A Proof Theoretic Analysis of 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 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 proof 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 for checking safety properties of security protocols, and we have applied it successfully to find error traces or proving correctness of practical examples. The specification of a protocol is done in Prolog, and does not require knowledge of the underlying proof system.

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. This gave impulse to research on the formal verification of security protocols. As explained in detail in Meadows [13], most of this research is based on modal logic and state machines. The best-known approach based on modal logic is the commonly called BAN-logic proposed by Burrows, Abadi and Needham in [6]. In modal logics one can express concepts such as belief and therefore model logically the evolution of knowledge in the system. On the other hand, a prominent early work on methods based on state machines is Dolev-Yao’s [10]. In such systems the protocol is translated into another executable formalism (e.g. rewrite systems) which allows its simulation in presence of an hostile intruder which is usually assumed to have complete control over the network and which is allowed to “randomly” perform actions such as intercepting communications and forging messages. By an exhaustive search, one can establish if the protocol is flawed or not. Among such systems we should mention Millen’s Interrogator [17], the NRL protocol analyzer [14], the works of [7] on multiset rewriting systems [7, 8], and, more recently, the works of Blanchet [4] and of Aiello and Massacci [1]. Clearly, a crucial problem in the development of such a protocol verifier is to limit the search space explosion that occurs when modeling “random” actions of the intruder. To this end, many solutions have been employed, ranging from human intervention to the use of approximations.

One further weakness of the “standard” way of designing security protocols lies also in the formalization process [7, 1], because of ambiguities and of the difficulties in formalizing what the actual goals of a protocol are. To this end, a number of protocol specification languages have been devised, notably CASPER [12], and CAPSL [5, 15, 16]. These languages are used as intermediate languages for performing state-base verification; however, as pointed out in [1] such verification systems lack flexibility, e.g., it is often not possible to modify the abilities of a potential intruder.

In this paper we propose a logic-based specification method which allows for reasoning over a protocol but also for implementing state-based verification in a particularly efficient and effective way. More specifically, our contribution is twofold.
We propose a proof theoretic view of authentication protocols. The proof theoretic approach is based on the observation that during the execution of a protocol stage two kind of changes take place: a change in knowledge (which is monotonic, as it is not modified during later stages) and a change of state (e.g. the presence of messages on the network) which is non-monotonic, for instance, a message in the network can be removed from it. In our approach, the knowledge is modeled by a first-order theory, and the state by a multiset. We show that 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 then to a protocol trace that from the initial state leads to the goal taken into consideration.

This allows to prove safety properties of a protocol: such properties can be encoded within the same proof system as closing rules (axioms).

In addition to this, the proof system allows to model a potential intruder using by adding few rules that model its behaviour. It is then possible to check if the intruder has a way of breaking the protocol.

Compared to other state-based methods like multi-set rewriting, the proof-system view has two great advantages. On one side it allows us to separate in a clear way knowledge and global state. Hereby, we can describe a protocol in a simple and modular way separating the specification of agents’ point-of-views from all details involving agent communication and knowledge transformation. On the other side, it allows us to apply proof theoretic tools to discover efficient strategies to explore the state space generated during a given session (i.e. to debug a protocol). Specifically, a proof-theoretic analysis we describe here allows us to define the notion of normal proof that dramatically prunes state exploration, and to isolate a large class of protocols we called fully-typed, for which there exists an effective (sound and complete) strategy for the verification of safety properties.

(2) As application of the first contribution, we show how we have employed it in the construction of a prototype for the specification and the verification of security protocols. The prototype is written in Prolog, and allows the user to specify formally a cryptographic protocol, by writing Prolog rules (i.e. Horn clauses) defining it. Furthermore, one can define the goals of a protocol and verify by simulation if they are reachable. Similarly, one can also check that unsafe situations are not reachable. Finally, the user can define the goals of a potential intruder, and therefore check the protocol for flaws.

To our knowledge, this is the first time that proof theoretic reasoning based on multi-conclusion sequent calculi has been applied to the specification and validation of security protocols. Furthermore, to demonstrate the practicality of our approach, in the paper we will briefly discuss some experimental results obtained with our prototype.

Plan of the paper. In Section 2, we describe our case-study the (classical) Needham-Schroeder public cryptography protocol. In Section 3, we present the language (formulas and sequents) used to describe security protocols in our framework. In Section 4, we present the proof rules for principal communication. In Section 5, we present the additional proof rules for modeling an intruder. In Section 6, we analyze 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 A Case-study: The Needham-Schroeder Protocol

As main example, we will consider the security protocol based on public-key cryptography proposed by Needham and Schroeder in [19]. 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
In order to validate security protocols it is fundamental to model the knowledge of each participant during the execution of the protocol, and to model the possible actions that a potential intruder can take in order to break the security of the communication.

