Proof Theory, Transformations, and Logic Programming for Debugging 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 this paper we define a sequent calculus to formally specify, simulate, debug and verify security protocols. In our sequents we distinguish between the *current knowledge* of principals and the *current global state* of the session. Hereby, we can describe the operational semantics of principals and of an intruder in a simple and modular way. Furthermore, using proof theoretic tools like the analysis of *permutability* of rules, we are able to find efficient proof strategies that we prove complete for special classes of security protocols including Needham-Schroeder. Based on the results of this preliminary analysis, we have implemented a Prolog meta-interpreter which allows for rapid prototyping and for checking safety properties of security protocols, and we have applied it for finding error traces and proving correctness of practical examples.

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 a hostile intruder can not 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. This gave impulse to research on the formal verification of security protocols see e.g. [13, 6, 20, 23, 18, 28, 29]. In this setting several approaches are based on Dolev and Yao's [13], where it is proposed to test a protocol explicitly against a hostile intruder who has complete control over the network, can intercept and forge messages. By an exhaustive search, one can establish whether the protocol is flawed or not as shown, e.g., in [23, 21, 8, 16]. Clearly, a crucial aspect in this approach is try to limit the search space explosion that occurs when modelling the intruder's behaviour . To this end, many solutions have been employed, ranging from human intervention to the use of approximations. In recent work [15, 30, 22], this problem has also been tackled by reducing the intruder's action to a constraint solving problem.

The origin of this paper was our intention to investigate the possible application of existing logic programming systems for debugging and verification of security protocols. Logic programming tools provide natural built-in mechanisms like backtracking to explore the search space generated by a transition system. A direct implementation of a protocol simulator, however, would suffer from problems like state explosion, infinite-derivations, etc. To tackle these problems, in this paper we use a combination of techniques stemming from proof theory and program transformation. Proof theory allows us to formally specify the protocol and intruder behaviours. Here, a systematic study of the structure of the resulting proofs serves as formal justification for an interesting series of optimizations. Via sound and complete transformation, we will derive specialised rules that form the core of the simulator we implemented in Prolog. Our techniques also allows us to isolate a class of protocols, we called *fully-typed* for which, in case of finite-number of sessions, forward exploration is guaranteed to terminate.

The reason we have chosen to specify protocols using sequent calculus is based on the following observation. During the execution of a protocol stage two kind of changes take place: a change in *knowledge* that is monotonic, as it is not modified during later stages, and a change of *state* that is non-monotonic (e.g. the presence of messages on the network). In our approach the knowledge is modelled by a first-order theory, and the state by a multiset of first order atoms. A protocol can be specified in a natural way by a *multi-conclusion proof system* in which every proof rule corresponds uniquely to a legal protocol action. In this setting a proof corresponds to a *protocol trace*. In addition to this, the proof system allows to model a potential intruder by adding few rules modelling its behaviour. It is then possible to check if the intruder has a way of breaking the protocol.

Via a natural translation of our specialised proof system into Horn clauses, we obtained a working prototype that we used in some practical experiments. The prototype is written in Prolog, and allows the user to formally specify a cryptographic protocol, by writing Prolog rules defining it. We want to mention that the Prolog prototype employs in a crucial way the notion of *delay* declarations. Specifically, delay declaration are used to make the *intruder* 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.

Plan of the paper. In Section 2, we present the language (formulae and sequents) used to describe security protocols and safety properties. In Section 3, we present additional proof rules for modelling the intruder. In Section 4, we analyse the resulting proof system, and we introduce the notion of fully-typed protocols. In Section 7, we briefly describe our Prolog implementation of a automated prover for our logic, some experimental results, and, finally, address related works.

2 Proof Theoretic Specification of Security Protocols

The Needham-Schroeder Protocol As a working example, we will consider the security protocol based on public-key cryptography proposed by Needham and Schroeder in [25]. The protocol allows two *principals*, say Alice and Bob, to exchange two *secret numbers*, that might be used later for *signing* messages. At first, Alice creates a *nonce* N_a and sends it to Bob, together with its identity. A nonce is a randomly generated value. 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 deciphers the message, and sends back the nonce N_b encrypted with the public key of Bob. At this point, Alice and Bob know both N_a and N_b. Following the notation commonly used to describe security protocol [6], the protocol has the following rules.

