PAMoChSA:
The Partial Model Checker Security Analyzer

USER’S MANUAL (Version 1.01)

Fabio Martinelli, Marinella Petrocchi, Anna Vaccarelli
{Fabio.Martinelli, Marinella Petrocchi, Anna Vaccarelli} @iit.cnr.it
Istituto di Informatica e Telematica - Consiglio Nazionale delle Ricerche
Pisa, Italy

May 15, 2002
Contents

1 Overview ................................................. 2
   1.1 Analysis framework .................................. 2
   1.2 Structure of this document ........................... 3

2 The PAMOCHSA input language ........ 4
   2.1 Typed messages ..................................... 4
       2.1.1 Keys’ types .................................. 4
   2.2 The PAMOCHSA inference system ....................... 5
   2.3 Grammar of the input language ....................... 5
       2.3.1 Reserved Key Words ........................... 9
   2.4 Representing security protocols ...................... 10
   2.5 From standard notation to PAMOChSA input language ......... 10
       2.5.1 The structure of an experiment file ............... 11
       2.5.2 An example: the Needham-Schroeder Public Key Protocol ....... 11

3 Using the PAMOChSA tool ............................. 16
   3.1 Invoking PAMOChSA ................................ 16
   3.2 The PAMOChSA graphical user interface .............. 16
       3.2.1 The check buttons ............................. 18
       3.2.2 Combo boxes management ....................... 19
       3.2.3 Syntax errors management ....................... 19
       3.2.4 Message Types Mismatch ....................... 19
       3.2.5 Out of memory unrecoverable error .............. 20
Chapter 1

Overview

In standard communication protocols all the interactions are made by means of communications, i.e. parties communicate with each other by exchanging a sequence of messages on channels of the net. Talking about security communication protocols, cryptographic functions are introduced in the structures of messages, in order to try to guarantee the satisfaction of certain security properties pre-established by the protocol itself. Common security properties analyzed are secrecy (i.e. confidentiality of exchanged messages: nobody, except the legitimate participants, should know the content of an exchanged secret message), message authenticity (i.e. no alteration of the content of a message) and entity authentication (i.e. capability of identifying other parties during a communication). Given the sensitive nature of information possibly exchanged in a run of a protocol (e.g. credit card numbers, passwords, current account numbers) it appears reasonable to consider the presence in the net of third malicious parties: such intruders try to interfere with the normal execution of a protocol, in order to achieve advantages for themselves. Due to the unpredictable behavior of such intruders, the design of distributed security protocols is very challenging. We may have some subtle attacks due to how messages are exchanged over the network.

The PAMoCHSA tool analyzes security protocols with bounded number of sessions and it is able to discover, at conceptual level, attacks against security procedures. The underlying theory has been proposed in [4, 8] and extends the partial model checking techniques of [1, 2] to a process algebra with cryptographic primitives.

PAMoCHSA is a prototype tool written in the functional language ocaml\(^1\) with a graphical interface based on the package mlgtk\(^2\). The current implementation runs over a Pentium III machine with Linux RedHat 6.2 operating system.

1.1 Analysis framework

For our analysis we consider honest participants \(S\) of a protocol plus an intruder \(X\), as a compound system \(S|X\), i.e. a system whose components run in parallel and can interact. Basically, our aim consists in verifying that a certain security property will be satisfied by the compound system. We wonder if, for every possible active intruder \(X\), \(S|X\) satisfies that property.

\(^1\)http://pauillac.inria.fr/ocaml/
\(^2\)http://cristal.inria.fr/ cuoq/mlgtk.html
To describe a protocol, we developed an operational language, actually CCS process algebra by Robin Milner ([9]), with slight variations for the treatment of cryptographic functions ([8, 5]). We do not need a priori any intruder specifications and moreover we make no assumption on its behavior, $X$ can be any term of the algebra. As the other protocol participants, it has a set of message manipulating rules, that are used to model cryptographic functions such as encryption and decryption. The intruder has the capability to send and receive messages over net channels to and from the other components of the system, it can also intercept and fake messages. It can build new messages by starting from its initial knowledge, i.e. the set of messages we can freely assume the intruder knows at the beginning of the computation, and other intercepted information obtained during the run of the protocol. The intruder does not implement cryptanalysis: data encryption is assumed to be “opaque”, i.e. a message encrypted by the public key of $i$ cannot be decrypted by anyone but the person who knows the corresponding private key, unless the decryption key is compromised. An adversary can intercept and store an encrypted message, and replay it later, but the structure of the message is not accessible to it, i.e., it cannot split the encrypted message, unless it knows the decryption key. The intruder knowledge grows as the computation goes on: we consider if that knowledge satisfies, at a certain point of the computation, a predicate based on the specific property. In the case of a positive answer, our analysis reports an attack against the security property. In other words, an attack is possible when $X$ is able to force the execution of protocol to be performed in a different way, by cheating the honest parties. PAMoChSA, the Partial Model Checking Security Analyzer, [8, 4, 6, 7].