3 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, the agent Bob will know the nonce 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 first order theory, denoted by $\Delta$, and states are modeled via multisets of atomic formulas, denote by $S$. We then define multi-conclusion sequents having the form $\Delta \rightarrow S$, 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

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

where $S$ and $S'$ are multisets, $\Delta$ and $\Delta'$ are first-order theories, $\phi$ is a first-order formula and $\vdash$ is a given provability operator. When $\Delta, \Delta' \rightarrow S'$ is absent from the premise we call it a closing rule.

Here, $\Delta, \Delta'$ denotes the set $\Delta \cup \Delta'$, while $a, b, \ldots, S$ denotes the multiset containing $a, b, \ldots$ and the elements in $S$ (i.e. on the left(right)-hand side of a sequent ‘,’ represents (multi)set union). In our system, each protocol transaction is modeled via a rule, having the previous form, in which $S$ and $\Delta$ are respectively the global state and knowledge before the transaction is fired; $\phi$ models the conditions under which the transaction can be fired; $S'$ is the state of the system after the transaction is completed; $\Delta'$ is the new knowledge, acquired during the transaction.

In the rest of the paper $\Delta$ and $\Delta'$ will be set of Horn Clauses, i.e. universally quantified implicational formulas indicated as $A \leftarrow B_1, \ldots, B_n$ ($\leftarrow$ denotes $\subset$, i.e., implication read from right to left) where the head $A$ is an atomic formula and the body $B_1, \ldots, B_n$ is a conjunction of atomic formulas. A unit clause or fact is a Horn clause without body (or having only true in the body). We allow $B_1$ to be an inequality $\tau \neq s$, other than this, negation is not involved. In the following we will use identifiers beginning with upper-case letters to denote implicitly universally quantified variables. A goal or query is a conjunction of atomic formulas. The relation $\vdash$ will denote the provability relation built on top of ground (variable-free) resolution defined by the rules in Fig. 1. Note, however, that one could use other sound methods in order to check the validity of the condition $\phi$ against the first order theory $\Delta$ without changing all results presented in the paper. A (partial) protocol execution starting from the state $S_0$ and knowledge $\Delta_0$ and

\[
\begin{align*}
\Delta & \vdash a & \text{if } a \in \text{Ground}(\Delta) \\
\Delta & \vdash b_1 \cdots \Delta \vdash b_n & \text{if } a : -b_1, \ldots, b_n \in \text{Ground}(\Delta)
\end{align*}
\]

Fig. 1. The proof rules for $\vdash$ (simplified resolution, which is sufficient for our purposes).
ending in the state $S_n$ and knowledge $\Delta_n$ is thus represented via a (partial) proof tree (here and in the sequel we omit the proof trees for $\vdash$ when possible)

$$
\begin{array}{c}
\Delta_0 \vdash \phi_0 \\
\vdash \\
\vdash \\
\vdash \\
\Delta_n \vdash \phi_n \\
\Delta_n \rightarrow S_n \\
\vdash \\
\Delta_0 \vdash \phi_0 \\
\Delta_1 \rightarrow S_1 \\
\vdash \\
\Delta_0 \rightarrow S_0
\end{array}
$$

A proof is *successful* if each premise is satisfied. In order to build successful proofs we need to add *closing rules* to the proof system. Thus, by modifying the *closing rules* of the proof system one can modify its meaning: a successful proof could be one validating a certain protocol, or just one cracking it.

Before going into the details of the definition of the inference rules for $\rightarrow$, let us fix the first order language used to specify the global knowledge $\Delta$ and the current state $S$.

### 3.1 Knowledge and State Description Language

**States** The *global state* of the principals involved in the protocol is described via a multiset containing *agent states* and/or *messages* (indicating the presence in the network of a message at a given stage).

An *Agent States* is specified the atom $agent(ID,S)$ where $ID$ is a term denoting the identifier, and $S$ is an integer denoting the current step of the protocol the agent is at.

*Messages* contain lists of objects (keys or nonces), which may in turn be encrypted. We use $enc(K,M)$ to represent a message encoded with the public key $K$ and $symenc(K,M)$ to represent a message encoded with a symmetric key $K$. More precisely, we will consider the following syntax

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

where $I$ denotes a natural number.

**Knowledge** 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, which 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 *decorations*, which serve as place-holders for facilitating the retrieval of stored values.

### 3.2 Protocol Specification

A protocol specification allows us to describe all traces leading from an initial state to a state that, according to a given safety criterion, we consider as *final*. Formally, a protocol specification is a tuple $\langle \Delta_0, S_0, \Phi_f \rangle$ consisting of the initial knowledge $\Delta_0$, the initial global state $S_0$, and a final set of global states $\Phi_f$. In turn, the initial knowledge $\Delta_0$ can be seen as the union of $\Delta_{rules}$ (the rules that specify the protocol) and $\Delta_{knowledge}$ (the agents’ initial knowledge).