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

2.1 The Proof System

Using a proof system for representing a protocol is 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, that is typically persistent and thus *monotonic*; secondly, a change in the *state* of the agents and of the network, that is typically *non-monotonic*. In our approach, *knowledge* (which includes the specification of the protocol rules) is modelled by a first order theory, denoted by Δ, and *states* are modelled via multisets of atomic formulae, denoted by $\mathcal{S}$. We then define multi-conclusion sequents having the form $\Delta \longrightarrow \mathcal{S}$, that will be used to represent *(instantaneous) configurations* during a protocol execution. The advantage of this approach wrt either coding immediately the whole system in Prolog (as in the NRL analyser [21]), is that in this way we can more easily prove properties of the whole system such as Theorems 4, 5 and 7. We will use *compound proof rules* to specify the behaviour of the principals. A *compound proof rule* has 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 $\Delta \cup \Delta'$. Each protocol transaction is modelled via a compound rule where $\mathcal{S}$ 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. In the rest of the paper Δ and Δ' will be sets of *Horn Clauses*, i.e. universally quantified implicational formulae indicated as $a \leftarrow b_1 \wedge ... \wedge b_n$ where $a, b_1, ..., b_n$ are atoms. We allow b_i to be an equality or an inequality $t \neq s$, other than this, negation is not involved, and no other tests are

allowed (like $<$, etc.). For the moment, the relation $\vdash$ will denote the provability relation built on top of *ground (variable-free) resolution*. A (partial) protocol execution starting from the state $\mathcal{S}_0$ and knowledge Δ_0 (i.e., in the configuration $\Delta_0 \longrightarrow \mathcal{S}_0$) and ending in the configuration $\Delta_n \longrightarrow \mathcal{S}_n$ is thus represented via a (partial) proof tree (here and in the sequel we omit the proof trees for $\vdash$ when possible) A proof is *successful* if each premise is satisfied. We will use non-closing rules to specify the protocol and the intruder and a closing rule to specify the (reachability) properties we wish to demonstrate or refute. Thus, by modifying the closing rules of the proof system one can modify its meaning.

Knowledge and State Description Language The *global state* of the system is described via a multiset containing *agent states* and *messages*. A *Agent State* is an atom `agent(ID,Role,Step)` where `ID` is the agent's identifier, `Role` is its role in the protocol (e.g., *initiator* or *responder*), and `Step` is a counter denoting the current step of the protocol the agent is at. In presence of parallel sessions, some agent ID may occur more than once in the global state, in particular, it may occur while having different roles (agent "a" could be at the same time initiator of one protocol session and responder in another session). On the other hand, *Messages* contain lists of objects (keys or nonces), that may in turn be encrypted. We use `enc(K,M)` to represent a message encoded with the (public) key `K`. More precisely, the set $\mathcal{M}$ of messages is defined by the following syntax (symmetric encryption is omitted for brevity): keys and nonces are represented as `key(I)` and `nonce(I)` where `I` is an integer; objects are either keys, nonces or terms like `enc(K,M)`; list of contents are either empty `[]`, or `[O|M]` where `O` is an object, and `M` is a list of contents; finally a message is a term `msg(M)`. *Knowledge* is encoded in *Horn clauses*. Knowledge can be roughly be divided into *global* knowledge, that is common to all principals (like the rules for describing a protocol, explained in the next section) and the agents' *private* knowledge, that is encoded in unit clauses of the form `knows(ID,D)`, where `ID` is the identifier of the agent that possesses it, and `D` is a term such as `keypar(k1,k2)`, `key(k3)`, `nonce(n)` and (overloading the notation) `msg(m)`, where `k1`, `k2`, `k3`, `n` are objects, `m` is a message, and the function symbols `keypar`, `key`, `nonce`, `msg` can be seen as labels, that serve as place-holders for facilitating the retrieval of stored values.

Protocol Specification A protocol specification allows us to describe all traces starting from a given initial state. Formally, a protocol specification is a pair $\langle \Delta_0, \mathcal{S}_0 \rangle$ consisting of the initial knowledge Δ_0, and the initial global state $\mathcal{S}_0$. In turn, the initial knowledge Δ_0 can be seen as the union of Δ_{rules} (the rules that specify the protocol) and $\Delta_{knowledge}$ (the agents' initial knowledge).