1.2 Structure of this document

This document is organized as follows: in chapter 1 we give some notions about the messages exchanged on the net between two or more participants, we give the syntax of the input language to build the experiment file needed as input by PAMoChSA, we show how to translate the standard notation in which security protocols are usually specified in literature in our input language. Chapter 2 is dedicated to practically describe how to use the tool and its Graphical User Interface (GUI). The appendix briefly describes the underlying theory that has lead to the development of the PAMoChSA tool.
Chapter 2

The PAMoChSA input language

This chapter describes the syntax of the PAMoChSA input language and gives examples for the drawing up of an PAMoChSA input experiment file.

2.1 Typed messages

The messages we take into consideration are typed messages, i.e., we label each message with a type that denotes its structure. Types are used to record the structure and kind of exchanged data. Since certain operations are meaningful only over data with a certain structure, types permit to define managing rules that precisely correspond to those operations. Messages will be the data manipulated by the protocol agents and typed messages are messages whose structure is explicitly represented.

A message $m$ of type $T$ is represented as: $m : T$. this expression forms a typed message. The pure message $m$ and its type $T$ are separated by colon and the type is written in capitals. Typed messages can be basic or compound, through a combination of pairs and encryptions (the user can refer to the following section “Grammar of the input language” for the recursive definition of the typed messages).

2.1.1 Keys’ types

The user may assign to each message the type she likes, i.e., she can give to the each name of the participants through a protocol the type “Name”, to a randomly generated message the type “Nonce”, and so on.

There are however two special types indicating the types for the encryption and decryption keys. At the state of the art, the PAMoChSA tool is able to simulate asymmetric cryptography operations, involving pairs of public/private keys. The type for each public key is $EKey$. The type for each private key is $DKey$. At syntax level, the correspondence between a private and a public key is denoted by means of the keys’ name, the same for each pair of correspondent private and public key. Suppose party A and party B are communicating each other by sending and receiving messages. Both party A and party B hold a pair of public/private keys: for the tool input language, the name of each key belonging to the A’s pair has to be the same, say $key_A$. With regard to the representation of the typed messages, it will be:
key_A : EKey to indicate the public key;
key_A : DKey to indicate the private key.

The same syntax holds for B’s keys.

According to the basilar schemes of asymmetric cryptography, party A digitally signs something with her private key, syntactically “Enc(m2 : Msg, key_A : DKey)”, and she encrypts something else, intended for party B, with B’s public key, “Enc(m2 : Msg, key_B : EKey)”. Party B, once received something likely signed by A, verifies the signature by applying to the signed message A’s public key, key_A : EKey. Again, party B tries to decrypt a received encrypted message, by applying to it his private key.

Note 2.1.1 Symmetric keys can be simulated by using pairs of public and private keys.

2.2 The PAMoChSA inference system

Agents are able to obtain new messages from the set of messages produced or received through an inference system1. This system consists in a set of inference schemata. An inference schema can be written as:

\[ IS = \frac{m_1 : T_1, \ldots, m_n : T_n}{m_0 : T_0} \]

where \( m_i : T_i \) are typed messages, \( m_1 : T_1, \ldots, m_n : T_n \) are a set of premises, while \( m_0 : T_0 \) is the conclusion. An inference schema is read from top to bottom, in this way: “if messages \( m_1 : T_1, \ldots, m_n : T_n \) belong to the agent knowledge, then message \( m_0 : T_0 \) belongs to the agent knowledge”.

The PAMoChSA inference system is shown in Fig. 2.2. Rule (1) builds the pair of two messages; rules (2) and (3) are used to obtain the components of a pair; rule (4) and (5) allow messages to be encrypted using a public key of type EKey or a private key of type DKey; rules (6) and (7) allow messages to be decrypted using the corresponding inverse keys.

Each \( T_i \) in Fig. 2.2 may be basic (e.g. Name, Nonce, EKey, DKey, etc) or compound (e.g. \( \text{Name} \times \text{Nonce} \), indicating the pair of a name and a nonce, \( \text{Enc}((\text{Name} \times \text{Nonce}) \times \text{EKey}) \), indicating the encryption of the pair composed by a name and a nonce with a public key), and so on.