**The Protocol 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 *synchronization* of the agents can be left as part of the *operational semantics* defined via our proof system. In our definition of a protocol, $\Delta_{rules}$ is the logic program that defines $expect$ and $compose$. The predicate $expect$ has the following fixed signature:
\[ \Delta \vdash \text{compose}(id, \text{step}, m, n, K) \quad \Delta, \Delta' \rightarrow \text{agent}(id, \text{step} + 1), \text{msg}(m), S \quad \text{compose} \]

\[ \Delta \rightarrow \text{agent}(id, \text{step}), S \quad \text{provided } n \text{ does not occur in } \Delta \rightarrow \text{agent}(id, \text{step}), S \text{ and } \Delta' = \{ \text{knows}(id, k) \mid k \in K \} \]

\[ \Delta \vdash \text{expect}(id, \text{step}, m, K) \quad \Delta, \Delta' \rightarrow \text{agent}(id, \text{step} + 1), S \quad \text{expect} \]

\[ \Delta \rightarrow \text{agent}(id, \text{step}), \text{msg}(m), S \quad \text{provided } \Delta' = \{ \text{knows}(id, k) \mid k \in K \} \]

\[ \Delta \vdash P \quad S \in \Phi_f \quad \text{final} \]

**Fig. 2.** The Compound Proof System for the protocol specification \((\Delta_0, S_0, \Phi_f)\), and the property P.

\[ \text{expect}(ID, \text{Step}, \text{Message}, \text{Knowledge}), \]

where \(ID\) is an agent identifier, \(\text{Step}\) is a protocol step, \(\text{Message}\) is a message, and \(\text{Knowledge}\) is a list of decorated terms. For the moment one can think of the first three argument as being *input arguments*, while the last one is an *output* one. The query \(\text{expect}(ID, \text{Step}, \text{Message}, \text{Knowledge})\) should succeed in the current global knowledge whenever agent \(ID\) at step \(\text{Step}\) can receive message \(\text{Message}\). \(\text{Knowledge}\) is the list of facts the agent learns during the transaction. As an example, Bob’s role in the first transaction is specified by the rule:

\[ \text{expect}(bob, 1, \text{msg}(\text{enc}(\text{key}(Pkb), [\text{key}(Pka), \text{nonce}(Na)])), \text{Info}) : - \text{knows}(bob, \text{keypar}(_., Pkb)), \text{knows}(bob, \text{key}(Pka)), \]

\[ \text{Info} = [\text{other} \_ \text{nonce}(Na), \text{other} \_ \text{key}(Pka)]. \]

Here \(\text{Info}\) is used to memorize Bob received nonce \(Na\) from the agent having public key \(Pka\).

The predicate \(\text{compose}\) is used to compose new messages. Its signature is:

\[ \text{compose}(ID, \text{Step}, \text{Nonce}, \text{Message}, \text{Knowledge}) \]

where \(ID\) is an agent identifier, \(\text{Step}\) is a protocol step, \(\text{Nonce}\) is a fresh name, \(\text{Message}\) is a message, and \(\text{Knowledge}\) is a list of decorated terms. Again, one can think of the first three argument as being *input arguments*, while the last two are *output*. The extra input argument \(\text{Nonce}\) is used for passing a fresh nonce to the rule, in case that it is needed for composing a message. The query \(\text{compose}(ID, \text{Step}, \text{Nonce}, \text{Message}, \text{Knowledge})\) succeeds when agent \(ID\) at step \(\text{Step}\) can produce message \(\text{Message}\), possibly using the nonce \(\text{Nonce}\), and \(\text{Knowledge}\) is the list of facts the agent learns during the transaction.

Each protocol transaction is fully specified by declaring one \(\text{expect}\) and one \(\text{compose}\) clause. Appendix A.1 reports the commented specification for the Needham-Schroeder protocol.

**The Initial Knowledge.** It describes the initial knowledge of each via a set of unit clauses. For Needham-Schroeder, it consists of the set of atoms: \(\{ \text{knows}(alice, \text{keypar}(ska, pka)), \text{knows}(\_., pka)\}, \text{knows}(bob, \text{keypar}(skb, pkb)), \text{knows}(\_., pkb)\} \) where the underscore \(\_\) is the anonymous variable: \(\text{knows}(\_., pka)\). indicates that every agent knows the public key of Alice.

**Initial Global State.** \(S_0\) represent the multiset of agents in their initial state. For Needham-Schroeder the initial global state is \(S_0 = \text{agent}(alice, 1), \text{agent}(bob, 1)\).