The Protocol Rules, Δ_{rules}. The rules of a generic protocol are always defined according to a fixed pattern: an agent receives a message, removes it from the communication media, composes a new one, and adds it to the communication media. In order to model a protocol rule like $A \rightarrow B : M$, we simply need to specify which messages an agent *expects* or *composes* at a given step of the

protocol. For this purpose, we use a special predicate, namely `compose`, to describe the behaviour of the sender A as well as the structure of the message M, and a special predicate, namely `expect`, to describe the behaviour of the receiver B. The *synchronisation* of the agents can be left as part of the *operational* semantics defined via our proof system. `expect` has the following fixed signature: `expect(ID,Role,Step,Message,Knowledge)`. One can think of the first four argument as being *input arguments*, while the last one is an *output* one. The query `expect(ID,Role,Step,Message,Knowledge)` should succeed in the current global knowledge whenever agent having identifier `ID` and role `Role` at step `Step` can receive message `Message`; `Knowledge` is the list of facts the agent learns during the transaction, represented by a list of terms of the form $identifier(value_1, \ldots, value_n)$. For instance, Bob's role in the first transaction is specified by:

```
expect(ID,responder,1,msg(enc(key(Pkb),[key(Pka),nonce(Na)])),Info):-
   knows(ID,keypar(_,Pkb)),knows(ID,key(Pka)),
   Info = [other_nonce(Na),other_key(Pka)].
```

here `Info` is used by the responder to memorise the reception of nonce `Na` from the agent having public key `Pka`. The predicate `compose` is used to compose new messages. In `compose(ID,Role,Step,Nonce,Message,Knowledge)`, one can think of the first four argument as being *input arguments*, while the last two are *output*. The extra input argument `Nonce` is used for passing a fresh nonce to the rule, in case that it is needed for composing a message. The query `compose(ID,Role,Step,Nonce,Message,Knowledge)` succeeds when agent `ID` in role `Role` at step `Step` can produce message `Message`, possibly using the nonce `Nonce`, and `Knowledge` is the list of facts the agent learns during the transaction. In `expect` and `compose` we only allow *equalities* and *inequalities* between free variables but not relations like $>$ (less than). Summarising, Δ_{rules} is the logic program that contains the definitions of `expect` and `compose`.

The Initial Knowledge. It describes the initial knowledge of each agent via a set of *unit clauses*. For a single session of Needham-Schroeder, it can consists of the set of atoms: { `knows(a,keypar(ska,pka)).`, `knows(_,key(pka)).`, `knows(b,keypar(skb,pkb).`, `knows(_,key(pkb)).`, } where the underscore `_` is the anonymous variable: `knows(_,key(pka))` indicates that every agent knows the public key of a; here "a" and "b" are the identifiers of the agents involved.

Initial Global State. $\mathcal{S}_0$ is the multiset of agents in their initial state. It determines which agents are present and their role. Therefore, it determines also how many *parallel sessions* we investigate (by doing so, we allow only a finite number of parallel sessions). For the single session of Needham-Schroeder, the initial global state is $\mathcal{S}_0 = \{agent(a, initiator, 1), agent(b, responder, 1)\}$. If we wanted to analyse two sessions of this protocol, in which agent a has role *initiator* in one session and role *responder* in the other session, all we needed to change is the initial state, which would be: $\mathcal{S}_0 = \{agent(a, initiator, 1), agent(b, responder, 1), agent(a, responder, 1), agent(c, initiator, 1)\}$.

Proof Rules Once we have a protocol specification $\langle \Delta_0, \mathcal{S}_0 \rangle$, we need to embed it in a proof system, whose interleaving operational semantics will model the protocol behaviour.

Definition 1 (Proof System for a Protocol Specification). *The* (compound) proof system derived from *a protocol specification* $\langle \Delta_0, \mathcal{S}_0 \rangle$*, consists of the following two rules*

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

provided n *does not occur in* Δ *and* $\mathcal{S}$*;* $\Delta' = \{knows(id, k) \mid k \in \mathbf{K}\}$.

