. One session between reader and intruder.

```
session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
  ∧ session(reader,i,h,sha1,pkifd,ki)
```

In our sixth experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. One session between intruder and card.

```
session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
   ∧ session(i,card,h,sha1,ki,pkicc)
```

In our seventh experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. One session between reader and intruder. One session between intruder and card.

```
session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
  ∧ session(reader,i,h,sha1,pkifd,ki)
   ∧ session(i,card,h,sha1,ki,pkicc)
```

In our seventh experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. Two sessions between reader and intruder. One session between intruder and card.

```
session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
  ∧ session(reader,i,h,sha1,pkifd,ki)
  ∧ session(reader,i,h,sha1,pkifd,ki)
   ∧ session(i,card,h,sha1,ki,pkicc)
```

In our eight experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. Session between reader and intruder. Two sessions between intruder and card.

```
 session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
   ∧ session(reader,i,h,sha1,pkifd,ki)
    ∧ session(i,card,h,sha1,ki,pkicc)
    ∧ session(i,card,h,sha1,ki,pkicc)
```

In our ninth experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. Two sessions between reader and intruder. Two sessions between intruder and card.

```
 session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
   ∧ session(reader,i,h,sha1,pkifd,ki)
   ∧ session(reader,i,h,sha1,pkifd,ki)
    ∧ session(i,card,h,sha1,ki,pkicc)
    ∧ session(i,card,h,sha1,ki,pkicc)
```

In our last experiment, we consider a configuration: Two parallel sessions between agents reader and card. One session between card and reader. Two sessions between reader and intruder. Two sessions between intruder and card.

```
 session(reader,card,h,sha1,pkifd,pkicc)
∧ session(reader,card,h,sha1,pkifd,pkicc)
∧ session(card,reader,h,sha1,pkicc,pkifd)
   ∧ session(reader,i,h,sha1,pkifd,ki)
   ∧ session(reader,i,h,sha1,pkifd,ki)
    ∧ session(i,card,h,sha1,ki,pkicc)
    ∧ session(i,card,h,sha1,ki,pkicc)
```

This is actually the limit of the analysis tool (i.e., AVISPA's OFMC and CL-AtSe) for this protocol: with more parallel sessions, no answer comes. While it may be interesting to use parallel computing to raise this limit, we think that the analyzed scenarios are the most relevant ones for the smart card protocol. Our analyzed scenarios fall into the following categories: Man in the middle attacks, replay attacks, impersonation and parallel session attacks attacks. In all experiments, the security properties under test are the correspondence properties explained above in section 4.6. **For all analyzed scenario, no attacks were found on the protocol in the presence of DY model.** According to [14],

> *"Finding an attack for a protocol with a fixed number of sessions is a NP-complete*[3] *problem with respect to a Dolev-Yao model [13] of intruders. Results does not assume a limit on the size of messages intruder can generate."*

Table 1 shows the summary of validation results using AVISPA.

[3] The complexity class NP (Non-deterministic Polynomial time) is the set of all decision problems solvable in polynomial time on a non-deterministic Turing machine.

AVISPA Tool Summary	
Backend Tool	**Result**
OFMC	Safe
CL-AtSe	Safe

Table 1. AVISPA Validation Results.

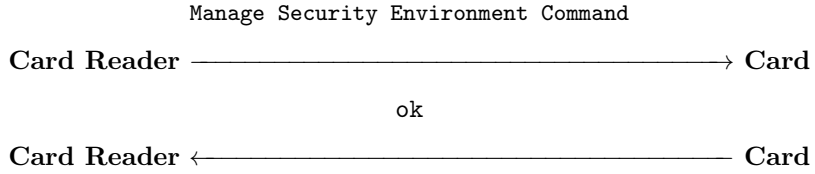
5 Specification and Analysis of Asymmetric Key Transport Scheme based on RSA

Key Transport is the process of transferring a secret key, chosen by one entity (or a trusted center), to another entity, suitably protected by asymmetric techniques. Key transport requires encryption of the key part, this is typically done with the public key of the counterpart [1].

The protocol mentioned is favorable for situations where the ICC should be authenticated prior to the IFD. It is appropriate to application fields where privacy is not a key issue but simplicity is required and/or the ICC is to be authenticated first. The Key Transport Protocol also specifies that smart card and reader agree on two session keys to protect subsequent communications. In smart card applications, one key is used for encryption/decryption operations while the second key is used to compute message authentication codes (MAC). CWA 14890-1 specifies two key triple DES (2KTDES) in cipher block chaining mode with fixed initialization vector 0 as encryption method. The length of challenges is specified as 64 bits. Additionally, the procedure to establish session keys is defined and the length of the key derivation data is set to 256 bits.

