CryptoLog: A Theorem Prover for Cryptographic Protocols

Bart De Decker and Frank Piessens*

K.U.Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

e-mail: `Bart.DeDecker@cs.kuleuven.ac.be` `Frank.Piessens@cs.kuleuven.ac.be`

Abstract. Logics for cryptographic protocols have proved very useful in validating the correctness of these protocols. However, neither the specification, nor the logic proof help the implementor of the protocol. Although the proof states the assumptions explicitly, it does not —in general— contain guidelines for the implementation. In this paper, we present a theorem prover whose input is very close to a program written in a procedural language. Moreover, explicit guidelines for implementors may be given.

1 Introduction

Logics for cryptographic protocols have proved to be of invaluable help in validating and correcting these protocols (see also [2]):

- these logics help to formalize the reasoning about useful abstract properties of these protocols;
- the logics require to write down the initial security assumptions of the protocol that might otherwise be buried in the implementation details;
- the logics achieve reasonably well-defined goals; the analysis process ties the evolution of beliefs to message contents and number of messages; this has certainly many advantages:
  - it helps to determine the minimum number of messages to achieve a certain set of beliefs;
  - useless or erroneous message exchanges can be spotted easily;
  - subtle differences between similar protocols come to the surface.

Despite their many merits, they also have their limitations:

- most logics are based on assumptions that are not written explicitly; for instance, many assume that the concatenation of expressions is commutative and that the sender can recognize its own messages; moreover, they assume that the integrity of encrypted (or signed) messages is always preserved (i.e. they withstand message-splicing attacks),

* Postdoctoral Fellow of the Belgian National Fund for Scientific Research (F.W.O.)

- many require an 'idealization' process, or at least the addition of extra annotations;
  this 'process', which need to be done prior to the analysis, is not fully automated, and therefore error-prone;
- often, the analyser uses a cumbersome syntax, which makes it harder to non-experts to understand the 'subtleties' of the protocol.

These limitations make it difficult for a programmer to implement the protocol correctly and interpret the assumptions in the right way. Many assumptions do influence the implementation. In this paper a new syntax is shown which offers extensive guidelines to the implementors of these protocols.

The rest of this paper is organized as follows. In the next section, we briefly introduce the notations used in this paper. Section 3 illustrates some pitfalls of the GNY and BAN logic. In section 4, we introduce the action-based syntax of CryptoLog, a new theorem prover for cryptographic protocols. Section 5 describes the tasks performed by the prover. Section 6 shows typical input and output for the analyser. In section 7, we discuss some future work. Finally, section 8 summarizes the conclusions.

2 Notations

The notations used in this paper are similar to those of the BAN- and GNY-papers [1, 3]. Table 1 summarizes briefly the most important formulae. However,

Table 1. Symbols used in this paper.

Formula	Explanation
N_a	a nonce generated by A
T_a	a timestamp generated by A
X, Y	Concatenation of two formulae X and Y
$\{X\}_K$	Conventional encryption of X with key K
$P \mid\equiv X$	P believes X
$P \longrightarrow Q$: X	P sends X to Q
$S \mid\Rightarrow X$	P has jurisdiction over X
$P \overset{K}{\longleftrightarrow} Q$	K is a suitable key to be shared between P and Q
$P \ni X$	P holds (or can calculate) X
$P \mid\sim X$	P once conveyed (sent) X
$P \lhd X$	P is told (receives) X
$P \lhd *X$	P is told X which he did not convey previously in the current run
$\Phi(X)$	X is recognizable
$-K, +K$	private and public part of a public key pair

3 Some Pitfalls of Logics

The use of logics for cryptographic protocols can be hampered in practical applications. In this section, we will show two examples where the 'use' of the logics is not optimal:

- in the enhanced Needham-Schroeder protocol (as described in [3]), a *hand-made* proof is given which contains, unfortunately, a small error, due to a misinterpretation of the concept of 'a good key'. Although the proof itself is neither difficult, nor very long, it is for humans very difficult to construct such a proof one deduction step at a time. Hence, a fully automated theorem prover is really necessary.
- in the modified wide-mouthed-frog protocol, the idealization step removed redundant fields. The order of these fields, however, is important in a real implementation.