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

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

Where $agent(id, role, step+1), msg(m), \mathcal{S}$ indicates the multiset consisting of $agent(id, role, step+1)$, $msg(m)$ and the element of the multiset $\mathcal{S}$. Notice that these rules depend on the predicates `expect` and `compose` that are defined in Δ. The side condition of the *send* requires n to be a fresh nonce. Actually, the above rule applies when `send` needs at most one fresh nonce. Rule for where more nonces are needed are obtained via a straightforward generalisation.

The Safety Property Π Intuitively, our objective is to answer questions like: *is it possible to reach a final state in which the intruder has got hold of the nonces* N_a *and* N_b*?* For this purpose, we interested in *safety* properties that can formulated in terms of reachability properties (if a given unwanted situation is not reachable, then the system satisfies the safety property). To describe the state to be tested for reachability we use a special query $closing(\mathcal{S})$, whose definition is given in a separate program Π (to be added to Δ) and we add to the proof system the following closing rule.

$$\frac{\Delta \vdash closing(\mathcal{S})}{\Delta \longrightarrow \mathcal{S}} \; final$$

For instance, to check if two agents can actually reach the end of the protocol while exchanging each others nonces, we need the following program Π

```
closing(S):- subset([agent(ID1,initiator,4),agent(ID2,responder,4)],S),
             knows(ID2,other_nonce(N_a)),knows(ID1,other_nonce(N_b)),
             knows(ID2,my_nonce(N_b)),knows(ID1,my_nonce(N_a)).
```

Proofs as Traces The proof system allows us to formally describe all possible *traces* of an (interleaving) execution of the principals. A trace can be viewed in fact as a single threaded proof for the $\longrightarrow$-sequents leading to a final state, and in which all auxiliary conditions are satisfied. Formally, a proof is a sequence of sequents $\Delta_0 \longrightarrow \mathcal{S}_0, \ldots, \Delta_k \longrightarrow \mathcal{S}_k$ where $\Delta_i \longrightarrow \mathcal{S}_i$ and $\Delta_{i+1} \longrightarrow \mathcal{S}_{i+1}$ correspond respectively to the lower and upper sequent of an instance of one of the proof rules *receive*, and *send*, in which all auxiliary sequents of the form $\Phi \vdash \phi$ are provable for each $0 \leq i < k$. Finally, $\Delta_k \longrightarrow \mathcal{S}_k$ is the lower sequent of an instance of the *final rule* in which all premises are satisfied. The following property then holds.

Proposition 2 (Soundness). *Given a protocol specification $\langle \Delta_0, \mathcal{S}_0 \rangle$, and a safety property specification Π, every proof for the sequent $\Delta_0, \Pi \longrightarrow \mathcal{S}_0$ corresponds to a trace that from the initial state of the protocol leads to a final configuration $\Delta \longrightarrow \mathcal{S}$ in which a closing rule of Π can be fired.*

3 Modelling an Intruder

In order to test a protocol specification against a malicious intruder, we assume the existence of a new agent, called *Trudy*, that has complete control over the network. The task of Trudy is to get hold of secret information and/or to spoil the communication between the honest principals, for instance by letting them believe they have correctly exchanged nonces, while they have not. Our system can be used for checking if such a malicious intruder is actually able to fulfil its objectives. This is achieved by providing the following fixed rules for specifying Trudy's behaviour

$$\frac{\Delta, \mathcal{T}(\Delta, m) \longrightarrow \mathcal{S}}{\Delta \longrightarrow msg(m), \mathcal{S}}\ intercept \qquad \frac{\Delta \vdash contrive(m) \quad \Delta \longrightarrow msg(m), \mathcal{R}}{\Delta \longrightarrow \mathcal{R}}\ forge$$

where $\mathcal{T}(\Delta, m) = \{knows(trudy, n) \mid \Delta \vdash decompose(m, n)$. In the rule for intercepting a message, the set of facts $\mathcal{T}(\Delta, m)$ represents the *closure* of the knowledge of the intruder with respect to what Trudy can decipher using the predicate `decompose`. Following from this definition, given a message m (i.e. a ground term), $\mathcal{T}(\Delta, m)$ always consists of a finite set of facts. Trudy can also create a message in order to try to cheat the other principals. Starting from her current knowledge, Trudy uses the predicate `contrive` to generate a random message of arbitrary size. This message is placed in the global state via the *forge* rule. Clearly, Trudy has to possess at least a public-private key pair, and a certain number of random numbers (nonces) with which it contrives messages (if the knowledge of Trudy is empty, Trudy is not able to contrive anything). Thus, an *intruder* is a set of clauses containing the rules for `decompose` and `contrive` together with a *set* of facts each one storing a new nonce, $\texttt{knows}(trudy, nonce(n))$, where n is a fresh integer, or a pair of keys, i.e., $\texttt{knows}(trudy, keypar(sk, pk))$ where sk, pk are fresh terms. We can now analyse the protocol traces in presence of Trudy. The *extended proof system* derived from a protocol specification