5.1 Authentication Steps

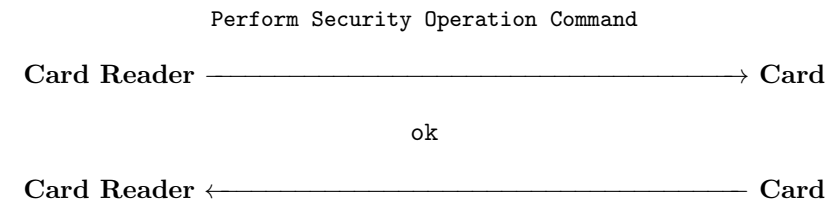
Step 1. The reader starts the protocol with a `Manage Security Environment` command informing the smart card which key reference will be used for subsequent messages. For simplicity we assume that the key reference is for the public key of the trusted certification authority. The public key is used for the verification of the IFD's certificate. 'OK' will be the ICC's response if referenced public key of the certification authority is present in card and in this case it is there.



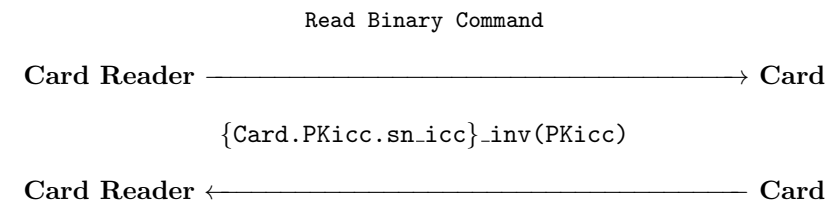
Step 2. The reader next sends the `Perform Security Operation` command in which reader sends the certificate to the smart card. The certificate is encrypted / signed[4] with the private key of the reader and it has the identity of

[4] Note that for the analysis tool as well as for the modeling in the tool's language, a signature is equivalent to an encryption with a private key i.e., {`Message`}`_inv(PublicKey)`.

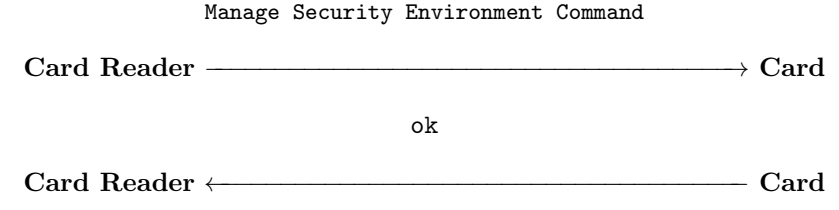
the sender along with its public key . The card verifies the certificate using the public key whose reference was received in the previous step. After verification card sends 'OK' as response. At this point of time the public key of the IFD is now known by the ICC, and can be trusted.



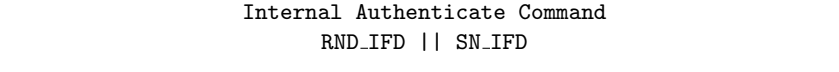
Step 3. Next, the reader fetches the serial number of the smart card along with the card certificate that contains the identity of the card(i.e., `Card`) along with its public key (i.e., `PKicc`) encrypted / signed by the help of card's private key (`inv(PKicc)`) by sending the `Read Binary` command. The IFD verifies the certificate using the public key of the trusted certification authority. The public key of ICC is now known by the IFD, and can be trusted.



Step 4. Next the `Manage Security Environment` command updates the current security environment by setting the ICC's private authentication key. Furthermore, the IFD's public key needs to be selected for encryption in order to transport the ICC's key contribution data i.e., `KICC`. `KICC` is a 32 byte random number generated by the ICC. Keys are set by sending the key references (private key of card plus public key of the reader) to the card. Card will send 'OK' as response.



Step 5. Upon receiving 'OK' from smart card, the reader generates its own 64 bits random number and sends random number along with its serial number in the `Internal Authenticate Command`.