We use commas to separate messages in pairs, and the * symbol to separate the types of the pairs.

2.3 Grammar of the input language

In this section we give the grammar of the PAMoChSA input language, which will be used to write the input experiment file (see section 2.5). The grammar is recursively specified as follows:

---

1The theory [5] is parametric w.r.t. the inference system, while the tool currently implements a specific one.
\[
\begin{align*}
\frac{x : T, y : T}{x, y : T \cdot T} \quad (1) \\
\frac{x : T \cdot T}{x : T} \quad (2) \\
\frac{x, y : T \cdot T}{y : T} \quad (3) \\
\frac{\text{Encrypt}(x, y) : \text{Enc}(T \cdot E\text{Key})}{x : T \cdot T \cdot T \cdot T} & \quad (4) \\
\frac{\text{Encrypt}(x, y) : \text{Enc}(T \cdot D\text{Key})}{x : T \cdot T \cdot T \cdot T} & \quad (5) \\
\frac{\text{Encrypt}(x, y) : \text{Enc}(T \cdot E\text{Key})}{x : T \cdot T \cdot T \cdot T} & \quad (6) \\
\frac{\text{Encrypt}(x, y) : \text{Enc}(T \cdot D\text{Key})}{x : T \cdot T \cdot T \cdot T} & \quad (7)
\end{align*}
\]

Figure 2.1: The PAMoChSA inference system

\[
\begin{align*}
\text{msg} & ::= \text{msgx} \text{ eof;} \\
\text{msgx} & ::= \text{pstr} \\
& \quad | \quad (\text{msgx}) \\
& \quad | \quad \text{msgx} \cdot \text{msgx} \\
& \quad | \quad \text{Enc} \{\text{msgx}\} \text{msgx} \\
& \quad | \quad ; \\
\text{msg\_type} & ::= \text{msg\_typex} \text{ eof;} \\
\text{msg\_typex} & ::= \text{id} \\
& \quad | \quad \text{E\text{Key}} \\
& \quad | \quad \text{D\text{Key}} \\
& \quad | \quad (\text{msg\_typex}) \\
& \quad | \quad \text{msg\_typex} \cdot \text{msg\_typex} \\
& \quad | \quad \text{Enc}(\text{msg\_typex} \cdot \text{msg\_typex}) \\
& \quad | \quad ; \\
\text{typed\_msg} & ::= \text{typed\_msgx} \text{ eof;} \\
\text{typed\_msgx} & ::= \text{msgx} : \text{msg\_typex};
\end{align*}
\]
\[\text{expr} ::= \text{expr} \text{ eof};\]

\[\text{expr} ::= \text{typed}_{\text{msg}}\text{expr}\]
\[\quad \text{ident}\]
\[\quad \text{Fst} \text{expr}\]
\[\quad \text{Snd} \text{expr}\]
\[\quad (\text{expr})\]
\[\quad (\text{expr}, \text{expr})\]
\[\quad \text{Encrypt}(\text{expr}, \text{expr})\]
\[\quad \text{Decrypt}(\text{expr}, \text{expr})\]

\[\text{term} ::= \text{term} \text{ eof};\]

\[\text{term} ::= 0\]
\[\quad \text{Send}(\text{pstr}, \text{expr})\text{term}\]
\[\quad \text{Recv}(\text{pstr}, \text{ident} : \text{msg}\text{type})\text{term}\]
\[\quad \text{If}(\text{expr} = \text{expr})\text{term}\text{Else}\text{term}\text{End If}\]
\[\quad \text{If}(\text{expr} = \text{expr})\text{term}\text{End If}\]
\[\quad \text{If Deduce}(\text{ident} = \text{expr})\text{term}\text{Else}\text{term}\text{End Deduce}\]
\[\quad \text{If Deduce}(\text{ident} = \text{expr})\text{term}\text{End Deduce}\]
\[\quad \text{Choice} \text{c_list}\text{End Choice}\]
\[\quad \text{Parallel} \text{p_list}\text{End Parallel}\]

\[\text{c_list} ::= \text{term}\]
\[\quad \text{term}\text{Or}\text{c_list}\]

\[\text{m_list} ::= \text{typed}_{\text{msg}}\text{expr}\]
\[\quad \text{typed}_{\text{msg}}\text{expr} ; \text{m_list}\]

\[\text{str_list} ::= \text{str}\]
\[\quad \text{str} ; \text{str_list}\]
\[p\text{list} \ ::= \ termx \]
\[ | \ termx \text{ And } p\text{list} \]
\[ ; \]

\[p\text{str} \ ::= \ str \]
\[ | \ p\text{str} \]
\[ ; \]