$\langle \Delta_0, \mathcal{S}_0 \rangle$, a safety property specification Π and combined with an intruder specification Δ_{trudy} consists of the rules *intercept* and *forge*, together with the rules *send*, *receive* and *final*. Now, we can describe all possible *traces* of an interleaving execution of the principals and of Trudy as stated in the following theorem.

Proposition 3 (Soundness). Given a protocol specification $\langle \Delta_0, \mathcal{S}_0 \rangle$, a safety property specification Π, every proof for the sequent $\Delta_0, \Pi, \Delta_{trudy} \longrightarrow \mathcal{S}_0$ corresponds to a trace of an interleaved execution of the principals and an intruder Δ_{trudy} that from $\mathcal{S}_0$ leads to a final configuration $\Delta \longrightarrow \mathcal{S}$ that satisfies the condition specified by Π.

For example, it is possible that after the protocol has completed Trudy got hold of one of the secret nonces. This is done by taking Π to be the following program.

```
closing(S):- subset([agent(ID1,initiator,4),agent(ID2,responder,4)],S),
             knows(ID2,other_nonce(N_a)), knows(ID1,other_nonce(N_b)),
             knows(ID2,my_nonce(N_b)),    knows(ID1,my_nonce(N_a)),
             (knows(trudy,object(N_a));knows(trudy,object(N_b))).
```

4 Analysis and Tuning of the Proof System

During a protocol execution, Trudy can generate an arbitrary number of messages, and at any step Trudy can generate a message of arbitrary size. Thus, the search space is infinite. In this section we show that by *proof theoretic* analysis we can drastically reduce the search space, and identify a large class of protocol specification for which the search space is finite. First, we have to solve the problem of the initial knowledge of the intruder. In fact Proposition 3 depends on a specific intruder Δ_{trudy}, that can be arbitrarily large. The following (new) result shows that it is possible to build a (small) Δ_{trudy} which suffices:

Theorem 4. *Given a protocol specification $\langle \Delta_0, \mathcal{S}_0 \rangle$, and a safety property specification Π then there exists an intruder $\Gamma_{\Delta_0,\Pi}$ such that for any other intruder Δ_{trudy}: If there exists a proof for $\Delta_0, \Pi, \Delta_{trudy} \longrightarrow \mathcal{S}_0$ then there exists a proof for $\Delta_0, \Pi, \Gamma_{\Delta_0,\Pi} \longrightarrow \mathcal{S}_0$.*

Secondly, we have to avoid possibly infinite branches. It is straightforward to check that this can only be achieved by bounding the number of the forge rule in a proof. We show now that each proof is equivalent to a proof in which the number of forge rules is equal to the number of expect rules (which is bounded by definition). We start by noting that we can move and remove some occurrence of the forge rule, preserving equivalence. Our proof system enjoys, in fact, the following properties. (a) The rule *forge* always permutes up with *send*: e.g. by permuting two adjacent send and forge rule, one obtains an equivalent proof. (b) The rule *forge* permutes up with *intercept* and *receive* whenever the contrived message is different from the intercepted or expected message. (c) An occurrence of the forge rule which is followed by the rule *final* can be removed provided we assume that the presence of messages in $\mathcal{S}$ does not influence the result

of the query $closing(\mathcal{S})$. (d) The rule *forge* and *intercept* cancel each other out whenever they are defined on the same message. In this case Trudy first contrives a new message m using her current knowledge in Δ and then intercepts it and decomposes it using again her current knowledge. It is not difficult to prove that in this case we can prune both rule instances form the proof.

