Transforming a Proof System into Prolog for Verifying Security Protocols

Giorgio Delzanno[1] and Sandro Etalle[2]

[1] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, via Dodecaneso 35, 16146 Italy
e-mail: giorgio@disi.unige.it
[2] CWI, Amsterdam and Faculty of Computer Science,
University of Twente, P.O.Box 217, 7500AE Enschede, The Netherlands
e-mail: etalle@cs.utwente.nl

Abstract. In [11] we developed a proof theoretic method for the specification and the verification of security protocols in which protocol rules are described via first order theories and the interaction of the principals and of a possible intruder via a collection of proof rules. In this paper we show how the resulting logic can be transformed into a logic programming language so as to use Prolog-like system for validating security protocols. Furthermore, we discuss some choices we made which allow for an efficient execution of the code.

1 Introduction

Cryptographic protocols are the essential means for the exchange of confidential information and for authentication. Their correctness and robustness are crucial for guaranteeing that an hostile intruder may not be able to get hold of secret information (e.g. a private key) or to force unjust authentication. Unfortunately, the design of cryptographic protocols appears to be rather error-prone. A great deal of published protocols has later been shown to contain error prejudicing their safety. In various cases such flaws were found years after the appearance of a protocol, demonstrating how subtle such errors can be.

In [11] we proposed a *logic-based* method for reasoning over formal specifications of cryptographic security protocols. Our method is based on the following ideas. Protocol rules are described from the perspective of a generic principal using a set of *Horn clauses* that use special `knows` predicates to make assumptions over his current knowledge. Such a theory forms the core of the specification of a protocol. The interaction of a given set of principals is described instead via a collection of *proof rules*, each one describing how a generic rule affects the current state and knowledge of the principals involved in the protocol run.

The advantage of using an inference system is twofold. On one hand, by keeping separate knowledge, state, and interaction, it allows to modularly specify a security protocol. On the other hand, via a proof theoretic analysis of the resulting proof rules, it allows us to devise validation strategies which are sound and complete for interesting classes of protocols (including the well-known Needham-Schroeder protocol as specified, e.g., in [2]).

In this paper we will show how the ideas presented in [11] can be turned into a practical tool by using a logic programming system as implementation platform. Specifically,

(1) we will first show that the logic we introduced in [11] and the corresponding proof rules can be naturally transformed into *executable* Prolog specifications. The translation can be done modularly by following general patterns that transform proof rules defined over *multi-conclusion sequents* into Prolog clauses. By taking advantage of features typical of Prolog systems like built-in tracers it is then possible to use the resulting Prolog program for validating the original specification.
(2) We will show that the resulting Prolog specification can be further improved by using the notion of *delay* declarations, the logic programming counterpart of the *lazy evaluation* mechanism of functional programming used in [2] for the same purposes. The idea is as follows. Following the Dolev-Yao model of intruders [12], validating a protocol amounts to solving a search problem over an infinite state-space. Infiniteness is due to the capability of the intruder of generating messages with arbitrary content (e.g. in a logic-based encoding of protocols, terms with arbitrary size). Delay declaration can be used then to force the intruder to generate *on-demand* messages that are partially instantiated: only the pattern expected by one of the other principals must be explicitly generated, the remaining part of the message can be represented via existentially quantified variables.

In this paper we will focus on the presentation of the main ideas behind the combination of the Prolog encoding of the proof system described in [11] and delay declarations. We are currently working on the experimental part of this work.

2 Proof Theoretic Specification of Security Protocols

Using a proof system for representing a protocol comes natural if one considers that at each protocol transaction two kind of changes occur in the system: First, a change in the *knowledge* of the agents involved in the transaction, which is typically persistent and thus *monotonic*; secondly, a change in the *state* of the agents and of the network, which is typically *non-monotonic*. For instance, after having completed the first transaction of the Needham-Schroeder protocol [17] (reported in the Appendix), the agent Bob will know the nonce N_a generated by Alice. This knowledge is persistent, as it will not be modified by successive transactions. At the same time, the state of the network will change “negatively”, as Bob will remove the message he received from it, moreover the state of Bob self will change to indicate that Bob has completed the first transaction and is now ready for the second one.