Card Reader ⟶ **Card**

Upon receiving the `Internal Authentication` command, the smart card then computes the digital signature. Generate the padding random number (`PRND1`) and concatenates with the random number generated for the key derivation (`KICC`) and takes the hash of padding random number, random number, reader's serial (`SN_IFD`) and random number (`RND_IFD`), encrypts the concatenated data using its private key and sends the encrypted data to the reader. The reader decrypts the data with the public key and checks whether the second random number equals the one stored in its internal memory and whether the second serial number matches its own. After verification the reader authenticated the card.

```
SIG = DS[Private Key.ICC.AUT](6A || PRND1 || KICC || hash(PRND1 ||
                        KICC || C) || BC)
                    where C = SN_IFD || RND_IFD
```

Card Reader ⟵ **Card**

Step 6. Then, the reader requests a 64 bits (8 bytes) random number (`RND_ICC`) from the smart card with a `Get Challenge` command in order to prove its authenticity dynamically. Upon receiving `Get Challenge`, the smart card generates a 64 bits random number used as nonce. The smart card stores random number in its internal memory and replies with random number to the reader.

`Get Challenge Command`

Card Reader ⟶ **Card**

`RND_ICC`

Card Reader ⟵ **Card**

Step 7. Upon receiving `Get Challenge`, the reader computes the signature. Generates a padding random number (`PRND`) and concatenates with `KIFD` i.e., a 32 byte random number generated by the IFD for key derivation / key token and takes the hash of padding random number, random number, card's serial (`SN_ICC`) and random number (`RND_ICC`), encrypts the concatenated data using its private key and sends the encrypted data to the card with the help of `External Authenticate` command. `External Authenticate` command delivers the digital signature of the IFD to the card.

```
                    External Authenticate Command
where SIG = DS [Private Key.IFD.AUT] (6A||PRND||KIFD||hash(PRND.||KIFD||
                        RND_ICC||SN_ICC)||BC)
```

Card Reader ⟶ **Card**

Upon receiving the `External Authentication` command, smart card decrypts the data with the public key of the counterpart and checks whether the second random number equals the one stored in its internal memory and whether the second serial number matches its own. After verification the card now authenticated the reader.

Once smart card and reader have stored both their own and received key derivation data, they can generate the session keys. CWA 14890-1 specifies two key triple DES (2KTDES) as the algorithm to be used for encryption, decryption, and integrity protection. Therefore, four 8 byte keys have to be generated. First, both protocol participants XOR the key derivation data of reader and card resulting in a value SK (session key). Then, two 32 bit counters are appended to SK, resulting in SK1 and SK2. The value of the first counter is 1, the value of the second is 2. Each protocol participant then calculates hash values of SK1 and SK2. CWA 14890-1 stipulates SHA-1 as hash algorithm. The first 8 bytes of SHA-1(SK1) are used as the first encryption key; the second 8 bytes are used for the second encryption key, while the last four bytes of the hash value are not used. The value of SHA-1(SK2) is used similarly. The first 16 bytes are used for both integrity keys and the last four bytes are not used. These keys are stored in the internal state of both communication partners. From now on, these keys can be used to secure further communications.

5.2 HLPSL Specifications

HLPSL specifications of Asymmetric Session Key Transport Protocol have been omitted for lack of space. Our intended target is to model the protocol as close as possible to the protocol description given in 5.1. The protocol behavior could be modeled, except issues / limitations as described in section 4.2. The detailed HLPSL model can be found in [18].

5.3 Modeling Secrecy in HLPSL

Secrecy of a message means that the specified set of agents can see the message. HLPSL supports secrecy goal with the help of predefined predicate `secret`. In key transport protocol and in symmetric authentication protocol (see section 6), both roles i.e., `ifd` and `icc` creates session key (`SK`), and so we augment transitions of both roles in both protocols, with the following `secret` facts where the primes is required there to refer to the new values of `SK'`:

```
secret(SK',sec_ifd_session_key,{Reader,Card}
secret(SK',sec_icc_session_key,{Card,Reader}
```

The labels `sec_ifd_session_key` and `sec_icc_session_key` are of type `protocol_id`. They must be declared in the `const` section of the `environment` role. In the case of secrecy facts, *protocol ids* serve merely to distinguish different secrecy goals.

5.4 Security Analysis

For security analysis our methodology for experiments 1-10 were the same as we did in section 4.9. **For all analyzed scenario, no attacks were found on the**

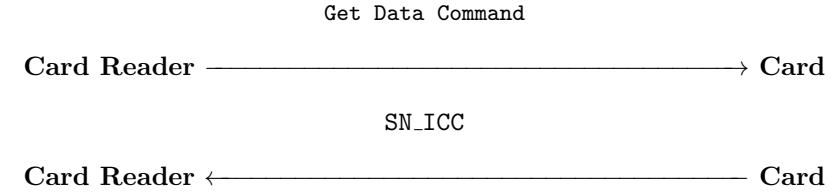
key transport protocol in the presence of DY model. In all experiments, the security properties under test are the correspondence properties explained in section 4.6 and secrecy properties explained above in section 5.3 for this protocol.