**Final Global State.** The last component \(\Phi_f\) is the (possibly infinite) set of final global states of a session. As an example, the final global state of our main example can be specified as *any* global state containing the multiset \(S_f = \text{agent}(bob, 4), \text{agent}(alice, 4)\).

### 4 Proof Rules

The behaviour of the principals involved in a protocol can be described using the interleaving operational semantics described in this section. We start with the following central definition.
\[
\Delta, \Gamma \Delta \longrightarrow S \quad \text{intercept}
\]
\[
\Delta \longrightarrow msg(m), S
\]

\[
\Delta \vdash \text{contrive}(m) \quad \Delta \longrightarrow \text{msg}(m), \Phi
\]

\[
\Delta \longrightarrow \Phi
\]

where \( T(\Delta, m) = \{ \text{iknows}(n) \mid \Delta \vdash \text{compose}(m, n) \} \)

**Fig. 3.** The proof rules relative to the intruder.

**Definition 2.** The compound proof system derived from a protocol specification \( \langle \Delta_0, S_0, \Phi_f \rangle \), and from a safety property \( \mathbb{P} \), consists of the three rules of Figure 2.

The `expect` and `compose` rules model the actual behaviour of the protocol. Notice that these rules depend from the predicates `expect` and `compose` which are defined in \( \Delta \). The `compose` rule requires that \( n \) does not occur in the lower sequent, indicating thus a fresh nonce. Actually, the above rule applies when `compose` needs at most one fresh nonce. Rule for where more nonces are needed are obtained via a straightforward generalization.

**Final Rule and the Property \( \mathbb{P} \)** In this paper we focus our attention on safety properties formulated over the knowledge accumulated by the agents during a session. Intuitively, our goal is to answer questions like: *is it possible to reach a final state in which Alice and Bob know the nonces \( N_a \) and \( N_b \)?* For this purpose, we assume that a given safety property is specified as a formula \( \mathbb{P} \) used to query the final knowledge reached by the agents. For instance, suppose we want to check that Alice and Bob can actually reach the end of the protocol while exchanging each others nonces, then, the final knowledge we want to observe can be described via the following program (note: added to the global knowledge)

\[
\text{final knowledge} \vdash \begin{array}{l}
\text{knows(bob,other_nonce(N_a))}, \text{knows(alice,other_nonce(N_b))}, \\
\text{knows(bob,my_nonce(N_b))}, \text{knows(alice,my_nonce(N_a))}.
\end{array}
\]

Based on this definition, \( \mathbb{P} \) is simply the goal `final knowledge`.

**Proofs as Traces** Our 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 \( \Delta_0 \longrightarrow S_0, \ldots, \Delta_k \longrightarrow S_k \) where \( \Delta_i \longrightarrow S_i \) and \( \Delta_{i+1} \longrightarrow S_{i+1} \) correspond respectively to the lower and upper sequent of an instance of one of the proof rules `expect`, and `compose`, in which all auxiliary sequents of the form \( \Phi \vdash \phi \) are provable for each \( 0 \leq i < k \). Finally, \( \Delta_k \longrightarrow S_k \) is the lower sequent of an instance of the final rule in which all premises are satisfied. The following property then holds.

**Proposition 1 (Soundness).** Given a protocol specification \( \langle \Delta_0, S_0, \Phi_f \rangle \), and a safety property \( \mathbb{P} \), every proof for the sequent \( \Delta_0 \longrightarrow S_0 \) corresponds to a trace that from the initial state of the protocol leads to a final configuration \( \Delta \longrightarrow S \) where \( S \in \Phi_f \) and \( \Delta \) satisfies the final condition specified in \( \mathbb{P} \).

Note that, all formulas occurring in a proof are ground, i.e., in order to apply a rule we first need to find a ground instance that matches the lower sequent, instantiate the premises so as to satisfy all side conditions, and so on.

## 5 Modeling an Intruder

An important application of our system is that of allowing to test the protocol specification against possibly malicious intruders. Following the literature, we assume the existence of a new agent, called Trudy, which has complete control over the network: Trudy can intercept and eavesdrop all messages that are sent. In addition to this, Trudy can `contrive` messages, by using the information she possesses or by generating new nonces and messages. 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. If this is possible, we can say that the protocol under exam is crackable. Our system can be used for checking if such a malicious intruder is actually able to fulfill its goal. This is achieved by providing fixed rules for specifying Trudy’s behaviour, reported
in Figure 3. To distinguish it from the “legal” knowledge, we introduce a new predicate \textit{iknows} that models the current knowledge of Trudy. In the rule for intercepting a message, the set of facts \( T(\Delta, m) \) represents the closure of the knowledge of the intruder with respect to what Trudy can decipher using the predicate \textit{decompose}. Following from this definition, given a message \( m \) (i.e., a ground term), \( T(\Delta, m) \) always consists of a finite set of facts. Nondeterministically, Trudy can create a message in order to cheat the other principals. Starting from her current knowledge, Trudy uses the predicate \textit{contrive} to generate a random message of arbitrary size. This message is placed in the global state via the \textit{contrive} rule. Appendix A.2 reports the definitions of \textit{define} and \textit{compose}.