The above properties demonstrate that we can restrict to proofs in which each forge rule is followed by (i.e. is just below) a receive rule, that reads the message generated by `contrive`. Since the description of a protocol is finite, there might be only a finite number of receive rules in a proof. Thus, if we disregard the steps needed to prove $\vdash$, such proofs are of bounded depth; in practice we can now restrict to bounded traces.

The third and last source of nontermination comes from the fact that, declaratively speaking, $\Delta \vdash contrive(m)$ might have an arbitrarily large proof. Operationally, `contrive` might generate an arbitrarily large message. In order to deal with this we need one last transformation operation. Consider the following schema

$$\dfrac{\Delta \vdash contrive(m) \qquad \dfrac{\Delta \vdash expect(i,r,m,s,\mathbf{K}) \quad \Delta, \Delta' \longrightarrow agent(i,r,s'),\mathcal{S}}{\Delta \longrightarrow msg(m), agent(i,r,s),\mathcal{S}}\ receive}{\Delta \longrightarrow agent(i,r,s),\mathcal{S}}\ forge$$

where $s' = s + 1$, $\Delta' = \{knows(id,k) \mid k \in \mathbf{K}\}$. Thus, we derive the rule

$$\dfrac{\Delta \vdash contrive(m) \wedge expect(i,r,s,m,\mathbf{K}) \quad \Delta, \Delta' \longrightarrow agent(i,r,s'),\mathcal{S}}{\Delta \longrightarrow agent(i,r,s),\mathcal{S}}\ forge\text{-}receive$$

This transformation alone does not guarantee termination: we have to combine it with a suitable search strategy. We do this in the next section. The following property summarises what we achieved so far.

Theorem 5 (Proof Normalisation). Given a protocol $\langle \Delta_0, \mathcal{S}_0 \rangle$, and a property Π, any proof π for the sequent $\Delta_0, \Pi, \Gamma_{\Delta_0,\Pi} \longrightarrow \mathcal{S}_0$ is equivalent to a proof π' that makes use of the derived rule *forge-receive* and in which there are no occurrences of *forge*.

Proofs in which the *forge* rule does not occur are called *normal* proofs.

4.1 Completeness of Normal Proofs for Fully-typed Protocols

The derived rule *forge-receive* is still nondeterministic and this can be an obstacle when trying to automatically build a proof. The problem is that `contrive` might still generate arbitrarily large messages. We note however that in protocols like Needham-Schroeder the predicate `expect` puts severe limitations to the nondeterminism of `contrive`. The crucial point here is that if `expect(id,r,s,m,k)` succeeds, then the shape of `m` is uniquely determined by `id` and `s`. This inspires the following definition of fully-typed protocol.

Definition 6 (Fully-Typed Protocol). A protocol $\langle \Delta_0, \mathcal{S}_0 \rangle$ is *fully-typed* if for any agent knowledge Δ, if $\Delta_0, \Delta \vdash expect(id, r, s, m_1, i_1)$ and if $\Delta_0, \Delta \vdash expect(id, r, s, m_2, i_2)$, then the m_1 is equal m_2 modulo *renaming* of constants terms.

If the protocol is fully typed, given *id* and *s*, each expected message has a fixed *term* structure in which only *constant* symbols are allowed to vary. By using abstract interpretation one can effectively check if a protocol is fully typed.

Fully-typed protocols allow for an effective use of the *forge-receive* rule by a guided generation of messages. The idea is to interleave the Prolog execution of `expect` and `contrive`: after extracting the pattern of an expected message using `expect(id,r,s,m,i)`, `contrive` simply has to check whether the message is "contrivable" (e.g., that it is not encrypted using a key that is not known to Trudy) and to fill all remaining holes using new nonces or keys and nonces Trudy has stored in previous steps. This can be naturally done within any Prolog implementation that allows the presence of *delay declarations*. This renders finite the search space, and leads us to the following completeness result.

Theorem 7 (Completeness of Normal Proofs). Given a fully typed protocol specification $\langle \Delta_0, \mathcal{S}_0 \rangle$, and a property specification Π, there exists an implementation $\vdash$ such we can decide whether or not there exists a proof for $\Delta_0, \Pi, \Gamma_{\Delta_0,\Pi} \longrightarrow \mathcal{S}_0$.