\[form \ ::= \ formx \text{ eof};\]

\[formx \ ::= \ typed\_msgx \]
\[ | \ formx \text{ & formx} \]
\[ | \ formx \text{ | formx} \]
\[ | \text{Not } formx \]
\[ | \ ( formx ) \]
\[ ; \]

\[formulas \ ::= \ formulasx \text{ eof};\]

\[formulasx \ ::= \text{ FORM form } /\text{FORM} \]
\[ ; \]

\[ experiment\_form \ ::= \ text{ experiment\_formx } \text{ eof};\]

\[ experiment\_formx \ ::= \text{ < FORMULA > form < /FORMULA >}\]
\[ \text{ < KNOWLEDGE > m\text{list} < /KNOWLEDGE >}\]
\[ \text{ < HIDE\_CHANNELS > str\text{list} < /HIDE\_CHANNELS >}\]
\[ \text{ < SPEC > termx < /SPEC > }\]
\[ ; \]

We explain how to read the grammar.

As regard the recursive definition of msgx, notation \(Enc [p\text{str}](msgx)\) is used to set the formula to be checked, when it is asked if an encrypted message belongs to the intruder’s knowledge.

Suppose the formula is a typed message consisting in the pair (name \(name\), nonce) encrypted with the public key \(pkey\). The formula will be:

\[<\text{FORMULA}>\]
\[Enc[p\text{key}](name, nonce) : Enc((Name * Nonce) * EKey)\]
\[</\text{FORMULA}>\]

\(str\) and \(p\text{str}\) are alphanumeric strings plus the special characters ‘.’ ‘’’ ‘’’. They can begin only with a small letter. \text{ident} is an alphanumeric string, plus the special characters ‘.’ ‘’’ ‘’’, whose first character is a capital letter.
The field `experiment_formx` represents exactly the structure of the experiment file described in Section 2.5.

`form` can be either a single typed message or a combination of typed messages by means of the logical operators & (AND), | (OR) and Not.

Between the `KNOWLEDGE` and `/KNOWLEDGE` identifiers there is a list of typed messages separated by semicolons.

Between the `HIDE_CHANNELS` and `/HIDE_CHANNELS` identifiers there is a list of strings, separated by semicolons, indicating channels’ names.

Between the `SPEC` and `/SPEC` identifiers there is the body of the protocol. Let us analyze the recursive definition of `termx`:

- 0, the process that does nothing;

- `Send (pstr, exprx).termx` is the process that can perform the sending of message `exprx` on channel `pstr` and then behave like `termx`;

- `Recv (pstr, ident : msgTypex). termx` is the process that can perform the reception of a message identified as `ident`, of type `msgTypex`, on channel `pstr` and then behave like `termx`;

- `If (exprx = exprx) Then termx Else termx End If` is the match construct. If the two expressions `exprx` are equal to each other, then the process behave as the first `termx`, else as the second;

- `If Deduce (ident = exprx) Then termx Else termx End Deduce` is the inference construct. It is typically used to perform decryption actions and to retrieve the first and the second element in a pair.

- `Choice c_list End Choice` is the process that non-deterministically decides to behave as one of the terms in `c_list`.

- `Parallel p_list End Parallel` is the process representing the parallel composition of all the processes in `p_list`. The parallel composition of two processes is the process that can perform an action if one of the sub-components performs the action, and a synchronization action if the sub-components perform complementary actions, i.e. Send-Recv actions.

### 2.3.1 Reserved Key Words

The following list is the list of the reserved key words of the PAMoChSA input language.
Security protocols are usually specified in literature in the standard notation described in the following.

Consider a set of agents able to send and receive messages. Basically, the sending and reception of a message is represented in this way:

\[ c_i \ A \rightarrow B : \text{msg} \]

where \( \text{msg} \) is the exchanged message, \( c_i \) is the \( i \)-th communication channel, on which the exchange takes place. \( A \) and \( B \) are the sender and the receiver of \( \text{msg} \).

Cryptographic keys and functions can be represented as follows:

\[ PK(I), PK(I)^{-1} := \text{respectively, public and private key of party } i \]
\[ \{\ldots\}PK(I)^{-1} := \text{message signed by party } i \]
\[ \{\ldots\}PK(I) := \text{message encrypted by public key of party } i \]
\[ \{\ldots\}_KEY := \text{message encrypted by symmetric key } KEY \]

The presence of a malicious agent \( X \), that can intercept and fake messages, can be denoted as:

(1) \( X(A) \rightarrow B : \text{msg} \)
(2) \( A \rightarrow X(B) : \text{msg} \)

Notation (1) describes intruder \( X \) that sends a message \( \text{msg} \) to party \( B \) pretending to be party \( A \) (forgery); (2) denotes: \( \text{msg} \), originally intended for \( B \), is actually intercepted by \( X \) (interception).