In our approach, *knowledge* (which includes the specification of the protocol rules) is modeled by a set of horn clauses (augmented with inequality), denoted by Δ, and *states* are modeled via multisets of atomic formulas, denote by $\mathcal{S}$. We then define multi-conclusion sequents having the form $\Delta \longrightarrow \mathcal{S}$, that will be used to represent *(instantaneous) configurations* (i.e. current knowledge and state) during a protocol execution. A collection of *compound* proof rules will be used to specify the operational behaviour of the principals.

Definition 1 (Compound Proof Rule). *We call* compound proof rule *a rule of the form*

$$\frac{\Delta \vdash \phi \quad \Delta, \Delta' \longrightarrow \mathcal{S}'}{\Delta \longrightarrow \mathcal{S}}$$

where $\mathcal{S}$ and $\mathcal{S}'$ are multisets, Δ and Δ' are first-order theories, ϕ is a first-order formula and $\vdash$ is a given provability operator. When $\Delta, \Delta' \longrightarrow \mathcal{S}'$ is absent from the premise we call it a closing *rule.*

Here, Δ, Δ' denotes the set $\Delta \cup \Delta'$, while $\mathcal{S}$ is a multiset. In our system, each protocol transaction is modeled via such a rule, in which $\mathcal{S}$ and Δ are respectively the *global state* and *knowledge before* the transaction is fired; ϕ models the conditions under which the transaction can be fired; $\mathcal{S}'$ is the state of the system after the transaction is completed; Δ' is the *new knowledge*, acquired during the transaction. As we said, Δ and Δ' are sets of *Horn Clauses* in whose bodies we allow inequalities $\mathtt{t} \neq \mathtt{s}$ [1] other than this, negation is not involved. Finally, $\vdash$ denotes provability by SLD resolution: $\Delta \vdash \phi$ denotes the presence of a successful derivation *with the empty computed answer substitution* (thus if the query ϕ succeeds in Δ with c.a.s. θ, then we simply write $\Delta \vdash \phi\theta$). In our original system $\vdash$ denotes *ground (variable-free) resolution.* In the sequel we discuss the implications of moving to non-ground resolution.

A (partial) protocol execution starting from the state $\mathcal{S}_0$ and knowledge Δ_0 and ending in the state $\mathcal{S}_n$ and knowledge Δ_n is thus represented via a (partial) proof tree (here the proof trees for $\vdash$ are omitted)

$$\cfrac{\Delta_0 \vdash \phi_0 \qquad \cfrac{\vdots \qquad \cfrac{\Delta_n \vdash \phi_n \qquad \Delta_n \longrightarrow \mathcal{S}_n}{\vdots}}{\Delta_1 \longrightarrow \mathcal{S}_1}}{\Delta_0 \longrightarrow \mathcal{S}_0}$$

A proof is *successful* if each premise is satisfied. In order to build successful proofs *closing rules* must be part of the proof system. In our framework, non-closing rules are used for defining (specifying) the protocol and the intruder's behaviours. On the other hand, closing rules are used for specifying the kind of *properties* we wish to prove. Thus, by modifying the *closing rules* of the proof

[1] We don't need to refer to the closure of a program: the meaning of inequalities is provided by a suitable equality theory, which we assume to be part of Δ.