**Intruder’s initial knowledge** Clearly, Trudy has to possess at least a public-private key pair, and has to be able to generate \textit{a priori} random numbers with which it contrives messages (if the knowledge of Trudy is empty, Trudy is not able to contrive anything).

**Definition 3.** An intruder’s specification is a set of clauses containing the rules for \textit{decompose} and \textit{contrive} (which are invariant) together with a set of unit facts each one storing a new nonce, i.e., \textit{iknows(}nonce\( (n) \text{) where} n \text{ is a fresh integer, or a pair of keys, i.e.,} \textit{iknows(}keypar\( (sk, pk) \text{) where} sk, pk \text{ are fresh integers.}

In the sequel, if no confusion arises we shall often say “intruder” instead of “intruder specification”. We can now analyze the protocol traces in presence of Trudy.

**Definition 4.** The extended compound proof system derived from a protocol specification \( \langle \Delta_0, S_0, \Phi_f \rangle \), a safety property \( \mathbb{P} \), combined with an intruder specification \( \Delta_{trudy} \) consists of the rules of Figure 3 together with the rules of Figure 2.

Here we assume that \( \Delta_{trudy} \) contains also the definitions for \textit{contrive}, and \textit{decompose} (which are invariant), and that the formula \( \mathbb{P} \) can also query Trudy’s knowledge. Then, the proof system enriched with the intruder rules allows us to formally describe all possible \textit{traces} of an interleaving execution of the principals and of Trudy as stated in the following theorem.

**Proposition 2 (Soundness).** Given a protocol specification \( \langle \Delta_0, S_0, \Phi_f \rangle \), a property \( \mathbb{P} \), every proof for the sequent \( \Delta_0, \Delta_{trudy} \rightarrow S_0 \) corresponds to a trace of an interleaved execution of the principals and an intruder \( \Delta_{trudy} \) that from \( S_0 \) leads to a final state in \( \Phi_f \) that satisfies the final condition specified in \( \mathbb{P} \).

## 6 Analysis and Tuning of the Proof System

By the definition of the intruder theory, even if the protocol specification describes a finite number of steps for the involved principals, there might be traces of arbitrary length (when Trudy generates an arbitrary number of messages) and there might be infinitely many different traces of a given length (e.g., at any step Trudy can generate a message of arbitrary depth). 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 (very large) class of protocol specification for which the search space is finite. For these protocols we have a soundness and completeness result.

**Limiting the Intruder’s knowledge** First, we have to resolve the problem of the initial knowledge of the intruder. In fact Proposition 2 depends from a specific intruder \( \Delta_{trudy} \), which can be arbitrarily large. The following result shows that it is possible to build a (small) \( \Delta_{trudy} \) which suffices.

**Theorem 1.** Given a protocol specification \( \langle \Delta_0, S_0, \Phi_f \rangle \), and a property \( \mathbb{P} \) then there exists an intruder \( \Gamma_{\Delta_0, \mathbb{P}} \) such that for any other intruder \( \Delta_{trudy} \): If there exists a proof for \( \Delta_0, \Delta_{trudy} \rightarrow S_0 \) then there exists a proof for \( \Delta_0, \Gamma_{\Delta_0, \mathbb{P}} \rightarrow S_0 \).

**Proof.** ¹ We give here just the intuition. The number of nonces and key pairs the intruder “needs” is equal to the number of inequalities present in \( \Delta_0, \mathbb{P} \), plus one. In fact assume there is a proof \( \pi \) of \( \Delta_0 \cup \Delta_{trudy} \rightarrow S_0 \), where \( \vdash \) is defines as in figure 1; and that two of the intruder’s nonces, \( n_1 \) and \( n_2 \) occur in \( \pi \), if \( \Delta_0 \cup \mathbb{P} \) contains no inequalities then one could replace each instance of \( n_2 \) with \( n_1 \) in the proof, obtaining another valid proof. The general case follows by induction.

¹ This holds if comparison between terms is done by means of equalities and inequalities. If other comparisons are allowed (e.g., \( t < s \)) then the Theorem does not apply any longer as such.
Properties of Proof Rules  Secondly, we have to avoid possibly infinite branches. Actually, to achieve practicality, we need to minimize the occurrences of the contrive rule in the proofs.