2.5 From standard notation to PAMoChSA input language

The PAMoChSA tool needs the protocol specifications to be written in an input language different from the standard notation. In the following, we show how to map the standard notation in the tool input language.
2.5.1 The structure of an experiment file

An experiment file contains a logical formula to be tested, a set of messages which belong to the intruder initial intruder knowledge, a set of communication channels and the protocol specifications. They are contained in special identifiers:

```
< FORMULA >
  Formula
< /FORMULA >

< KNOWLEDGE >
  Intruder Initial Knowledge
< /KNOWLEDGE >

< HIDE_CHANNELS >
  Hidden Channels List
< /HIDE_CHANNELS >

< SPEC >
  Protocol Specifications
< /SPEC >
```

Each formula can be either a single typed message or it can logically consist of typed messages tied by the logic operators and, or, not.

The initial knowledge is the set of typed messages known by the intruder before a run of the protocol.

The hidden channels list is a list of communication channels where the intruder can not interfere during a run of the protocol.

The protocol specifications is actually the body of the protocol, the sequence of sending, reception and control actions each party has to act to participate through the protocol. Next subsection it is shown the user how to translate a protocol from the literature standard notation to the PAMoChSA input language. The Needham-Schroeder Public Key Protocol is taken as an example to show the translation.

2.5.2 An example: the Needham-Schroeder Public Key Protocol

This protocol has became paradigmatic for testing analysis tools for cryptographic protocols. It has a subtle flaw discovered by Lowe [3] which arises in the presence of a malicious agent.

In Tab. 2.1, we show the intended execution of the protocol by using the notation which is commonly found in literature. In the flawed version, the sender A communicates to B a fresh nonce $N_a$ (i.e. a randomly guessed value) and its name encrypted with the public key of B (thus only B can decrypt this message). Next, the receiver B communicates to A the just received nonce $N_a$ and a fresh nonce $N_b$, both encrypted with the public key of A. Finally, the sender A communicates to the receiver the nonce $N_b$ encrypted with the public key of B. At the end of a successful run between a sender A and a receiver B, only these two processes should know $N_a$ and $N_b$.

$^2$These nonces could be used to establish a new communication channel with a new shared key that is a function
Table 2.1: Needham Schroeder Public Key protocol.

The PAMoCHSA input language is more expressive than the standard sequence of messages used to describe protocols. In the standard notation many controls are not explicitly declared. Consider the second line in both the flawed and the corrected version Table 2.1: in order to retrieve the nonce $N_a$, as a clear text, party B has to have previously decrypted what he received in the first message. To retrieve $N_a$, it is supposed that party B effectively owns the inverse key corresponding to $PK(B)$.

The experiment file corresponding to the specifications of the flawed version of the NSPK protocol is reported below.

In writing the input file, we consider three agents participating through the protocol: $A$, $B$ and $X$. The agent $A$ may act as initiator both with $B$ and $X$; the agent $B$ acts as responder, while the agent $X$ may act as initiator or responder of the protocol. The specification for $X$ is not given, thus we check whether the system is secure against whatever behavior the agent $X$ could have. We only specified the intruder’s initial knowledge, i.e. the public keys of $A$ and $B$, the names of $A$ and $B$, its name and its private and public key. We need not give the nonces to the intruder because it can guess them by itself.

\[
(\text{ba} : \text{Special} \& \text{ax} : \text{Special})
\]

\[
\text{<FORMULA>}
(\text{ba} : \text{Special} \& \text{ax} : \text{Special})
\text{<KNOWLEDGE>}
\text{keya} : \text{EKey}; \text{keyb} : \text{EKey}; \text{keyx} : \text{EKey}; \text{keyx} : \text{DKey}; \text{agent}_\text{x} : \text{Name}; \text{agent}_\text{a} : \text{Name}; \text{agent}_\text{b} : \text{Name}
\text{<KNOWLEDGE>}
\text{<HIDE_CHANNELS>}
\text{no_hide}
\text{</HIDE_CHANNELS>}
\]

By setting the formula in such a way, we actually ask if the intruder $X$ is able, to a certain point of the computation, to know two special typed messages, $\text{ba} : \text{Special}$ and $\text{ax} : \text{Special}$, inserted in the specifications of the protocol just as control messages (the sending and reception of these values in the original version of the protocol is not planned).
Party A, wishing to start the protocol with party X, first sends on the special channel \textit{up} the value \textit{ab}, to say “I am A and I am going to start talking with X”.