Compose

$$\frac{\Delta \vdash compose(id, step, m, n, \mathbf{K}) \quad \Delta, \Delta' \longrightarrow agent(id, step + 1), msg(m), \mathcal{S}}{\Delta \longrightarrow agent(id, step), \mathcal{S}}$$

$$\text{provided } n \text{ does not occur in } \Delta \longrightarrow agent(id, step), \mathcal{S}$$
$$\text{and } \Delta' = \{knows(id, k) \mid k \in \mathbf{K}\}$$

Expect

$$\frac{\Delta \vdash expect(id, step, m, \mathbf{K}) \quad \Delta, \Delta' \longrightarrow agent(id, step + 1), \mathcal{S}}{\Delta \longrightarrow agent(id, step), msg(m), \mathcal{S}}$$

$$\text{provided } \Delta' = \{knows(id, k) \mid k \in \mathbf{K}\}$$

Final

$$\frac{\Delta \vdash \mathbb{P} \quad \mathcal{S} \in \Phi_f}{\Delta \longrightarrow \mathcal{S}}$$

Intercept

$$\frac{\Delta, \mathcal{T}(\Delta, m) \longrightarrow \mathcal{S}}{\Delta \longrightarrow msg(m), \mathcal{S}}$$

$$\text{where } \mathcal{T}(\Delta, m) = \{knows(trudy, n) \mid \Delta \vdash decompose(m, n)\}$$

Contrive − expect

$$\frac{\Delta \vdash contrive(m) \wedge expect(id, s, m, \mathbf{K}) \quad \Delta, \Delta' \longrightarrow agent(id, s + 1), \mathcal{S}}{\Delta \longrightarrow agent(id, s), \mathcal{S}}$$

Fig. 1. The Compound Proof System for the protocol specification $\langle \Delta_0, \mathcal{S}_0, \Phi_f \rangle$, and the property $\mathbb{P}$.

system one can modify its meaning: a successful proof could be one validating a certain protocol, or just one cracking it.

Figure 1 shows the actual rules we employed in [11] for the verification of security protocol. $a, b, \mathcal{S}$ denotes the multiset containing a, b and the elements in $\mathcal{S}$ (i.e. in a sequent ',' represents multiset union). Predicates `compose`, `expect`, `intercept` and `contrive` are appropriately defined in Prolog, and their definitions are included in Δ. In particular, the Prolog definition for `expect` and `compose` alone fully specify the behaviour of the protocol: notice that in a security protocol an agent can do one of two actions: receive or send a message. In our system, `expect` fully defines the behaviour of a receiving agent: the query `expect(id,step,message,knowledge)` should succeed in the current global knowledge whenever agent `id` at step `step` can receive message `message`, and `knowledge` is the list of facts the agent learns during the transaction. On the other hand, the definition of `compose` specifies the behaviour of an agent when enabled to send a message: `compose(id,step,nonce,message,knowledge)` succeeds when agent `id` at step `step` can produce message `message`, possibly using the nonce (new, fresh value) `nonce`, and `knowledge` is the list of facts the agent learns during the transaction.

In addition to this, we have the rules *intercept* and *contrive*, whose purpose is modeling the behaviour of a possible hostile intruder. The hostile intruder is called by default *trudy*. The rule *intercept* models the possibility that trudy intercepts a message (and tires to decrypt it, learning its content), while the rule *contrive-expect* models the situation in which trudy forges a message and passes it to an honest principal. Notice that trudy has no state, this is reflected by the fact that the predicate symbols `contrive` and `decompose` have no *Id* nor *State* arguments: in fact the behaviour of the intruder does not change in time, and trudy is at each stage able to intercept and contrive fake messages.

Clearly, while the Prolog definitions for `expect` and `compose` depend from the protocol being examined, `contrive` and `decompose` are protocol-independent, and are assumed to be part of Δ.

3 Transformation into Prolog

We now show how we can literally translate the above proof system into Prolog. Let us take the rule *expect* as working example. First, one needs to eliminate functions application in the top-left side of the sequent. Special care has to be taken for mathematical expressions. After this operation the rule becomes

$$\frac{\Delta \vdash \mathit{expect}(id, s, m, \mathbf{K}) \wedge s' \ \mathit{is}\ s + \mathit{1} \qquad \Delta \cup \{\mathit{knows}(id, k) \mid k \in \mathbf{K}\} \longrightarrow \mathit{agent}(id, s'), \mathcal{S}}{\Delta \longrightarrow \mathit{agent}(id, s), \mathit{msg}(m), \mathcal{S}}$$