3.1 Enhanced Needham-Schroeder Protocol

Table 2 shows the enhanced Needham-Schroeder Protocol as it has been published in [3]. The purpose of the modifications (which are underlined) is

- to make Q believe that K is a suitable key for P and Q, and
- to make P believe that Q believes that K is a suitable key.

In the original proof, the authors derive from the 6^{th} message:

$$P \models Q \models P \stackrel{K}{\longleftrightarrow} Q$$

The authors come to that conclusion by applying the following deduction rule:

$$\frac{P \lhd *\{X\}_K \ , P \ni K, \ P \models P \stackrel{K}{\longleftrightarrow} Q \ , P \models \Phi(X) \ , \\ P \models \#(X,K)}{P \models Q \mid\sim X, \ P \models Q \mid\sim \{X\}_K \ , P \models Q \ni K}$$

However, this rule can not be applied. P does not believe that (N_q,Q,K) is fresh! It could be a replay of a previous run! In the initial assumptions, S believes that K is a suitable key for P and Q ($S \models P \stackrel{K}{\longleftrightarrow} Q$). However, nowhere is mentioned that K should be freshly generated by S. Hence, a final modification is necessary (as shown in table 3; the modification is underlined twice). Another possibility is to let S generate a new key K for every run of the protocol ($S \models \#(K)$).

This short example proves clearly that a hand-made analysis is error-prone, and should be examined very cautiously.

Table 2. The enhanced Needham-Schroeder protocol

(a) The initial assumptions:

$P \ni K_{ps}$	$P \ni N_p$	
$P \mid\equiv P \overset{K_{ps}}{\longleftrightarrow} S$	$P \mid\equiv \#(N_p)$	$P \mid\equiv \Phi(Q)$
$Q \ni K_{qs}$	$Q \ni N_{q1}$	$Q \ni N_q$
$Q \mid\equiv Q \overset{K_{qs}}{\longleftrightarrow} S$	$Q \mid\equiv \#(N_{q1})$	$Q \mid\equiv \#(N_{q1})$
$P \mid\equiv S \mid\Rightarrow (P \overset{K}{\longleftrightarrow} Q)$	$P \mid\equiv S \mid\Rightarrow S \mid\equiv *$	$P \mid\equiv Q \mid\Rightarrow Q \mid\equiv *$
$Q \mid\equiv S \mid\Rightarrow (P \overset{K}{\longleftrightarrow} Q)$	$Q \mid\equiv S \mid\Rightarrow S \mid\equiv *$	$Q \mid\equiv P \mid\Rightarrow P \mid\equiv *$

(b) The Protocol:

1. $P \longrightarrow Q$: $\underline{P}$
2. $Q \longrightarrow P$: $\{P, N_{q1}\}_{K_{qs}}$
3. $P \longrightarrow S$: $\overline{P,\ Q,\ N_p,}\ \{P, N_{q1}\}_{K_{qs}}$
4. $P \longrightarrow S$: $\{N_p, Q, K, \overline{\{\mathrm{K}, \underline{N_{q1}}, \mathrm{P}\}_{K_{qs}}}\ \}_{K_{ps}}$
5. $P \longrightarrow Q$: $\{K, \underline{N_{q1}}, P\}_{K_{qs}}$
6. $Q \longrightarrow P$: $\{N_q, \underline{Q}\}_K$
7. $P \longrightarrow Q$: $\{N_q - 1\}_K$

3.2 Modified Wide-Mouthed-Frog Protocol

The second example illustrates the 'gap' that exists between an 'idealized' version of a protocol and its actual implementation.

Table 4-b shows a modified version of the 'wide-mouthed-frog protocol' (table 4-a; see also [1]). The role of the two timestamps in the original protocol is now played by a nonce, thus removing the need for synchronized clocks. However, an extra (first) message has to be sent.

When this protocol is analysed through the BAN-logic, the messages have to be idealized and the initial assumptions have to be listed. The first message is omitted, since it does not contribute to the logical properties of the protocol.

Since B trusts S, which says that A conveyed $A \overset{K}{\longleftrightarrow} B$, B trusts A's jurisdiction over the generation of such keys, and B knows that the last message is 'fresh' (it contains N_b), B can believe $A \overset{K}{\longleftrightarrow} B$.