5 Implementation

We now show how we can literally translate the proof system presented in the previous section into Prolog. However, we want to stress that the framework so far developed is amenable to different implementation methodologies (e.g., bottom-up instead of top-down). 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, rule *expect* becomes

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

Where $\Delta' = \{knows(id, k) \mid k \in \mathbf{K}\}$. 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 realise 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 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. 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.

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

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

is transformed into the clause

```
state(State):- substate([t̄1,...,t̄k], State, Rest_of_state),Ḡ,
   add_knowledge(id̄, [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.). 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`)`.

6 Coroutining

As we have seen, the logical nature of Prolog allowed us to find an almost literal translation from the proof system into Prolog. The correctness of such translation is evident. Completeness is guaranteed provided that we avoid non-terminating derivations, which is always a risk, considering that Prolog selection rule is not fair. 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 *forge-receive*, which is translated as follows:

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

This rule models Trudy forging a message via `contrive(M)`, and trying to send it to the agent whose identifier is Id. If `expect(Id,Role,Step,M,Info)` succeeds, then the honest principal has accepted the contrived message as a legal one. The problem here is that `contrive(M)` has an infinite SLD tree (the intruder is allowed to contrive arbitrarily large and random messages). In order to avoid nontermination we can force `contrive` to generate only those messages that can be accepted by the corresponding `expect`; for this we can profitably exploit the availability of *logical variables*. For instance, if the definition of expect does not employ negation 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. 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)):-
   (  knows_object(trudy,K), contrive_content(M1)
   ;  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 such that `knows_object(trudy,K)` succeeds if `K` is 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)
```

With these declarations, `contrive` will only generate messages that are readable by the other agent involved in the communication. It is worth noticing that this optimisation is possible thanks to two factors. The first is that we are not using any longer a ground resolution method. Declaratively, a ground semantics, is perfectly suitable. However, a direct implementation of such a semantics would unavoidably be very inefficient. The second is that we use (bidirectional) logical variables in a non-trivial way.

7 Conclusions

The contribution of this paper is threefold: (1) a proof-theoretic method for specifying security protocols. (2) a proof-theoretic methodology for specialising the proof system wrt the underlying protocol. It is worth mentioning that part of the specialisations we presented in Section 4 and in Section 6, were also present in previous papers, but only as "rule of thumb"; this is the first paper in which such optimizations ore proved correct, by means of rule transformations. (3) An implementation in Prolog of the system, which is now a tool allowing to specify, simulate and thus debug security protocols. In addition, it allows to verify safety properties of protocols, and in this we proved that it achieve completeness when protocols are *fully-typed*.

Theorem 7 applies when we consider a bounded number of parallel execution of the same protocol. The tool we have developed can be easily extended

to cover also unbounded number of sessions, but completeness would be lost: if no attack exists the tool might run forever (Durgin et al. showed in [14] that the problem of finding an attack in presence of unbounded number of sessions is undecidable). Our future work focus on finding safe approximations for guaranteeing the termination of the tool also in presence of unbounded number of parallel sessions.

Lowe has tackled this problem in an orthogonal way. In [19] he demonstrates that if the protocol satisfies certain conditions, a parallel attack exists if and only if an attack exists within a single run of the protocol. For such protocols, demonstrating absence of attacks in a single session (which can be done with our tool) is sufficient for demonstrating absence of attacks in parallel sessions.

In [27], Paulson models a protocol in presence of an intruder as an inductively defined sets of traces, and uses Isabelle and 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 to Basin's method [3], 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. Trace semantics were also used for model-checking based analysis. In this context, we find the work of Lowe [17, 18], who first identified and fixed the flaws in the Needham-Schroeder protocol, and the work of Mitchell et al. [24]. As pointed out e.g., in [16], 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 rewriting-based approach of [7, 8] and the declarative way of specifying protocols using logic programs taken in [2, 12]. 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 [16] by processing the rewrite rules in the theorem-prover daTac. Conceptually, our system shares some features with that developed (independently) by Chevalier and Vigneron [11, 10]. As in [11, 10], we deal with communication in a constraint-based way, in which the intruder only checks if he is able to generate a certain message. Differently from the previous approaches, we separate in clear way knowledge and state and, thus we reason about knowledge and specify every principal using very simple *Horn programs*. In our opinion, this makes our framework more suitable for a specification tool.