Then, we have to build a Prolog rule for modelling it. We use lists in order to model multisets, and – in order to benefit from the control predicates offered in Prolog – we decided not to write a meta-interpreter, but to incorporate Δ into the actual program (the same program `expect` and `compose` are defined in), and realize the changes that have to occur in Δ by means of `assert` and `retract` actions. It is worth mentioning that for our first prototype we *did* build a meta-interpreter, but that we eventually decided to change methodology mainly for debugging reasons: our system is also meant for simulation and testing of (security) protocols. To this end, it is important that bugs in the protocol be promptly traceable. By avoiding the use of a meta-interpreter, we could benefit directly from the (alas, not astonishing) debugging tools offered by the Prolog implementation. When we were using a meta-interpreter tracing and spying choice-points was a much more complicated matter, as we had to "see through" the meta-level to check what was happening at object level. As it turned out, using a non-meta program has another advantage which is invaluable for our purposes: namely the direct use of coroutining tools such as delay declarations. We are going to address this point later in the paper.

Each rule of the proof system, once modified according to the first step above is translated into a Prolog clause defining the predicate `state/1`, according to the following schema: the rule

$$\frac{\Delta \vdash A_1, \ldots, A_n \quad \Delta \cup \{knows(id, c_1), \ldots, knows(id, c_m)\} \longrightarrow s_1, \ldots, s_j, \mathcal{S}}{\Delta \longrightarrow t_1, \ldots, t_k, \mathcal{S}}$$

is transformed into the clause

```
state(State):- substate([t̄1,...,t̄k], State, Rest_of_state),
      Ā1,
      ⋮
      Ān,
      add_knowledge(īd, [c̄1,...,c̄m]),
      state([s̄1,...,s̄j|Rest_of_state]).
```

Where the over-lining is a straightforward mapping that takes care of translating atoms and terms according to Prolog's naming conventions (variables in uppercase, terms in lowercase, etc.). For instance, the clause resulting from the expect rule is:

```
state(State):- substate([agent(Id,Step),msg(M)],State,Rest),
      expect(Id,Step,msg(M),Info),
      NStep is Step +1,
      add_knowledge(Id,Info),
      state([agent(Id,NStep)|Rest]).
```

In the practical tool, we added an argument to `state`, so that it builds the trace of the actions performed. In the specific case the rule we use is:

```
state(State,[[Id,"receives",msg(M)]|Trace]):-
      substate([agent(Id,Step),msg(M)],State,Rest),
      expect(Id,Step,msg(M),Info),
      NStep is Step +1,
      add_knowledge(Id,Info),
      state([agent(Id,NStep)|Rest],Trace).
```

The predicate `add_knowledge` takes care of adding the appropriate clauses to the program. It has to behave in such a way that the clauses are removed upon backtracking (we need backtracking to be able to explore the search space of the traces in presence of an intruder). Since the clause we need to add are always ground, then we can undo an `assert(`c`)` simply by performing a `retract(`c`)`. Note that this would not be the case if c was a nonground clause. The code is simply:

```
add_clause(H):-
      (nonground(H) -> write('warning: nonground clause '),
```

```
                    write(H),
                    ;true),
      (assert(H);retract(H),!,fail).

add_knowledge(Id,[]).
add_knowledge(Id,[H|T]):- add_clause(knows(Id,H)),
      add_knowledge(Id,T).
```

4 Coroutining

As we have seen, the logical nature of Prolog allowed us to find an almost literal translation from the compound proof system into Prolog. The correctness of such translation is evident, and the don't know nondeterminism of Prolog implemented via backtracking allows for the systematic search of a proof. This, provided that we avoid non-terminating derivations, which is always a risk, considering that Prolog selection rule is not fair. Therefore, one of the aspects we have to check is the termination (and in general, the efficiency) of the code that results from this translation. Fortunately, Prolog allows for a number of control methods; in our particular case, *delay declarations* allow for a very efficient execution, which is terminating for a large class of protocol's specification. We now show why. The crucial rule in this respect is *contrive-expect*, which is translated as follows:

```
state(State):- substate([agent(Id,Step)],State,Rest),
      contrive(M),
      expect(Id,Step,M,Info),
      NStep is Step +1,
      add_knowledge(Id,Info),
      state([agent(Id,NStep),agent(trudy,NStepIntruder)|Rest]).
```