In the BAN-logic, the participants should be able to recognize and ignore its own messages. There are two possibilities:

- either every participant stores every message sent in a database and checks every message received whether or not it is included in that database,

Table 3. The correctly enhanced Needham-Schroeder protocol

1.	$P \longrightarrow Q$: $\underline{P}$
2.	$Q \longrightarrow P$: $\{P, N_{q1}\}_{K_{qs}}$
3.	$P \longrightarrow S$: $\overline{P,\ Q,\ N_p,\ \{P, N_{q1}\}}_{K_{qs}}$
4.	$P \longrightarrow S$: $\{N_p, Q, K, \overline{\{\mathrm{K}, \underline{N_{q1}}, \mathrm{P}\}_{K_{qs}}}\ \}_{K_{ps}}$
5.	$P \longrightarrow Q$: $\{K, \underline{N_{q1}}, P\}_{K_{qs}}$
6.	$Q \longrightarrow P$: $\{N_q, \underline{\underline{N_p}}, \underline{Q}\}_{K}$
7.	$P \longrightarrow Q$: $\{N_q - 1\}_K$

Table 4. The wide-mouthed-frog protocol

(a) The original wide-mouthed-frog protocol

1.	$A \longrightarrow S : A,\ \{T_a, B, K_{ab}\}_{K_{as}}$
2.	$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}$

(b) The modified wide-mouthed-frog protocol

1.	$B \longrightarrow A : A,\ B,\ N_b$
2.	$A \longrightarrow S : \{A, B, N_b, K_{ab}\}_{K_{as}}$
3.	$S \longrightarrow B : \{A, B, N_b, K_{ab}\}_{K_{bs}}$

– or, every participant makes sure that he can recognize that a received message could not possibly originate from him.

In the non-idealized protocol description, it is clear that the second possibility is true. S never sends messages of the form $\{A, \ldots\}_{K_*}$ where $A \stackrel{K_*}{\longleftrightarrow} S$. However, messages received always have that format! If a programmer modified the message sent by S into $\{B, A, N_b, K_{ab}\}_{K_{bs}}$, because he could reuse a procedure used by participant A, then the implicit assumption that every participant can recognize its own messages is no longer true. Hence, the formal proof is not valid any more.

4 Action Based Input

From the previous section, we may conclude that it is extremely important to have (a) a fully automated theorem prover, but also (b) a protocol description

Table 5. The wide-mouthed-frog protocol

(a) Initial assumptions:

$A \mid\equiv A \xleftrightarrow{K_{as}} S$	$A \mid\equiv A \xleftrightarrow{K} B$
$B \mid\equiv B \xleftrightarrow{K_{bs}} S$	$B \mid\equiv \#(N_b)$
$B \mid\equiv (A \mid\Rightarrow A \xleftrightarrow{K} B\ )$	$B \mid\equiv (S \mid\Rightarrow A \mid\sim A \xleftrightarrow{K} B\ )$
$S \mid\equiv A \xleftrightarrow{K_{as}} S$	$S \mid\equiv B \xleftrightarrow{K_{bs}} S$

(b) Idealized protocol:

2. $A \longrightarrow S\ :\ \{N_b, \mathrm{A} \xleftrightarrow{K_{ab}} \mathrm{B}\ \}_{K_{as}}$
3. $S \longrightarrow B : \{N_b, A\mid\sim \mathrm{A} \xleftrightarrow{K_{ab}} \mathrm{B}\ \}_{K_{bs}}$

that is suitable for implementors. In the best case, the description can lead to an automatic code generation. This releases us from the burden of proving that an implementation corresponds to the (idealized?) specification of the protocol.

Our research group has built a theorem prover 'CryptoLog', which has action-based input (i.e. input written in some imperative language). The following rules were adopted when this input-language was designed:

- The initial assumptions should be reduced to the absolute minimal. They should only concern conditions that need be true **before** any run of the protocol. Hence, they list only trust in:
  - pre-installed (shared, public/private) keys,
  - jurisdictions of the participants.

  The assumptions are written in a rather intuitive language. Table 6 lists the assumptions that are understood by the analyser.
- All other assumptions about keys that are generated during a run of the protocol, about the validity/freshness of timestamps, the freshness of nonces, etc. are generated by the theorem prover as a result of particular **actions**. Table 7 lists the actions that are implemented by the analyser.