The correct execution of the protocol should expect that when X just finishes a run of the protocol, apparently with A, he sends the special value \textit{xa} : \textit{Special}, to say “I am X and I have just finished talking with A”.

To send the special value \textit{ba} : \textit{Special} when previously the value \textit{ax} : \textit{Special} has been sent means that something wrong has occurred during the run of the protocol, because B is persuaded to have spoken with A rather than X, while A has actually started a run with X.

Note that the special control messages are sent on public channels as cleartexts. As soon as they are sent, the intruder can act its capability to eavesdrop on the net and the control values are added to its knowledge.

\textless SPEC\textgreater

\texttt{Parallel}

(* Initiator specifications (A) *)

\texttt{Choice}

\texttt{Send(\textit{up, ab} : \textit{Special}).}
\texttt{Send(c1ab, Encrypt((nab:Nonce, agent_a: Name), keyb:EKey)).}
\texttt{(* nab: A’s nonce for B; keyb: B’s public key *)}
\texttt{Recv(c2ab, \textit{XA} : \textit{Enc}((\textit{Nonce*Nonce})*EKey)).}
\texttt{If Deduce (YA = Decrypt(XA, keya:DKey)) Then}
\texttt{(* keya: A’s private key *)}
\texttt{If Deduce (NA = Fst(YA)) Then}
\texttt{If (NA = nab : Nonce) Then}
\texttt{If Deduce (NA1 = Snd(YA)) Then}
\texttt{Send(c3ab,Encrypt(NA1, keyb : EKey)).0}
\texttt{End Deduce}
\texttt{End If}
\texttt{End Deduce}

Or

\texttt{Send(\textit{up, ax} : \textit{Special}).}
\texttt{Send(c1ax, Encrypt((nax:Nonce, agent_a:Name), keyx:EKey)).}
\texttt{(* nax: A’s nonce for X; keyx: X’s public key *)}
\texttt{Recv(c2ax, \textit{XA} : \textit{Enc}((\textit{Nonce*Nonce})*EKey)).}
\texttt{If Deduce (YA = Decrypt(XA, keya:DKey)) Then}
\texttt{If Deduce (NA = Fst(YA)) Then}
\texttt{If (NA = nax : Nonce) Then}
\texttt{If Deduce (NA1 = Snd(YA)) Then}
\texttt{Send(c3ax,Encrypt(NA1, keyx:EKey)).0}
End Deduce
End If
End Deduce
End Deduce

End Choice

And

(* Responder specifications (B) *)

Choice

Recv(c1xb, Z : Enc((Nonce*Name)*EKey)).
If Deduce (X = Decrypt(Z, keyb : DKey)) Then
 (* keyb: B’s private key *)
 If Deduce (A = Snd(X)) Then
  If Deduce (Na = Fst(X)) Then
   Send(c2xb, Encrypt((Na,nbx : Nonce), keyx : EKey)).
   (* nbx: B’s nonce for X *)
 Recv(c3xb, V:Enc(Nonce*EKey)).
   If Deduce (Vb = Decrypt(V, keyb : DKey)) Then
    If (Vb = nbx : Nonce) Then
     Send(up, bx : Special).
    End If
   End Deduce
  End Deduce
End Deduce
End Deduce

Or

Recv(c1ab, Z : Enc((Nonce*Name)*EKey)).
If Deduce (X = Decrypt(Z, keyb : DKey)) Then
 If Deduce (A = Snd(X)) Then
  If Deduce (Na = Fst(X)) Then
   Send(c2ab, Encrypt((Na,nba : Nonce), keya : EKey)).
   (* nba: B’s nonce for A *)
 Recv(c3ab, V:Enc(Nonce*EKey)).
   If Deduce (Vb = Decrypt(V, keyb : DKey)) Then
    If (Vb = nba : Nonce) Then
     Send(up, ba : Special).
    End If
   End Deduce
  End Deduce
End Deduce
End Deduce

14
Table 2.2: The Lowe’s attack described in common notation.

\[
\begin{align*}
1) & \quad c1 \quad A \leftrightarrow X & : & \{N_a, A\}_PK(X) \\
2) & \quad c1 \quad X(A) \leftrightarrow B & : & \{N_a, A\}_PK(B) \\
2) & \quad c2 \quad B \leftrightarrow X(A) & : & \{N_a, N_b\}_PK(A) \\
1) & \quad c2 \quad X \leftrightarrow A & : & \{N_a, N_b\}_PK(A) \\
1) & \quad c3 \quad A \leftrightarrow X & : & \{N_b\}_PK(X) \\
2) & \quad c3 \quad X(A) \leftrightarrow B & : & \{N_b\}_PK(A)
\end{align*}
\]

Tags (* and *) contain comments.