We start by noting that we can move and remove some occurrence of the contrive rule, preserving equivalence. Our proof system enjoys, in fact, the following permutability properties.

1. The rule contrive always permutes up with contrive and compose: e.g. by permuting two adjacent compose and contrive rule, one obtains an equivalent proof.
2. The rule contrive permutes up with intercept and expect whenever the contrived message is different from the intercepted or expected message.
3. The rule final can be permuted down with contrive if we assume that the messages are not essential in the definition of the set of final states Φf, i.e., for any m, if msg(m), S is a final state than S ∈ Φf is final, too.

After permuting up as far as possible all contrive rules, we obtain a proof system in which each contrive rule is always followed by an except or an intercept rule concerning the same message. In this second case we can prune both rules, in fact: The rule contrive 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. Without getting into the details, the following lemma is the central result which allows for the pruning.

Lemma 1. Let Δ ⊬ contrive(m) hold, then for all n and k the following properties hold: (i) Δ ⊬ contrive(n) if and only if Δ, T(Δ, m) ⊬ contrive(n); (ii) Δ ⊬ decompose(n, k) if and only if Δ, T(Δ, m) ⊬ decompose(n, k).

Proof Strategies and Derived Rules  The above properties demonstrate that we can restrict to proofs in which each contrive rule is followed by (i.e. is just under) an expect rule, which reads the message generated by contrive. Since the description of a protocol is finite, there might be only a finite number

of expect rules in a proof. Thus, if we disregard the steps needed to prove ⊬, such proofs are of bounded depth; in practice we can now restrict to bounded traces. Still, the contrive-rule can determine a high level of non-determinism in the proof construction process. Declaratively, Δ ⊬ 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.

\[
\begin{align*}
\Delta \vdash \text{contrive}(m) & \quad \Delta \vdash \text{expect}(id, m, s, K) \\
\Delta, \Delta' \rightarrow agent(id, s + 1), S
\end{align*}
\]

\[
\begin{align*}
\text{contrive} & \quad \Delta \rightarrow msg(m), agent(id, s), S \\
\text{expect} & \quad \Delta \rightarrow agent(id, s), S
\end{align*}
\]

where \( \Delta' = \{ \text{knows}(id, k) \mid k \in K \} \). This can be transformed into the following contrive-expect rule.

\[
\begin{align*}
\Delta \vdash \text{contrive}(m) \land \text{expect}(id, s, m, K) & \quad \Delta, \Delta' \rightarrow agent(id, s + 1), S \\
\text{contrive-expect} & \quad \Delta \rightarrow agent(id, s), S
\end{align*}
\]

The following property summarizes what we achieved so far.

Theorem 2 (Proof Normalization). Given a protocol \( \langle \Delta_0, S_0, \Phi_f \rangle \), and a property \( \mathbb{P} \), any proof π for the sequent \( \Delta_0, \Gamma_{\Delta_0, \mathbb{P}} \rightarrow S_0 \) can be transformed into a normal proof π’ that makes use of the derived rule contrive-expect and in which there are no occurrences of contrive.

6.1 Completeness of Normal Proofs for Fully-typed Protocols

The derived rule contrive-expect still contains a lot of non-determinism that 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 non-determinism of contrive. The crucial point here is that if expect(id,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 5 (Fully-Typed Protocol). A protocol \( \langle \Delta_0, S_0, \Phi_f \rangle \) is fully-typed if for any agent knowledge \( \Delta \), if \( \Delta_0, \Delta \vdash \text{expect}(id, s, m_1, i_1) \) and if \( \Delta_0, \Delta \vdash \text{expect}(id, s, m_2, i_2) \), then \( m_1 \) is isomorphic to \( m_2 \), when viewed as trees (their term structure).

This condition implies that, 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 specified as explained in Section 3 is fully-typed.

Fully-typed protocols allow for an effective use of the \text{contrive-expect} rule by a guided generation of messages. Intuitively, the idea is to interleave the Prolog execution of \text{expect} and \text{contrive}: after extracting the pattern of an expected message using \text{expect}(id, s, m, i) \text{ contrive} simply has to fill all remaining holes using new nonces or keys and nonces Trudy has stored in previous steps. This renders finite the search space, and leads us to the following completeness result.

Theorem 3 (Completeness of Normal Proofs). Given a protocol \( \langle \Delta_0, S_0, \Phi_f \rangle \). There exists a method for checking \( \vdash \) such that the class of normal proofs of the sequent \( \Delta_0, \Gamma \vdash \Pi \rightarrow S_0 \) is a complete test for safety properties specified via \( \Pi \) and \( \Phi_f \). Furthermore, only a finite number of normal proofs are needed to verify a property.

