Automated Protocol Verification in Linear Logic

Marco Bozzano
ITC - IRST
Via Sommarive 18
Povo, 38050 Trento, Italy
Tel. ++39-0461-314481
bozzano@irst.itc.it

Giorgio Delzanno
D.I.S.I., Università di Genova
Via Dodecaneso 35
16146 Genova - Italy
Tel. ++39-010-3536638
Fax: ++39-010-3536699
giorgio@disi.unige.it

ABSTRACT
In this paper we investigate the applicability of a bottom-up evaluation strategy for a first order fragment of linear logic \cite{20} for the purposes of automated validation of authentication protocols. Following \cite{11}, we use multi-conclusion formulas to represent the behavior of agents in a protocol session, and we adopt the Dolev-Yao intruder model and related messages and cryptographic assumptions. Also, we use universal quantification to provide a logical and clean way to express creation of nonces. Our approach is well suited to verify properties which can be specified by means of minimality conditions. Unlike traditional approaches based on model-checking, we can reason about parametric, infinite-state systems, thus we do not pose any limitation on the number of parallel runs of a given protocol. Furthermore, our approach can be used both to find attacks and to prove correctness of protocols. We present some preliminary experiments which we have carried out using the above approach. In particular, we analyze the ffg protocol introduced by Millen \cite{30}. This protocol is a challenging case study in that it is free from sequential attacks, whereas it suffers from parallel attacks that occur only when at least two sessions are run in parallel.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Model checking; D.3.2 [Software Engineering]: Language Classifications—Constraint and logic languages; D.4.6 [Software Engineering]: Security and Protection—Authentication

General Terms
Security, Languages, Verification

Keywords
Linear Logic, Bottom-up Evaluation, Model Checking, Authentication Protocols

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP’02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 ACM 1-58113-528-9/02/0010 ...$5.00.

1. INTRODUCTION
Linear logic \cite{20} provides a logical characterization of concepts and mechanisms peculiar to concurrency like locality, recursion, non-determinism in the definition of a process \cite{3, 22}, and synchronization. Following the paradigm of proofs as computations proposed in \cite{2, 28}, provability in fragments of linear logic can be used then as a formal tool to reason about behavioural aspects of concurrent systems (see \textit{e.g.} \cite{31}). In other paradigms for concurrency like the theory of Petri Nets there exist however a number of consolidated algorithmic techniques for the validation of system properties. In \cite{5, 6}, we made a first attempt of relating these techniques with propositional fragments of linear logic, and, more precisely, with the linear logic programming language called LO \cite{3}. LO was originally introduced as a theoretical foundation for extensions of logic programming languages. The appealing feature of this fragment, however, is that it can also be viewed as a rich specification language for protocols and concurrent systems. In fact, specification languages like Petri Nets and multiset rewriting can be naturally embedded into propositional LO\cite{12}. In \cite{5}, we established a connection between provability in LO and reachability of Petri Nets via the definition of an effective procedure to compute the set of linear logic goals (multisets of atomic formulas) that are consequences of a given propositional program. In other words we defined a bottom-up evaluation procedure for propositional programs. Our construction is based on the backward reachability algorithm of \cite{1} used to decide the so called control state reachability problem of Petri Nets. The algorithm presented in \cite{5} is defined, however, for the more general case of propositional LO specifications (i.e. with nested conjunctive and disjunctive goals).

In our setting, a natural way of augmenting the expressivity of the specification language is to consider first order fragments of linear logic. First order formulas can be used, in fact, to color the internal state of processes with structured data \cite{3, 28}. The combination between first order formulas and linear connectives provides a well-founded interpretation of the dynamics in the evolution of the internal state of a process \cite{3, 27, 28}. First order quantification in goal formulas has several interesting interpretations here; it can be viewed either as a sort of hiding operator in the style of π-calculus \cite{27}, or as a mechanism to generate fresh names as in \cite{11}.

\footnote{According to the usual terminology in logic programming, bottom-up evaluation is intended to denote derivation of logical consequences of a program, starting from the axioms.}
In [10, 7] we defined a procedure for the bottom-up evaluation of first order LO programs with universally quantified goals. Via the connection between provability and reachability established in [5], we can view such an evaluation procedure as a validation technique for colored specifications. The bottom-up evaluation procedure is based on an effective fixpoint operator and on a symbolic and finite representation of a potentially infinite collection of first-order provable LO goals (multisets of atoms). The use of this symbolic representation is crucial when trying to prove properties of parameterized systems, i.e., systems in which the number of individual processes is left as a parameter of the specification like for multi-agent protocols with multiple parallel sessions.