If the code that implements the protocol for each participant were written as a single procedure, the initial assumptions would be input parameters, be it pre-installed keys, or a list of participants (servers, trusted third parties, ...) that fulfil the jurisdiction conditions.

The second kind of assumptions refer to actual actions that have to be performed in the procedure.

Table 6. Assumptions understood by the Analyser

Assumption	Meaning
P, S, ... **Trust** K_{ps} :=: **SharedKey** (P, S)	P, S, ... that know K_{ps} is a suitable key for P and S
P **Trusts** K :=: **PublicKey**(Q)	P knows the public key of Q
P, S, ... **Trust** Pwd :=: **Password** (P, S)	P and Q share a password It may be subject to a dictionary attack!
P **Trusts** $Q \models\Rightarrow$ **CompetentHonest**	P believes in Q's competence and honesty
P **Trusts** $Q \models\Rightarrow$ **QualitySecrets** (S, T)	P trusts Q's jurisdiction over the generation of suitable shared keys for participants S and T

Table 7. Actions understood by the Analyser

Action	Meaning
P **Knows** Q	P knows the participant Q
P: V := **Value**	P calculates a value
P: N_p := **Nonce**	P generates a new nonce N_p
P: T_p := **CurrentTime**	P reads its current clock
P: **valid** (T_q)	P checks the validity of a timestamp T_q i.e. CurrentTime - $\Delta \leq T_q \leq$ CurrentTime + Δ
P: **fresh** (T_p)	P checks the freshness of a timestamp T_q
P: K := **SharedKey** (S, T)	P generates a new suitable key for participants S and T
P: K := **PublicKeyPair** (S)	P generates a public and secret key for participant S
P ---> Q: M	P sends message M to Q
P <--- Q: M	Q sends message M to P

5 The CryptoLog Analyser

The CryptoLog Analyser has been written in Prolog. Also, the input for the analyser (i.e. the protocol to be analysed) is a genuine prolog program: it consists of one or more Horn-clauses. Every initial assumption and every action is a subgoal of one of these clauses. In the next section, an extensive example of the input-syntax is shown.

The analyser performs the following tasks:

- it checks the soundness of every assumption/action;
  e.g. every assumption/action should be preceded by a principal name, ...
- it modifies the holds- and believes-relations;
  see tables 8 and 9.

- when the action is 'send a message', the analyser will additionally:
  - expand the included function calls:
    i.e. expand pre-defined short-cuts;
  - canonicalize the message;
  - append annotations:
    these are beliefs of the sender (see [3]);
  - check the sender-rules:
    the analyser verifies the possession and belief consistency rules (sender may only send messages it was able to assemble; and only current beliefs may be included as annotations in a message);
  - apply receiver-rules:
    in this step, the holds- and beliefs-relations of the receiver are expanded (if possible);
    currently, the deduction rules, described in the GNY-paper [3], have been implemented;
  - apply the eavesdropper-rules:
    in this step, the analyser checks whether secret information is leaked.
- at the end, the analyser will print an overview of the holds- and beliefs-relations.
- finally, for every password used as a cryptographic key, the analyser checks the severeness of a successful dictionary attack. It lists all the secrets that will become known to the eavesdropper if the attack succeeds.

Table 8. Transformation of the initial Assumptions

Assumption	**Generated Assumption(s)**
P, S ... **Trust** K_{ps} :=: **SharedKey** (P, S)	$P \mid\equiv P \overset{K_{ps}}{\longleftrightarrow} S$
	$S \mid\equiv P \overset{K_{ps}}{\longleftrightarrow} S$
P **Trusts** K :=: **PublicKey**(Q)	$P \ni +K$
	$P \mid\equiv \overset{K}{\longmapsto} Q$
	$P \mid\equiv \Phi(+K)$
	$Q \ni +K, -K$
	$Q \mid\equiv \overset{K}{\longmapsto} Q$
	$Q \mid\equiv \Phi(+Q)$, $\Phi(-K)$
P, S, ... **Trust** Pwd :=: **Password** (P, S)	idem as SharedKey
P **Trusts** $Q \mid\Rightarrow$ **CompetentHonest**	$P \mid\equiv Q \mid\Rightarrow *$
P **Trusts** $Q \mid\Rightarrow$ **QualitySecrets** (S, T)	$P \mid\equiv Q \mid\Rightarrow S \overset{K}{\longleftrightarrow} T$