A bit of explanation is in order. This rule models a potential *intruder* that literally *forges* a message via `contrive(M)`, and then tries to feed it to a honest principal. If `expect(Id,Step,M,Info)` succeeds, then the honest principal has received and accepted the contrived message as if it was a legal one. Modeling an intruder correctly is crucial for verifying the protocol's correctness (see e.g. [12]). In this specific case, `contrive(M)` has clearly an infinite SLD tree, this reflects the fact that the intruder is allowed to contrive arbitrarily large and random messages.

In order to avoid nontermination we want `contrive` to generate only the messages that can be accepted by the corresponding `expect`, we can profitably exploit the availability of *logical variables*. For instance, if the definition of expect does not employ negation nor comparisons such as `t < s` (and the majority of protocols we saw could be modeled without using the negation nor $<$), then we can simply call `expect` *before* calling `contrive`; `expect` will (partially) instantiate `M` to the message the agent `Id` is actually expecting, and `contrive` only needs to check if that message is contrivable, eventually "filling the gaps" that `M` presents: i.e. instantiating the still free variables appearing in `M` with values the intruder knows. This makes the rule terminating. More in general, we can enforce an optimal execution strategy also without atom's reordering and without the limitation to specifications that do not employ inequalities and predicates such as $<$. Let us see briefly how `contrive` is defined. First of all, the grammar for the messages is the following:

```
Keys:       K ::= key(I)
Nonces:     N ::= nonce(I)
Objects:    O ::= K | N
Content     M ::= [] | [O|M] | enc(K,M)
Message:    msg(M)
```

Then the definition of contrive is

```
contrive(msg(M)) :- contrive_content(M).

contrive_content([H,H2|T]):- contrive_token(H),
      contrive_content([H2|T]).
contrive_content([H]):- contrive_token(H).
contrive_content(enc(key(K),M1)):-
   (  % if trudy knows the encryption key
      knows_object(trudy,K), contrive_content(M1)
   ;  % if trudy uses a previously intercepted message
      knows(trudy,msg(enc(key(K),M1)))).

contrive_token(M):- (M=nonce(N);M=key(N)), knows_object(trudy,N).
```

Where `knows_object` is an obvious predicate that succeeds returning an object (key or nonce) trudy is aware of (either because it had invented it in advance, or because it had intercepted it). In order to let it be terminating (and efficient), we simply have to add the declarations

```
delay contrive(X) until nonvar(X)
delay contrive_content(X) until nonvar(X)
delay contrive_token(X) until nonvar(X)
```

In this way, we achieve that `contrive` will not generate a message with an arbitrary structure, in fact, with the above delay declaration it follows the structure determined by the other agent (`expect`). In practice, in the above clause the call to `contrive(M)` will suspend until `expect(Id,Step,M,Info)` instantiates `M` to a non-variable term. This behaviour propagates by recursion.

It is worth noticing that this optimization is possible thanks to two factors. The first is that we are not using any longer a ground resolution method. When we developed the proof system we gave it only a ground semantics, which declaratively is perfectly suitable for our purposes. On the other hand, a direct implementation of such a semantics would unavoidably lead to a very inefficient execution strategy, as in the above rule contrive-expect we would be able to call `expect` only once `contrive` has finished making up a message. The second is that we use (bidirectional) logical variables in a non-trivial way. The use of logical variables allows for the interleaving of `contrive` and `expect`, which is a crucial improvement our system offers wrt other protocol specification system in the literature.