In this paper we investigate the applicability of the bottom-up evaluation strategy of [7] for the purposes of automated validation of authentication protocols. The design and implementation of cryptographic protocols are difficult and error prone. Authentication protocols should be reliable enough to be used in a potentially compromised environment. While cryptographic primitives are a common means to achieve these goals, they are not sufficient to ensure authentication. Exchanging nonces, i.e. fresh values, is a commonly used technique which is exploited in combination with cryptography to achieve authentication. Different approaches have been followed to specify and analyze protocols. An incomplete list include for instance using belief logics [4], rewriting techniques [17, 11, 14], theorem proving [33], logic programming [26, 16, 8], and model-checking [23, 25, 34].

Following [11], as specification language we will use multi-conclusion clauses to represent a given set of agents (called principals) executing parallel protocol sessions by exchanging messages over a network. We will use the Dolev-Yao intruder model and related message and cryptographic assumptions. Also, enriching linear logic specifications with universal quantification in goal formulas will provide a logical and clean way to express creation of new values like nonces.

In order to reason about security properties, we will apply our general purpose bottom-up evaluation scheme for first order linear logic. Our approach is well suited to verify properties which can be specified by means of minimality conditions (e.g., a given state is unsafe if there are at least two principals which have completed the execution of a protocol and a given shared secret has been unintentionally disclosed to a third malicious agent). The resulting verification method has connections both with (symbolic) model checking [1] and with theorem proving [2]. Unlike traditional approaches based on model-checking, we can reason about parametric, infinite-state systems, thus we do not pose any limitation on the number of parallel runs of a given protocol (we also allow a principal to take part into different sessions at the same time, possibly with different roles).

We have built a prototype, written in standard ML, to implement the bottom up evaluation of LO programs. We present some preliminary experiments which we have carried out using the above approach. In particular, we will focus our attention on the validation of the ffgg protocol introduced by Millen [30]. This protocol is a challenging case study for the following reasons. First, the protocol is free from sequential attacks, whereas it suffers from parallel attacks that occur only when at least two sessions are run in parallel. Secondly, the scheme underlying ffgg can be generalized so as to obtain higher order attacks (i.e. attacks that need at least k sessions in parallel). Since with our bottom-up evaluation scheme we do not need to put a bound on the number of parallel sessions, the application of our method is sound for any instance of the protocol. In this paper we will discuss the experiments obtained on the original formulation presented in [30]. Furthermore, we will present experimental results obtained for a corrected version of Needham-Schroeder, and for the Otway-Rees protocol [15]. Although much work remains to be done, experiments show that our methodology can be effective to analyze interesting aspects of authentication such as secrecy or confidentiality.

Structure of the paper. The rest of this paper is structured as follows. In Section 2 we introduce the language LO with universally quantified goals, and in Section 3 we brieﬂy discuss the bottom-up evaluation scheme for this language. In Section 4 we explain how authentication protocols can be encoded in linear logic and we present our case-study, namely Millen’s ffgg protocol. In Section 5 we discuss the application of our bottom-up evaluation algorithm for the veriﬁcation of security properties of authentication protocols, and we show some experimental results. Finally, in Section 6 we discuss related work and draw some conclusions.

2. THE FRAGMENT LO.

LO [3] is a logic programming language based on a fragment of LinLog [2]. Its mathematical foundations lie on a proof-theoretical presentation of a fragment of linear logic defined over the linear connectives \( \odot \) (linear implication), we use the reversed notation \( H \vdash G \) for \( G \vdash H \), & (additive conjunction), \( \otimes \) (multiplicative disjunction), and the constant \( \bot \) (additive identity). In this section we present the proof-theoretical semantics, corresponding to the usual top-down operational semantics for traditional logic programming languages, for an extension of LO. First of all, we consider a slight extension of LO which admits the constant \( \bot \) in goals and clause heads. More importantly, we allow the universal quantifier to appear, possibly nested, in goals. This extension is inspired by multisets rewriting with universal quantification [11]. The resulting language will be called LO\(^u \) hereafter.

Following [3], we give the following definitions. Let \( \Sigma \) be a signature with predicates including a set of constant and function symbols \( P \), and a set of predicate symbols \( \mathcal{P} \), and let \( \forall \) be a denumerable set of variables. An atomic formula over \( \Sigma \) and \( \mathcal{P} \) has the form \( p(t_1, \ldots, t_n) \) (with \( n \geq 0 \)), where \( p \in \mathcal{P} \) and \( t_1, \ldots, t_n \) are (non-ground) terms in \( \Sigma^2 \). We denote the set of such atomic formulas as \( \mathcal{A}^2 \), and the set of ground (i.e., without variables) atomic formulas as \( \mathcal{A}_G \). Finally, given a formula \( F \), we denote by \( FV(F) \) the set of free variables of \( F \).