Table 9. Transformation of actions by the Analyser

Action	Generated Assumption(s)
P **Knows** Q	$P \models \Phi(Q)$ $P \ni Q$
P: V := **Value**	$P \ni V$
P: N_p := **Nonce**	$P \ni N_p$ $P \models \#(N_p)$ $P \models \Phi(N_p)$
P: T_p := **CurrentTime**	$P \ni T_p$ $P \models \#(T_p)$ $P \models \#(T_p)$
P: **valid** (T_q)	$P \models \Phi(T_q)$
P: **fresh** (T_p)	$P \models \Phi(T_q)$ $P \models \#(T_q)$
P: K := **SharedKey** (S,T)	$P \models S \stackrel{K}{\longleftrightarrow} T$ $P \ni K$ $P \models \#(\mathrm{K})$ $P \models \Phi(K)$
P: K := **PublicKeyPair** (S)	$P \models \stackrel{K}{\longmapsto} S$ $P \ni +K$ $P \ni -K$ $P \models \#(+\mathrm{K})$ $P \models \Phi(+K)$ $P \models \#(\text{-K})$ $P \models \Phi(-K)$
P ---> Q: M	Q's beliefs may be extended
P <--- Q: M	P's beliefs may be extended

6 An Example: The Kerberos Protocol

Figures 1 and 2 show the input for the Kerberos protocol. It consists of three Horn-clauses, `kerberos`, `kerberosAssumptions` and `kerberosProtocol`. A call of the first clause:

```
?- kerberos.
```

starts the analyser. In this section, the input and output will be briefly explained. Comments are written between curly braces ({ ...}).

In figure 1, the participants that play a role are listed: C (the client), S (the server), AS (the authentication server) and TGS (the ticket granting server).

In the first part of the initial assumptions, the pre-installed password and keys are listed: `PWDc` is the password shared between the client C and the AS; `Kt` is the key shared between the AS and the TGS; finally `Ks` is the key known by S and TGS.

```
kerberos :-
   Principals ([C, S, AS, TGS]),
   kerberosAssumptions,
   kerberosProtocol.

kerberosAssumptions :-
   BeginAssumptions,
      { ------------------------------------------ }
      { I) Pre-installed keys / passwords / secrets }
      { ------------------------------------------ }
      (C,AS) Trust PWDc :=: Password(C,AS),    { Client C and the AS share a password }
      (AS,TGS) Trust Kt :=: SharedKey(AS,TGS), { The AS and TGS share a key }
      (S,TGS) Trust Ks :=: SharedKey(S,TGS),   { Server S and the TGS share a key }
      { ---------------- }
      { II) Jurisdictions }
      { ---------------- }
      (C, TGS) Trusts AS ==> CompetentHonest,        { The AS is competent & honest and }
      (C, TGS) Trusts AS ==> QualitySecrets(C,TGS),  { generates good keys }
      (C, S) Trust TGS ==> CompetentHonest,
      (C, S) Trust TGS ==> QualitySecrets(C,S),

      { ------------------------------- }
      { III) Definitions / Abbreviations }
      { ------------------------------- }
      Define ticket(Client,SessionKey,Timestamp,Lifetime,MasterKey) :=:
              Enc(Client,SessionKey,Timestamp,Lifetime,MasterKey),
      Define auth(Client,Timestamp, SessionKey) :=:
              Enc(Client, Timestamp, SessionKey),
EndAssumptions.
```

Fig. 1. Parser's input for the Kerberos protocol (part a)

The second part consists of the jurisdictions: who trusts who to generate good keys, and who trusts whose competence and honesty.

In the last part, a few abbreviations are defined. A call of these functions will be expanded according to the definition.

Figure 2 shows the last clause `kerberosProtocol`, which implements the actual protocol. It consists of three parts: the single sign-on, the retrieval of an appropriate ticket for the server, and the mutual authentication.