6 Specification and Analysis of Symmetric Authentication Scheme

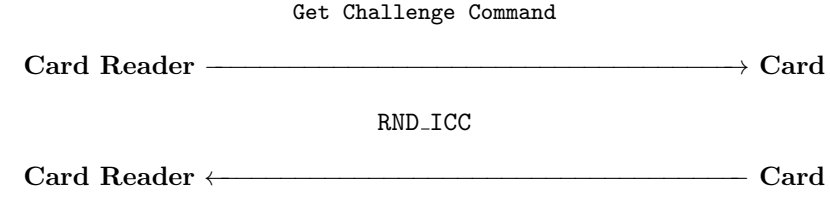
The symmetrical authentication protocol may be used to construct a *secure channel* between an application and a signature creation device providing either only integrity or both integrity and confidentiality. The long term shared secret keys i.e., `Kenc` (encryption key) and `Kmac` (message authentication code/integrity key) are supposed to be already present in the IFD (reader) and ICC (smart card) and are used in authentication protocol. The long term shared secret keys are replaced as soon as a fresh session keys are available.

6.1 Authentication Steps

Step 1. The IFD requires the ICC's identity serial number (`SN_ICC`) for the mutual authentication token. The card reader starts the protocol with the help of `Get Data Command` by fetching the serial number (`SN_ICC`) of the smart card and stores it for later use.



Step 2. Then, the reader requests a 64 bits (8 bytes) random number (`RND_ICC`) from the smart card with a `Get Challenge` command in order to prove its authenticity dynamically. Upon receiving `Get Challenge`, the smart card generates a 64 bits random number used as nonce. The smart card stores random number in its internal memory and replies with random number to the reader.



Step 3. Once the reader receives random number of the smart card, it generates its own 64 bits random number. Further 256 random bits are selected for use as key derivation data. The reader stores key derivation data in its internal memory. Next, the reader generates the command data for a `Mutual`

`Authenticate` command. The `Mutual Authenticate` command authenticates ICC and IFD in one command. It concatenates random and serial numbers of reader with random and serial numbers of card and key derivation data of reader, and encrypts the resulting string under the encryption key selected in the initial `Manage Security Environment` command. Last, the reader generates a MAC of the encrypted data using the selected integrity key. Then, the reader sends the `Mutual Authentication` command with the command data just generated to the smart card.

```
Mutual Authenticate Command
E[Kenc](S) || MAC[Kmac]( E[Kenc](S) )
where S = RND_IFD || SN_IFD || RND_ICC || SN_ICC || KIFD
```

Card Reader $\longrightarrow$ **Card**

Upon receiving the `Mutual Authentication` command, the smart card first checks the integrity of the message. If it can confirm command data integrity, the command data are decrypted. Next, the smart card checks whether the second random number has the same value as the random number stored in the card's internal memory, and whether the second serial number equals its own serial number. If these two tests are successful, the card stores the key derivation data in its internal memory. Now, the smart card selects its own 256 bits key derivation data and stores it in its internal memory, concatenates random and serial numbers of card with random and serial numbers of reader and key derivation data of the card, encrypts the concatenated data using the previously selected encryption key, and calculates a MAC over the encrypted data using the integrity key. The smart card then sends the encrypted data and MAC back to the reader. After verifying the MAC value, the reader decrypts the response and checks whether the second random number equals the one stored in its internal memory and whether the second serial number matches its own.

```
E[Kenc](R) || MAC[Kmac](E[Kenc](R))
R = RND_ICC || SN_ICC ||RND_IFD || SN_IFD || KICC
```

Card Reader $\longleftarrow$ **Card**

Once smart card and reader have stored both their own and received key derivation data, they can generate the session keys. The procedure to generate session keys is same as mentioned in section 5.1.

6.2 HLPSL Specifications

HLPSL specifications of symmetric authentication protocol have been omitted for lack of space. Our intended target is to model the protocol as close as possible to the protocol description given in section 6.1. The protocol steps could be modeled, except next issue / limitation. The detailed HLPSL model can be found in [18].

Restriction Applied:
During the generation of session keys symmetric authentication protocol make

use of bit-level selection of data [5]. This could not be modeled in HLPSL specifications while all the protocol steps are modeled in HLPSL.