The classes of G-formulas (goal formulas), and D-formulas (multi-headed clauses) over \( \Sigma \) and \( \forall \) are defined by the following grammar:

\[
\begin{align*}
G & ::= \exists \ G \ | \ \exists V G \ | \ \forall x G \ | \ A \ | \ \bot \\
H & ::= \exists V G \ | \ A \ | \ \bot \\
D & ::= \forall (H \vdash G)
\end{align*}
\]

where \( A \) stands for an atomic formula over \( \Sigma \) and \( V \), and \( \forall (H \vdash G) \) stands for \( \forall x_1 \ldots x_n \ (H \vdash G) \), with \( \{x_1, \ldots, x_n\} = FV(H \vdash G) \).
An \( LO_v \) program over \( \Sigma \) and \( \mathcal{V} \) is a set of \( \mathcal{D} \)-formulas over \( \Sigma \) and \( \mathcal{V} \). A multiset of goal formulas will be called a context hereafter. In the following we usually omit the universal quantifier in \( \mathcal{D} \)-formulas, i.e., we consider free variables as being implicitly universally quantified.

Let \( \Sigma_r \) be a signature with predicates and \( \mathcal{V} \) a denumerable set of variables. An \( LO_v \) sequent has the form

\[
P \vdash \Sigma G_1, \ldots, G_k,
\]

where \( P \) is an \( LO_v \) program over \( \Sigma_r \) and \( \mathcal{V} \), \( G_1, \ldots, G_k \) is a context (i.e., a multiset of goals) over \( \Sigma_r \) and \( \mathcal{V} \), and \( \Sigma \) is a signature such that \( \Sigma \subseteq \Sigma_r \). In the following we will use \( \text{Sig}_P \) to denote the set of all possible extensions of \( \Sigma_P \).

### 2.1 Top-down Provenability

We now define provability in \( LO_v \). Let \( \Sigma \) be a signature with predicates and \( \mathcal{V} \) a denumerable set of variables. Given an \( LO_v \) program \( P \) over \( \Sigma \) and \( \mathcal{V} \), the set of ground instances of \( P \), denoted \( \text{Gnd}(P) \), is defined as follows:

\[
\text{Gnd}(P) = \{ (H \circ \gamma \theta) \mid \forall (H \circ \gamma \theta) \in P \},
\]

where \( \theta \) is a grounding substitution for \( H \circ \gamma \theta \) (i.e., it maps variables in \( \text{FV}(H \circ \gamma \theta) \) to ground terms in \( T_\Sigma \)). The execution of a multiset of \( G \)-formulas \( G_1, \ldots, G_k \) in \( P \) corresponds to a goal-driven proof for the \( LO_v \) sequent \( P \vdash \Sigma G_1, \ldots, G_k \). According to this view, the operational semantics of \( LO_v \) is given via the uniform (focusing) [2] proof system presented in Figure 1, where \( P \) is a set of clauses, \( \mathcal{A} \) is a multiset of atomic formulas, and \( \Delta \) is a multiset of \( G \)-formulas. We have used the notation \( \hat{H} \), where \( H \) is a linear disjunction of atomic formulas \( a_1 \lor \ldots \lor a_n \), to denote the multiset \( a_1, \ldots, a_n \) (by convention, \( \bot = 0 \), where \( 0 \) is the empty multiset). We say that \( G \) is provable from \( P \) if there exists a proof tree, built over the proof system of Figure 1, with root \( P \vdash \Sigma G \), and such that every branch is terminated with an instance of the \( \top_r \) axiom. The proof system of Figure 1 is a specialization of more general uniform proof systems for linear logic like Andreoli’s focusing proofs [2] and Forum [28]. Rule \( bc \) is analogous to a resolution step in traditional logic programming languages. By the uniformity of the proof system, it can be executed only if the right-hand side of the current \( LO_v \) sequent consists of atomic formulas. Consider now a branch of a proof terminated by the sequent \( P \vdash \Sigma \mathcal{A} \). When a backchaining step over a clause \( H \circ \gamma \theta \rightarrow \mathcal{A} \) is possible, we immediately obtain an instance of the axiom \( \top_r \), i.e., a successful (branch of) computation independently of the current context \( \mathcal{A} \). This observation is formally stated in the following proposition, where \( \subseteq \) denotes the multiset inclusion relation.