Figure 3 shows part of the result produced by the analyser. For every participant, it lists the holds- and beliefs-relations. At the end, an overview of leaked secret information is shown, and possible victims for a dictionary attack. For every successful attack, the consequences are listed too.

```
kerberosProtocol :-
   BeginProtocol,
      C: NcP := Nonce,  { Client generates a Nonce }
      C Knows TGS,

      { ----------------- }
      { A) Single Sign On }
      { ----------------- }

      C   --->   AS : (C, TGS, NcP),
                 AS: Kc := SharedKey (C, TGS),
                 AS: LTtgt := Value,          { AS generates the LifeTime }
                 AS: TStgt := CurrentTime,    { AS reads its clock }
      C   <---   AS : Enc(TGS, Kc, NcP, TStgt, LTtgt, ticket(C, Kc, TStgt, LTtgt, Kt), PWDc),

      { ------------------------------ }
      { B) Get a ticket for the server }
      { ------------------------------ }

      C Knows S,
      C: fresh(TStgt),        { Timestamp is fresh? i.e. no replay? }
      C: TSc := CurrentTime,  { Client reads its clock }
      C   --->   TGS: (S, ticket(C, Kc, TStgt, LTtgt, Kt), auth(C, TSc, Kc)),
                 TGS: fresh(TStgt),
                 TGS: fresh(TSc),
                 TGS: Kcs := SharedKey (C, S),   { TGS generates a new key }
                 TGS: TSt := CurrentTime,
                 TGS: LTt := Value,
      C   <---   TGS: Enc(S, Kcs, TSt, LTt, ticket(C, Kcs, TSt, LTt, Ks), Kc),
      C: valid(TSt),
      C: TScs := CurrentTime,

      { ------------------------ }
      { C) Mutual Authentication }
      { ------------------------ }

      C   --->    S : (ticket(C, Kcs, TSt, LTt, Ks), auth(C, TScs, Kcs)),
                  S: fresh(TSt),
                  S: fresh(TScs),
      C   <---    S : Enc(F(TScs), Kcs),

      EndProtocol.
```

Fig. 2. Parser's input for the Kerberos protocol (part b)

```
...
TGS >- Kc
TGS >- Kcs
TGS |= # Kc
TGS |= # Kcs
TGS |= C >- Kc
TGS |= AS |= C <-Kc-> TGS
TGS |= AS <-Kt-> TGS
TGS |= C <-Kc-> TGS
TGS |= C <-Kcs-> S
...
S >- Kcs
S |= # Kcs
S |= C >- Kcs
S |= TGS |= C <-Kcs-> S
S |= C <-Kcs-> S
...
```

```
>- means: holds
|= means: believes
#  means: is fresh
0  means: is recognizable

 AS >- Kc
 AS |= # Kc
 AS |= C <-Kc-> TGS
 ...
 C >- Kcs
 C |= S >- Kcs
 C |= AS |= C <-Kc-> TGS
 C |= AS |= # Kc
 C |= TGS |= # Kcs
 C |= TGS |= C <-Kc-> TGS
 C |= TGS |= C <-Kcs-> S
 C |= C <-Kc-> TGS
 C |= C <-Kcs-> S
 ...
```

```
+--------------------------------+
| Leakages of secret information |
+--------------------------------+
No Leakages Found!
+----------------------------+
| Dictionary attack victims |
+----------------------------+
DICTIONARY ATTACK POSSIBLE ON:
- Enc([TGS,Kc,NcP,TStgt,LTtgt,Enc([C,Kc,TStgt | LTtgt],Kt)],PWDc)
 through enumerating PWDc
Hence, the following information will be disclosed:
- [TGS,Kc,NcP,TStgt,LTtgt,Enc([C,Kc,TStgt | LTtgt],Kt)]
- PWDc
+====================================+
| IF DICTIONARY ATTACK SUCCEEDS, ... |
+====================================+
| Secret leaked to   | Trusted by | As being a secret   | Issued by |
| to the world       | principals | between principals  | Principal |
+===================+============+====================+===========+
|       PWDc        |      AS, C |              C, AS |   ??????? |
+-------------------+------------+--------------------+-----------+
|         Kc        | TGS, C, AS |             C, TGS |        AS |
+-------------------+------------+--------------------+-----------+
|        Kcs        |  S, C, TGS |               C, S |       TGS |
```

Fig. 3. Partial output of the analyser

7 Future Work

We intend to extend the analyser in several ways.