Interestingly, the specification of Needham-Schroeder given in appendix (e.g. in accord to [2]) is fully-typed: our proof strategy can thus be used for debugging this protocol as well as for the verification of its safety properties.

7 Conclusions

Based on the results given in the previous section, we have built an interpreter that generates normal proofs only. Our prototype is a complete tool for safety properties of fully-typed security protocols. As implementation language we have chosen Prolog, thus committing to an SLDNF strategy to check provability of \( \vdash \)-sequents. The soundness of the strategy based on normal proofs ensures that the prototype can be used as a debugging tool for all protocols that are not fully-typed: if an error-trace is found, then the protocol is not safe.

Trace-based interleaving semantics is at the basis of the verification methods proposed, e.g., in [20, 21, 3, 2]. In [21], 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. This representation allows us to reason about ‘permutability’ of rules and thus to isolate classes of proofs and protocols for which we can automatically verify the correctness (Needham-Schroeder being one of this case-studies). Our approach is closer 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 preliminary analysis of the proof system: in this way we only generate proofs without useless steps of the intruder. Inspired by [2], as future work we plan to investigate in the use of lazy evaluation procedures for \text{contrive} goals in order delay the generation of Trudy’s messages as much as possible.

Among other remarkable approaches based on state-machines, in [11] Lowe found subtle attacks to Needham-Schroeder using the model checker FDR. Model checking however works only for a finite state space and thus cannot be used to verify the correctness of a protocol. Meadow’s NRL Protocol Analyzer [14] (implemented in Prolog) performs instead 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 the previous works in that: using proof theory we can formally reason on class of proofs and protocols for which finite-state exploration is both sound and complete; Prolog is used to declaratively specify each principal behaviour.

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, 9]. In [7], Cervesato et al. use a formalism based on multiset-rewriting (linear logic) to specify protocol rules and actions of intruders. As mentioned in the introduction, differently from [7] our sequent-calculi specification of proof rules allows us to separate in clear way knowledge and state so the specification can be given in a modular way. Our approach is inspired in fact to the (general purpose) multi-conclusion specification language introduced by Miller in
[18], that represents a generalization of the logic underlying the framework of [7]. 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.

**An attack to the Needham-Schroeder Protocol** To conclude, we report one of the many attacks to the Needham-Schroeder protocol our system finds when `final_knowledge` is specified by the rule

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

This rule allows to find traces in which after the protocol execution, Alice and Bob knows each other’s nonces, but Trudy knows at least one of the nonces as well. The verbatim output of the prototype is:

```prolog
[alice, "sends", msg(enc(key(trudypubkey), [key(alicepubkey), nonce(1355745039)])),
[trudy, "incercepts", msg(enc(key(trudypubkey), [key(alicepubkey), nonce(1355745039)]))],
[trudy, "contrives", msg(enc(key(bobpubkey), [key(alicepubkey), nonce(1355745039)])),
[bob, "receives", msg(enc(key(bobpubkey), [key(alicepubkey), nonce(1355745039)]))],
[bob, "sends", msg(enc(key(bobpubkey), [key(alicepubkey), nonce(1355745039), nonce(809766603)]))],
[alice, "receives", msg(enc(key(alicepubkey), [nonce(1355745039), nonce(809766603)]))],
[alice, "sends", msg(enc(key(trudypubkey), [nonce(809766603)]))],
[trudy, "incercepts", msg(enc(key(trudypubkey), [nonce(809766603)]))],
[trudy, "contrives", msg(enc(key(bobpubkey), [nonce(809766603)]))],
[bob, "receives", msg(enc(key(bobpubkey), [nonce(809766603)]))]]
```

**References**

A Appendix: Commented Source Code

Note In this appendix, we report for the referee's convenience, part of the source code of the prototype for the specification and verification of the Needham Schroeder Protocol. Should this submission be accepted and unless explicitly requested by the referee, this appendix will not be part of the final paper. Instead, this part will be made available electronically.

A.1 Needham-Schroeder Protocol.

Protocol Rules.