**Proposition 1.** Given an \( LO_v \) program \( P \) and two multisets of goals \( \Delta, \Delta' \) such that \( \Delta \subseteq \Delta' \), if \( P \vdash \Sigma \Delta \) then \( P \vdash \Sigma \Delta' \).

Finally, rule \( \forall_r \) can be used to dynamically introduce new names during the computation. The initial signature \( \Sigma \) must contain at least the constant, function, and predicate symbols of a given program \( P \), and it can dynamically grow thanks to rule \( \forall_r \). Namely, every time rule \( \forall_r \) is fired, a new constant \( c \) is added to the current signature, and the resulting goal is proved in the new one. The idea is that all terms appearing on the right-hand side of a sequent are implicitly assumed to range over the relevant signature. This behaviour is standard in logic programming languages [32].

**Example 2.** Let \( \Sigma \) be a signature with a constant symbol \( a \), a function symbol \( f \) and predicate symbols \( p, q, r, s \). Let \( P \) be the program consisting of the clauses

1. \( r(w) \leftarrow q(f(w)) \)
2. \( s(z) \leftarrow \forall x. p(f(x)) \)
3. \( \bot \leftarrow r(v) \land q(w) \land r(v) \)
4. \( p(x) \leftarrow r(s) \)

The goal \( s(a) \) is provable from \( P \). The corresponding proof is shown in Figure 2 (where \( bc(1) \) denotes backchaining rule over clause number \( i \) of \( P \)). Notice that the notion of ground instance is relative to the current signature. For instance, backchaining over clause 3 is possible because the corresponding signature contains the constant \( c \), and therefore \( \bot \leftarrow q(f(c)) \land r(c) \) is a valid instance of clause 3.

In the rest of the paper we will focus our attention on an observational semantics that captures the provability of a restricted form of \( LO_v \) goals, namely goals consisting of a multiset of ground atomic formulas. Specifically, given a program \( P \) we define its top-down operational semantics as

\[
\mathcal{O}(P) = \{ \mathcal{A} \mid \mathcal{A} \text{ multiset of ground atoms in } \mathcal{A}_{\Sigma_P}, P \vdash_{\Sigma_P} \mathcal{A} \}
\]

Notice that a multiset \( \mathcal{A} = A_1, \ldots, A_k \) (in the r.h.s. of a sequent) is logically equivalent to the multiplicative disjunction \( A_1 \otimes \ldots \otimes A_k \).

### 3. Bottom-Up Evaluation for \( LO_v \)

In this section we introduce the basic ideas underlying the bottom-up evaluation scheme of \( LO_v \) programs. For more details, the reader may refer to [10, 7]. As anticipated in the previous section, we are interested in observing the set of disjunctive atomic goals that are provable in a given program \( P \). By the admissibility of weakening, we observe that if \( \mathcal{A} \in \mathcal{O}(P) \) then \( \mathcal{A} \cup \mathcal{C} \in \mathcal{O}(P) \) for any multiset \( \mathcal{C} \) (of
Given an interpretation $I$, its denotation $[I]$ is the family of ground interpretations $\{[I]_\Sigma\}_{\Sigma \in \mathcal{S}(\varnothing)}$ defined as follows:

$$[I]_\Sigma = U_{p}(\text{Inst}_\Sigma(I)),$$

where the operator $\text{Inst}_\Sigma$ is defined as

$$\text{Inst}_\Sigma(I) = \{A\theta \mid A \in I, \theta \text{ subst. over } \Sigma\}$$

and the operator $U_{p}$ is defined as

$$U_{p}(I) = \{A + \mathcal{C} \mid A \in I, \mathcal{C} \text{ multiset over } A\mathcal{C}\}.$$

Here we assume the substitution $\theta$ and the multiset $\mathcal{C}$ to be defined over $\Sigma$. Notice that, in the definition of $[I]_\Sigma$, the operations of instantiation and upward-closure are performed for every possible signature $\Sigma \in \mathcal{S}(\varnothing)$.

Given an LO$_\Sigma$ goal $G$, we need to define a notion of satisfiability w.r.t. to our definition of interpretation. For this purpose, in Fig 3 we introduce the satisfiability judgment

$$I \models_{\Sigma} \Delta \models \theta,$$

where $I$ is an interpretation, $\Delta$ is a multiset of goal formulas (a context), $\mathcal{C}$ is an output multiset of atomic formulas, and $\theta$ is an output substitution. In Figure 3 $A \backslash B$ denotes the multiset difference between $A$ and $B$. $|A|$ denotes the cardinality of $A$. $\text{FV}(A, \mathcal{C})$ denotes the set of free variables in $A + \mathcal{C}$, and $\equiv$ (as mentioned in the previous section) denotes multiset inclusion.