6.3 Automatic Analysis of the Protocol

For security analysis our methodology for experiments 1-10 were the same as we did in section 4.9. In all experiments, the security properties under test are the correspondence properties explained in section 4.6. **For all analyzed scenario, no attacks were found on the protocol in the presence of DY model.**

7 Conclusion

In this paper device authentication protocols mentioned in CWA 14890-1, have been presented and analyzed. We model core security properties as correspondence properties and use the AVISPA tool to automate our security analysis. We have found that device authentication protocols mentioned in CWA 14890-1 are safe w.r.t. given finite number of sessions. Since we have carefully reviewed our formalizations to validate that they faithfully describe the protocols mentioned in CWA 14890-1, and since the tool used is mature enough, we can be confident that in the device authentication protocols there are no design flaws that can lead to attacks on *authentication* and *secrecy*. Of course, vulnerabilities at the cryptographic or implementation level cannot be excluded with this approach. In summary, we found our analysis has provided a greater degree of confidence in the correctness of device authentication protocols mentioned in CWA 14890-1.

Acknowledgments
This work was supported by Master Scholarship grant from Higher Education Commission (HEC), Pakistan and DAAD, Germany. We would like to thank Dieter Gollmann, Sebastien Canard, Jan Meier, Helmut Scherzer and the anonymous reviewers for helpful feedback and suggestions for improvements.

References

1. CEN Workshop Agreement, `ftp://ftp.cenorm.be/PUBLIC/CWAs/e-Europe/eSign/cwa14890-01-2004-Mar.pdf`
2. Signature Creation Smart Cards, `http://www.security-technologynews.com/article/signature-creation-smart-cards.html`
3. ISO/IEC 7816-4 Identification cards, Integrated circuit cards Part 4: Organization, security and commands for interchange (November 2004), `http://www.ttfn.net/techno/smartcards/iso7816_4.html`
4. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J.,Hankes Drielsma, P., Heam, P.-C., Kouchnarenko, O., Mantovani, J., Modersheim,S., von Oheimb, D., Rusinowitch, M., Santos Santiago, J., Turuani, M.,

[5] The first 64 bits of SHA-1(SK1) are used as the first encryption key, the second 64 bits are used for the second encryption key, while the last 32 bits of the hash value are not used. The value of SHA-1(SK2) is used similarly. The first 128 bits are used for both integrity keys and the last 32 bits are not used.

Vigano, L.,Vigneron, L.: The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281-285. Springer, Heidelberg (2005)
5. AVISPA: Automated Validation of Internet Security Protocols and Applications. FET Open Project IST-2001-39252 (2003), http://www.avispa-project.org/
6. A Security Protocol ANimator for AVISPA, http://www.irisa.fr/celtique/genet/span/
7. Deliverable D2.1: The High Level Protocol Specification Language, http://www.avispa-project.org/delivs/2.1/d2-1.pdf
8. Deliverable D2.3: The Intermediate Format, http://www.avispa-project.org/delivs/2.3/d2-3.pdf
9. D. Basin, S. Mödersheim, and L. Vigano. OFMC: A Symbolic Model-Checker for Security Protocols. International Journal of Information Security, 2004
10. M. Turuani. The CL-Atse Protocol Analyser:. F. Pfenning (Ed.): RTA 2006, LNCS 4098, pp. 277–286, 2006.
11. Armando, A., Compagna, L.: SATMC: A SAT-based model checker for security protocols. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 730-733. Springer, Heidelberg (2004)
12. Boichut, Y., Heam, P.-C., Kouchnarenko, O., Oehl, F.: Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols. In: Proc. of Automated Verification of Infinite States Systems (AVIS 2004). ENTCS, pp. 1-11 (2004)
13. D. Dolev and A. Yao. On the Security of Public-Key Protocols. IEEE Transactions on Information Theory, 2(29), 1983
14. M. Rusinowitch, M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In 14th IEEE Computer Security Foundations Workshop, pages 174 – 187. IEEE Computer Society, 2001.
15. W. Diffie and M. E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, IT-22(6):644–654, 1976.
16. G. Lowe. A Hierarchy of Authentication Specifications.In Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW 97), pages 31-43. IEEE Computer Society Press, 1997.
17. L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16(3):872-923, 1994.
18. Javed. A. HLPSL Models of CWA 14890-1 Protocols, http://ashar-javed.blogspot.com/2011/05/formal-analysis-of-cwa-14890-1-protocol.html