5 Conclusions

We have shown the Prolog implementation of combined proof systems we have employed for the verification of security protocols. The implementation is done by a literal translation into Prolog of the proof rules. After experimenting with a meta-interpreter, we decided to translate the system at "bottom-level", that is, without building a meta-interpreter, but by incorporating the clauses defining the rules in the same program that defines the predicates employed in the rules themselves. This – which at first might seem an unlucky design choice – allowed for a far superior debugging possibilities (the proof system is also employed for the *specification*, and thus the debugging, of security protocols), and in particular, for more accurate control strategies, which allow to avoid nontermination and *thus* yield to the *completeness result* we could state in [11]. We should mention that this result applies to the case of a single protocol run (it can easily be extended to the case of a bounded number of parallel sessions, but completeness is lost in the case of unbounded number of them).

To give a simple example of the properties that can be checked with our tool, one can – in the case of the Needham-Schroeder protocol – check whether the intruder can get hold of one of the nonces alice and bob have successfully exchanged. This can be done by adding the rule

```
final_knowledge:- knows(alice,other_nonce(Nb)),
   knows(bob,my_nonce(Nb)),
   knows(alice,my_nonce(Na)),
   knows(bob,other_nonce(Na)),
   (knows(trudy,object(Na));knows(trudy,object(Nb))).
```

And subsequently stating the query `init`. If the query succeeds then a trace is reported leading to the above undesired situation. If the query did not succeeded, then by completeness it would have failed, indicating the absence of such an attack.

6 Related Works

In recent years several *logic*-based approaches have been applied to verification problems of security protocols.

Among them, trace semantics is at the basis of the verification methods proposed, e.g., in [18, 19, 3, 2]. In [19], Paulson models a protocol in presence of an intruder as an inductively defined sets of traces, and uses Isabelle/HOL to *interactively* prove the absence of attacks. Paulson's approach works on an infinite-state search space. In our approach we use instead proofs structured as *trees* to represent protocol traces. Our approach is closer then to Basin's method [2], where *lazy data structures* are used to generate the infinite-search tree representing protocol traces in a *demand-driven* manner. To limit state explosion Basin applies however heuristics that prune the generated tree. In our work we get similar results via a formal analysis of the proof system: in this way we *only* generate proofs without useless steps of the intruder. By the presented translation into Prolog, we can exploit then an existing programming system as debugging tool.

In the past, many approaches based on finite-state model-checking tools have been investigated. To name a few, Casper [14], which employs FDR, and the work of Mitchell et al. [16]. As pointed out e.g., in [13], these approaches require ad-hoc solutions for limiting the search space. In contrast, our approach allows for a finite search space for all *fully-typed* protocols.

Our approach combines aspects related to the multiset-based approach of [7, 8] and the declarative way of specifying protocols using logic programs taken in [1, 10]. In [7], Cervesato et al. use a formalism based on multiset-rewriting (linear logic) to specify protocol rules and actions of intruders. Differently from [7], our sequent-calculi specification of proof rules allows us to separate in clear way knowledge and state and, thus reason about knowledge and specify every principal using very simple *Horn programs* (we use multi-conclusion rules only for specifying inter-agent communication).

A multiset rewriting formalism has also been used by Rusinowitch et al. by implementing it in OCAML in the CASRUL system [9], and, in the later [13] by processing the rewrite rules in the theorem-prover daTac.

The concept of *delayed* generation of intruder's messages have been considered by Rusinowitch and Turuani in [20] (a paper pointed out to us by one reviewer) for specification given via rewrite rules. In this approach *constraints* are used to symbolically represent the relation between the messages generated by the intruder and those expected by the other principals. In [20], the authors show that, in general, solving the resulting constraints is a difficult problem (NP-complete). Our approach has a different nature in that we try to exploit advanced operational aspects of logic programming in order to directly obtain the effect of a sort of *constraint solving* procedure over first order terms. An interesting direction to investigate might be the complexity of solving this kind of *term*-constraints in the case of *fully typed* protocols (the class introduced in [11]).