At this moment, the assumptions generated by the action-statements are hard-coded in the analyser. We would like to remove the definition of the action-statements from the analyser and give them as input to the theorem prover. This has three major advantages:

- the user can alter the definition
- they serve as implementation guidelines to the implementor
- it makes automatic code generation easier

For instance, the action `Nonce` ("generate a new nonce"), might mean different things in different protocols. In one protocol, it is sufficient that it is a number that has never been used before. Hence, a counter (kept in stable storage) that is incremented for each run of the protocol suffices. However, for another protocol, it might be essential that the nonce is not predictable (i.e. is chosen at random). By redefining the definition of `Nonce` we can easily accommodate the two different uses.

Secondly, we would like to add properties of cryptographic algorithms to the input part of the analyser and hence, remove the implicit assumption that the integrity of encrypted messages is always preserved.

However, a precondition for this extension is a complete type-system, which we intend to add to the analyser.

Thirdly, we are going to build a code generator that transforms the protocol specification into executable code. Since the specification contains useful implementation guidelines, this transformation should not be too difficult.

Finally, we will further investigate whether the analyser could easily be extended to predict hidden flaws (such as some attacks based on the interleaved execution of two runs of the protocol).

For instance, the analyser should be able to alert for the flaw in the original Needham-Schroeder protocol with public keys [1, 5]. See also table 10-a. An intruder C waits until A starts the protocol with C; C can start another run of the protocol with B, pretending to be A (see also table 10-b).

We have here two separate runs of the protocol, in which a message (2b) cannot be deciphered by C; however, C can reuse that message in the first protocol run and get the secret N_b from A.

We expect that it will not be too hard to add this capability to the analyser.

8 Conclusions

Logics for cryptographic protocols have proved to be of invaluable help in validating and correcting these protocols. However, they are not very helpful to the implementor of these protocols.

Table 10. The Needham-Schroeder protocol with public keys

(a) Original Protocol

1.	$A \longrightarrow B : \{N_a, A\}_{+K_b}$
2.	$B \longrightarrow A : \{N_a, N_b\}_{+K_a}$
3.	$A \longrightarrow B : \{N_b\}_{+K_b}$

(b) Interleaved execution of two runs of the protocol
C succeeds to authenticate as A to B

1a.	$A \longrightarrow C : \{N_a, A\}_{+K_c}$
1b.	$C \longrightarrow B : \{N_a, A\}_{+K_b}$
2b.	$B \longrightarrow C : \{N_a, N_b\}_{+K_a}$
2a.	$C \longrightarrow A : \{N_a, N_b\}_{+K_a}$
3a.	$A \longrightarrow C : \{N_b\}_{+K_c}$
3b.	$C \longrightarrow B : \{N_b\}_{+K_b}$

It is often difficult to grasp the subtleties of the initial assumptions. Some of them are implicitly imposed by the logic (such as integrity of encrypted messages). Moreover, the idealization process widens the gap between the specification and the implementation even further.

In this paper, we have presented an action-based syntax for the specification of cryptographic protocols, which can be analysed correctly and provides useful guidelines for implementors or even enable automatic generation of code.

References

1. M. Burrows, M. Abadi, R. Needham, *A logic of Authentication*, Digital System Research Center Report 39, Palo Alto, California, February 1989.
2. V.D. Gligor, R. Kailar, S. Stubblebine and L. Gong, *Logics for Cryptographic Protocols – Virtues and Limittions*, In Proceedings of the IEEE Computer Security Foundations Workshop IV, Franconia, New Hampshire, June 1991, pp. 219-226.
3. Li Gong, Roger Needham, and Raphael Yahalom, *Reasoning about Belief in Cryptographic Procols*, in Proceedings of the IEEE 1990 Symposium on Security and Privacy, Oakland, California, May 1990, oo. 234-248.
4. L.C. Paulson, *Isabelle, A Generic Theorem Prover*, Springer, 1994, LNCS 828.
5. G. Lowe, *Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR*, in T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Second International Workshop, TACAS'96, LNCS 1055, pp. 147-166, 1996.