We performed our analysis and, as expected, we found the flaw in [3]: an intruder \(X\) masks as \(A\) to \(B\) and discovers both nonces \(nab\) and \(nba\).

For completeness, the Lowe’s attack is reported in Tab. 2.2 in the literature standard notation.

The attack consists of two concurrent sessions: in the first one, the agent \(A\) initiates the protocol with \(X\); in the second one, the agent \(X\) communicates with \(B\) pretending to be \(A\). The steps of the attack can be summarized as follows: the agent \(A\) starts a run of the protocol with the agent \(X\); then the agent \(X\) can simulate \(A\) in a run of the protocol with the agent \(B\). The agent \(B\) sends to \(X(A)\) the message \(\{N_a, N_b\}_PK(A)\), which contains the fresh nonce \(N_b\), encrypted with the public key of \(A\). Now, the intruder cannot directly decrypt the message, but can send the message to the agent \(A\). The agent \(A\) correctly decrypts \(\{N_a, N_b\}_PK(A)\) and resends the nonce \(N_b\) to \(X\), encrypted with the public key of \(X\), since it thinks it is the second message of its run with \(X\). Eventually, \(X\) discovers \(N_b\) and sends it to \(B\).

The corrected version of the protocol as given in [3] (see Tab. 2.1) has not this flaw. Indeed, the second message encodes also the name of the sender, i.e. \(B\). Thus, the presented attack is no more possible, because \(A\) would receive from \(X\) a message whose sender is \(B\). At that point, \(A\) should quit the session with \(X\). The tool can be used to check that the previous attack is no longer possible.
Chapter 3

Using the PAMoCHSA tool

This section explains how to use the PAMoCHSA tool. The general procedure for using the tool is for the user to first create an experiment file in the Crypto-CCS operational language, (see Chapter 2 for details on the syntax of the language), describing the behaviour of a concurrent system.

The file is loaded into the PAMoCHSA tool. Once the file is loaded the user may invoke the graphical user interface to evaluate the behaviour of the system under investigation.

3.1 Invoking PAMoCHSA

The tool is invoked with the command:

```
./pamochsa
```

from the directory in which the PAMoCHSA executable file has been previously installed. The graphical user interface will be invoked. The PAMoCHSA tool, by means of its graphical interface, loads a specific security protocol, analyzes a run of the protocol and displays the elaboration results.

3.2 The PAMoCHSA graphical user interface

The PAMoCHSA graphical user interface is shown in Fig. 3.2. In the following, we describe the sequence of operations which lead to the GUI as in the figure.

1. Click once in the File menu. Select the first item of the menu, Load experiment. The Select a file to load an experiment window will appear. Use this browser to select the .exp file to be analyzed.

2. Select the .exp file, then press the ok button in the browser. The browser will disappear.

3. Check the correctness of the entry in the Combo Box Current Spec Name. The name of the current loaded .exp file should appear.
1. File menu
2. Show menu
3. About menu
4. Current Spec Name combo box
5. Elaboration & Result window
6. Intruder Knowledge combo box
7. Formula combo box
8. Hide Channels combo box
9. Hide Channels check button
10. Random Generation check button
11. Store States check button
12. Update button
13. Elapsed Time entry box
14. Run button
15. Significant Stored States entry box
16. Stop button
17. Elaboration & Result window scrollbars

Figure 3.1: The PAMOChSA GUI after loading an experiment file and running an elaboration
4. Check the correctness of the entry in the Combo Box *Intruder Knowledge*. This entry contains the list of the typed messages which belong to the intruder initial knowledge, as specified in the `.exp` file. Messages are separated each other by semicolons. A `$ : $` symbol appears in the box in case of unsuccessful loading.

5. Check the correctness of the entry in the Combo Box *Formula*. This entry contains the current formula to be checked, as specified in the `.exp` file. A ( `$ : $` ) symbol appears in the box in case of unsuccessful loading.

6. Check the correctness of the entry in the Combo Box *Hide Channels*. This entry contains the list of the channels to be hidden during a run of the protocol under investigation. PAMoCHSA offers the user the possibility to declare over which communication channels an adversary is *a priori* not able to interfere. The list of these channels is originally specified by the user in the `.exp` file. When no element is present in the list, the entry shows the words “no hide”. Nothing appears in the box in case of unsuccessful loading.