Logic programming languages have been applied to specify security protocols in several different ways. Meadow's NRL Protocol Analyzer [15] performs a reachability analysis using state-enumeration enriched by lemma proved by induction. This way, NRL can cope with a potential infinite search space. Our approach differs from this previous work firstly in that NRL explores the search space is explored in a backward fashion, i.e., starting from the unwanted situation and working backwards in order to find possible traces leading to it. Secondly, using proof theory we can formally reason on class of proofs and protocols for which finite-state exploration is both sound and complete [11]. Furthermore, Prolog is used in our approach to *declaratively* specify each principals.

In [1], Aiello and Massacci use a logic programming language with *stable semantics* to specify and debug protocols. In this setting knowledge, protocol rules, intruder capabilities and goals are specified in a declarative way. Apart from the different logic used to encode protocols, the method of [1] is proved complete only after fixing a bound on the length of runs in presence of the intruder.

Finally, Blanchet [4] uses Prolog to specify *conservative abstractions* that can be used to prove security protocols free from attacks. Intuitively, Horn clauses are used here as *constructors* and *deconstructors* for the messages exchanged on the net and intercepted by the intruder. The verification procedure combines aspects of *unfolding* and *bottom-up* evaluation (following, however, a depth-first strategy). A way of better understanding the relationships with our approach would be to extend the bottom-up evaluation strategy to the multi-conclusion setting as suggested in [5].

References

1. L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *Transactions on Computational Logic*, 2001.
2. D. Basin. Lazy infinite-state analysis of security protocols. *CQRE [Secure] '99*, 1999.
3. G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, editor, *Proc. 5th European Symposium on Research in Computer Security*, pages 361–375, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag LNCS 1485.
4. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In S. Schneider, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.
5. M. Bozzano. Ensuring security through model checking in a logical environment (preliminary results). In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Proc. Workshop on Specification, Analysis and Validation for Emerging Technologies (SAVE01)*, 2001.
6. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
7. I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *PCSFW: Proc. 12th Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society Press, 1999.
8. I. Cervesato, N. Durgin, J. Mitchell, Lincoln, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *PCSFW: Proc. 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.
9. Y. Chevalier, F. Jacquemard, M. Rusinowitch, M. Turuani, and L. Vigneron. CASRUL web site. `http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/`.
10. G. Delzanno. Specifying and debugging security protocols via hereditary harrop formulas and lambda prolog - a case study. In *Proc. 5th International Symposium on Functional and Logic Programming*, pages 123–137, 2001.

11. G. Delzanno and S. Etalle. A proof theoretic analysis of security protocols. Technical Report DISI-TR-01-08, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2001. http://www.cs.utwente.nl/~etalle.
12. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
13. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In M. Parigot and A. Vonkorov, editors, *Proc. LPAR: International Conference on Logic for Programming and Automated Reasoning*, number 1995 in LNCS. Springer-Verlag, 2000.
14. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.
15. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
16. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 Conference on Security and Privacy*, pages 141–153. IEEE Press, 1997.
17. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
18. L. C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, University of Cambridge, Computer Laboratory, January 1997.
19. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
20. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In S. Schneider, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.

A The Needham-Schroeder Protocol

For the referee's convenience, we report here the Needham-Schroeder protocol we refer to in the paper. The protocol allows two *principals*, say Alice and Bob, to exchange two *secret numbers*. The secret numbers will be used later for *signing* messages. In this setting, any agent (*principal*) involved in the protocol has a pair of keys: a *public* key, and a *secret* key. Typically, the secret key of agent A is used to decipher messages (encrypted with A's public key) that other agents send to A. The core of the protocol is defined as follows. Suppose that Alice knows the public key of Bob. As a first step, Alice creates a *nonce* N_a and sends it to Bob, together with its identity. A nonce is a randomly generated value used to defeat 'playback' attacks. The message is encrypted with Bob's public key, so that only Bob will be able to decipher the message. When Bob receives the message, he creates a new nonce N_b. Then, he sends the message $\langle N_a, N_b\rangle$ encrypted with the public key of Alice. Alice decrypts the message, and sends back the nonce N_b encrypted with the public key of Bob. At this point, Alice and Bob know the two secret numbers N_a and N_b. Following the notation used to describe security protocol [6], the protocol can be described via the following rules.

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