Recently, a few works have appeared that employ constraint-solving techniques, possibly in combination with traces [15, 22, 30]. Typically, the task of the intruder is brought down to that of checking whether certain messages are *contrivable*. In a way, these work have something in common with our approach in that the transformation we performed by combining the receive and the forge rule aims at reducing the task of the intruder to *check* whether he is able to generate a certain message. Differently from us, [15, 22, 30] do this in a more radical

way, and the checking phase is postponed *after* that the whole trace has been completed (while we check it after the next protocol step has been carried out). An advantage of our approach is that it permits to define a prototype for the simulation and fast prototyping of security protocols. In fact, our model allows to simulate all traces of a protocol, including thus those leading to failure. This is of course crucial when debugging a protocol. Because of their symbolic nature, this would not be possible using the models of [15, 22, 30].

Logic programming languages have been applied to specify security protocols in several different ways. Meadow's NRL Protocol Analyser [21] performs a reachability analysis using state-enumeration enriched by lemmata 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 in a backward fashion. Secondly, using proof theory we can formally reason on class of proofs and protocols for which finite-state exploration is both sound and complete. Furthermore, Prolog is used in our approach to *declaratively* specify each principal. In [2], Aiello and Massacci use a logic programming language but using a different perspective, i.e., with *stable semantics*, to specify and debug protocols. In this setting knowledge, protocol rules, intruder capabilities and objectives are specified in a declarative way. Finally, Blanchet [5] 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. In [1] Abadi and Blanchet relate this approach to that of using *types* for guaranteeing the secrecy of communication, an approach substantially different from ours. The verification procedure combines aspects of *unfolding* and *bottom-up* evaluation (following, however, a depth-first strategy).

References

1. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *Proc. POPL 2002*, pages 33–44, 2002.
2. L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *Trans. on Computational Logic*, 2001.
3. D. Basin. Lazy infinite-state analysis of security protocols. Secure Networking — CQRE [Secure] '99", LNCS 1740, pages 30–42, 1999.
4. G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In *Proc.5th Symp. on Research in Computer Security*, LNCS 1485, pages 361-375, 1998.
5. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proc. CSFW'01*, 2001.
6. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. 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 *Proc. CSFW '99*, pages 55–69, 1999.
8. I. Cervesato, N. Durgin, J. Mitchell, Lincoln, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *Proc. CSFW '00*, pages 35-51, 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. Y.Chevalier and L.Vigneron. A tool for lazy verification of security protocols. In *Proc. Int'l Conf. Automated Software Engineering*, 2001.
11. Y.Chevalier and L.Vigneron. Towards efficient automated verification of security protocols. In *Proc. VERIF '01*, 2001.
12. G. Delzanno. Specifying and debugging security protocols via hereditary harrop formulas and lambda prolog. In *Proc. FLOPS '01*, pages 123–137, 2001.
13. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983.
14. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Proc. FMSP'99 (FLOC '99)*, 1999.
15. A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. FMSP '99 (FLOC '99)*, 1999.
16. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *Proc. LPAR 2000*, LNCS 1995, pages 131-160, 2000.
17. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, (17):93–102, 1996.
18. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc.CSFW '97*, pages 18–30, 1997.
19. G. Lowe. Towards a completeness result for model checking of security protocols. *J. of Computer Security*, 7(2-3):89–146, 1998.
20. C. Meadows. Formal verification of cryptographic protocols: A survey. In *Proc. ASIACRYPT '94*, pages 133–150, 1995.
21. C. Meadows. The NRL protocol analyzer: An overview. *J. of Logic Programming*, 26(2):113–131, 1996.
22. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 2001 ACM Conf. on Computer and Communication Security*, pages 166 – 175, 2001.
23. J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Trans. on Software Engineering*, 13(2):274–288, 1987.
24. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proc. Conf. on Security and Privacy*, pages 141–153, 1997.
25. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Comm. of the ACM*, 21(12):993–999, 1978.
26. L. C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. T.R. 413, Univ. of Cambridge, Computer Laboratory, January 1997.
27. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. of Computer Security*, 6:85–128, 1998.
28. A. W. Roscoe. Modelling and verifying key-exchange protocols using csp and fdr. In *IEEE Symp. on Foundations of Secure Systems*, 1995.
29. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.
30. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In *Proc. CSFW '01*, 2001.