7. Click once on the run button to start the elaboration. The system will execute a run of the protocol, making the adversary interfere with the normal behaviour of the honest agents participating through the protocol. Essentially, the model checker is now working. The elaboration might require a while depending on the protocol complexity under examination. When the computation stops, either PAMoCHSA gives the trace of a possible attack against the security property under investigation\(^1\) or a message about the fulfillment of the property. An attack is possible when an adversary is able to force the execution of the protocol to be performed in a different way, by cheating the honest parties.

8. During the elaboration, the user can monitor the work in progress by means of the *Update* button. To click on that button causes the elapsed CPU time (sec) and the visited states to appear in the entries *Elapsed Time* and *Significant Stored States*.

Each formula can be either a single typed message or it can logically consist of typed messages tied by the logic operators *and*, *or*, *not*. Substantially, given the three inputs present in every `.exp` file: (i) the protocol specifications; (ii) the intruder initial knowledge; (iii) the formula on the intruder knowledge to be checked, the elaboration in step 7 analyzes if the intruder knowledge satisfies, at a certain point of the computation, the formula encoding a secret initially known only by the honest participants.

The user can stop the elaboration before its natural conclusion clicking on the *stop* button.

### 3.2.1 The check buttons

Once the experiment file has been loaded, the user can select specific features characterizing the subsequent elaboration. To this purpose the GUI is supplied with three check buttons:

\(^1\)See ?? for an explanation about the link between the security property to be analyzed and the formula to be checked.


- **Hide Channels**: to deselect this option allows the intruder to enter each communication channel defined in the protocol specifications (i.e., to do not care about the original hide channels list);

- **Random Generation**: to select this option allows the intruder to generate a particular kind of randomly generated messages during the run of the protocol;

- **Store States**: to select this option allows the storage of the actual state of investigation (i.e., the record of the number of the states visited from the beginning of the computation).

### 3.2.2 Combo boxes management

The entries in the *Intruder Knowledge*, *Formula* and *Hide Channels* combo boxes can be changed on the fly. Consider the possibility to change the formula to be checked. (i) Click in the entry of the *Formula* combo box bringing it in focus and hit return in order to save the current formula. (ii) Select the current formula and delete it. (iii) Write the new formula according to the Crypto-CCS syntax (see Chapter 2, Section ??). (iv) Hit return to effectively load the current new formula. To verify the effective loading of the new formula click once in the *Show* menu and select the last item, *Print formula*. The current loaded formula will appear in the *Elaboration & Results* window.

### 3.2.3 Syntax errors management

This subsection explains how to manage some syntax errors possibly present in a .exp file. When the user tries to load an experiment file containing some tokens not recognized by the PAMoChSA parser, the error is highlighted in the main window:

```
Error at: enspkaut.exp line: 19 char 8
```

Such a message tells the user that in the currently loaded .exp file, called enspkaut.exp, at char 8 of line 19 there is a syntax error, such as a not recognized token, an omission of a bracket, etc.

### 3.2.4 Message Types Mismatch

Once a .exp file has been loaded, a lack of consistency may happen not only for syntax errors in that file. Before the elaboration starts, be sure that the same type of messages transit over the same communication channels. Click once in the *Show* menu and select the second item, *Print types*. The types of the messages in transit over the channels will appear in the *Elaboration & Results* window. Let us give a simple example: suppose you have just loaded a protocol specifications where on the communication channel called c1 the encryption of a name with a public key must transit. If the selection of the item *Print types* in the *Show* menu leads to the following:

Types of enclosed channels:
- c1 - Enc(Name*EKey)
- c1 - Enc(Nonce*EKey)
- c2 - ...
- c3 - ...
- c4 - ...

19
this means that you made a mistake in the specifications, writing different types of the message on channel c1 on the sender side and on the receiver side. To run an elaboration with this type mismatch leads either to a false result (you have not analyzed the intended protocol) or to an application crash with an error message in the Linux shell. In practice, only one type for channel has to appear in the *Types of enclosed channels* list.

### 3.2.5 Out of memory unrecoverable error

The PAMOCHESA tool, similarly to other tools based on model checking techniques, suffers the problem of the explosion of the states. Analyzing a protocol with many participants running in parallel and/or with a great number of exchanged messages \(^2\) may lead to a non termination of the elaboration and, as a consequence, to an application crash. The user can consider to voluntarily stop the elaboration by means of an opportune monitoring of its memory resources.

---

\(^2\)Anyway, the tool has been realized for the analysis of protocols with finite behaviour and with a finite number of participants.
Bibliography