% First transaction A -> B:{A,Na}Kb

expect(bob,1,M,Info) :-
    M = msg(enc(key(Pkb),[key(Pka),nonce(Na)])), % checks that the message has the right format
    knows(bob,keypar(_,Pkb)), % checks that it is encoded using bob's key
    knows(bob,key(Pka)), % checks that Pka is the key of an agent bob knows
    Info = [other_nonce(Na),other_key(Pka)]. % bob received Na from the agent having key Pka

compose(alice,1,Nonce,M,Info) :-
    knows(alice,keypar(_,Pka)), % fetch alice own public key
    knows(alice,key(Pkb)), % fetch bob's public key
    M = msg(enc(key(Pkb),[key(Pka),nonce(Nonce)])), % produce the message, using Nonce
    Info = [my_nonce(Nonce),other_key(Pkb)]. % alice has sent a Nonce encrypted via Pkb

% Second transaction B -> A:{NaNb}Pka

compose(bob,2,Nb,msg(enc(key(Pka),[nonce(Na),nonce(Nb)]),[my_nonce(Nb)])) :-
    knows(bob,other_nonce(Na)),
    knows(bob,other_key(Pka)).

expect(alice,2,msg(enc(key(Pka),[nonce(Na),nonce(Nb)]),[other_nonce(Nb)])) :-
    knows(alice,keypar(_,Pka)),
    knows(alice,my_nonce(Na)).

% Third Transaction A -> B:{Nb}Kb.

expect(bob,3,msg(enc(key(Pkb),[nonce(Nb)]),[])) :-
    knows(bob,keypar(_,Pkb)),
    knows(bob,my_nonce(Nb)).

compose(alice,3,_,msg(enc(key(Pkb),[nonce(Nb)]),[])) :-
    knows(alice,other_nonce(Nb)),
    knows(alice,other_key(Pkb)).

Initial knowledge and state.

$S_0$ is the multiset $agent(alice,1), agent(bob,1)$; $\Delta_0$ is the set of unit facts:

knows(alice,keypar(ska,pka)). knows(bob,keypar(skb,pkb).
knows(_,key(pkb)). knows(_,key(pka)).
Final states Any state that contains the multiset \texttt{agent(alice,4), agent(bob,4)} is a final state.

A.2 Intruder Theory

The predicate \texttt{in\_knows\_objects} is used to swap keys and nonces (free variables are indicated as H, N, \texttt{Info}, \texttt{Otherkey}, etc.; whereas constants are indicated with lower-case letters, e.g., \texttt{alice}, \texttt{bob}, etc.).

\begin{verbatim}
iknows\_object(N):- iknows\(\texttt{object}(N))\).

iknows\_object(N):- iknows\(\texttt{keypar}(N,\_))\).

iknows\_object(N):- iknows\(\texttt{keypar}(\_,N))\).
\end{verbatim}

The predicate \texttt{decompose} is defined as follows.

\begin{verbatim}
decompose([],[])\).
decompose([H|T],Info):- decompose(H,Info).
decompose([H|T],Info):- decompose(T,Info).
decompose(nonce(N),[\texttt{object}(N)])\).
de(decompose(key(N),[\texttt{object}(N)]))\).
de(decompose(M,\texttt{msg}(N)))\).
\end{verbatim}

For public-key cryptography, we also need the following rules.

\begin{verbatim}
de\(\texttt{decompose}(\texttt{enc}(\texttt{key}(K),M),\texttt{Info}):- \texttt{iknows}(\texttt{keypar}(\texttt{Otherkey},K)),\texttt{decompose}(\texttt{Id},\texttt{Step},M,\texttt{Info})\).
de\(\texttt{decompose}(\texttt{enc}(\texttt{key}(K),M),\texttt{Info}):- \texttt{iknows}(\texttt{keypar}(K,\texttt{Otherkey})),\texttt{decompose}(\texttt{Id},\texttt{Step},M,\texttt{Info})\).
\end{verbatim}

That is, Trudy can decipher all messages encrypted with her keys. Moreover, the following rules apply when the message is encrypted with someone else’s key that Trudy somehow got hold of.

\begin{verbatim}
de\(\texttt{decompose}(\texttt{enc}(\texttt{key}(K),M),\texttt{Info}):- \texttt{knows}(\_,\texttt{keypar}(\texttt{Otherkey},K)),\%fetch the key that is needed \texttt{iknows\_object}(\texttt{Otherkey}),\%check if trudy accidentally knows it \texttt{decompose}(\texttt{Id},\texttt{Step},M,\texttt{Info})\).
\end{verbatim}

Finally, the predicate \texttt{contrive} is defined as follows.

\begin{verbatim}
contrive\(\texttt{nonce}(M)):- \texttt{iknows\_object}(N)\).
contrive\(\texttt{key}(M)) :- \texttt{iknows\_object}(N)\).
contrive([H,H2|T]):- contrive\(\texttt{H}),\texttt{contrive}([H2|T])\).
contrive([H]) :- contrive\(H)\).
contrive\(\texttt{enc}(\texttt{key}(K),M1)):- \texttt{iknows\_object}(K),\texttt{contrive}(M1)\).
contrive\(\texttt{enc}(\texttt{key}(K),M1)):- \texttt{iknows\_msg}(\texttt{enc}(\texttt{key}(K),M1))\).
\end{verbatim}