The judgment is used to compute the set of resources $\mathcal{C}$ ($I$ denotes its upward closure) and the corresponding variable bindings that are needed for $\Delta$ to be provable in $I$. Intuitively, if $I \models_{\Sigma} \Delta \models \theta$ holds then the sequent $P, P' \models_{\Sigma, \gamma} \Delta \theta \gamma, \mathcal{C} \theta \gamma$, $\gamma$ being a grounding substitution, is provable by augmenting $P$ with the program $P'$ consisting of clauses like $A \arrow \top$ for any $A \in [I]$. Technically, the idea behind the definition is that the output multiset $\mathcal{C}$ and the output substitution $\theta$ are minimal so that they can be computed effectively given a program $P$, an interpretation $I$, and a signature $\Sigma$. The output substitution $\theta$ is needed in order to deal with clause instantiation, and its minimality is ensured by using most general unifiers in the definition.

Note: The notation $I \models_{\Sigma} \Delta \models \mathcal{C} \models \theta$ requires that $\Delta, \mathcal{C}$, and $\theta$ are defined over $\Sigma$. As a consequence, the newly introduced constant $c$ in the $\mathcal{C}$-case of the $\models$ definition below cannot be exported through the output parameters $\mathcal{C}$ or $\theta$. This way, universal quantification is always resolved locally. For simplicity, in Fig 3 we present only the formal definition of the judgement for goals without conjunction (see [7] for the complete definition). We recall that two multisets in general may have more than one (not necessarily equivalent) most general unifier and that using the notation $m.g.u.(B', \mathcal{A}')$ we mean any unifier which is non-deterministically picked from the set of most general unifiers of $B'$ and $\mathcal{A}'$.

We are now ready to define the symbolic fixpoint operator $S_P$ working on our notion of interpretation. Given an interpretation $I$, the operator is defined as follows:

$$S_{P}(I) = \{(\tilde{H} + \mathcal{C}) \theta \mid (H \models G) \in \text{Vrn}(P), I \models_{\Sigma, \gamma} G \models \mathcal{C} \models \theta \}.$$  

Here, $+$ denotes multiset union. Furthermore, given $H = A_1 \cdots A_k$, we recall that $\tilde{H}$ is the multiset $A_1, \ldots, A_k$. Finally, $\text{Vrn}(P)$ denotes the set of clauses that are variant (renamed with fresh variables) of clauses in $P$.

**Example 3.** Let us consider a signature with a function symbol $f$ and predicate symbols $p, q, r$. Let $I$ be the interpretation consisting of the multiset $\{p(x), q(x)\}$ (for simplicity, hereafter we omit brackets in multiset notation), and $P$ the program

1. $r(w) \models q(f(w))$
2. $s(z) \models \forall x.p(f(x))$

Let's consider (a renaming of) the body of the first clause, $q(f'(x))$, and (a renaming of) the element in $I$, $p(x')$, $q(x')$. Using the atomic clause for the $\models_{\Sigma, P}$ judgment, with $A =$
Thus, the multiset its instances is provable in A with nately, we can't choose A to be atomic
Permutation of B, so B' ∈ B. In fact, by unifying
Substitution θ must be defined to represent the information in possession of the intruder
First of all, we need a representation for the entities (e.g. principals and messages) involved. In particular, we will use a notation like

\[
pr(id, s)
\]

to denote a principal with identifier id and internal state s. The internal state s can store information about an ongoing execution of any given protocol (for instance, the identifier of another principal, which step of the protocol has been executed, the role of the principal, and so on). Typically, the state s will be a term like init (indicating the initial state of a principal, before protocol execution), or a term like

\[
step(data),
\]

where the constructor \( \text{step} \) denotes which is the last step executed and \( \text{data} \) represents the internal data of a given principal. In general, we allow more than one atom \( pr(id, s) \) inside a given configuration. In this way, we can model the possibility of a given principal to take part into different protocol runs, possibly with different roles. Messages sent over a given network can in turn be represented by terms like

\[
m(message),
\]

where \( \text{message} \) is the content of the message. Depending on the particular protocol under consideration, we can fix a specific format for messages. For instance, a message encrypted with the public key of a principal a could be represented as the term

\[
\text{enc}(pub(a), m),
\]

Finally, we will use the Dolev-Yao intruder model (see [11]) and the associated assumptions. In particular, we need a way to store the intruder knowledge. We will use terms such as

\[
m(message),
\]

to represent the information in possession of the intruder (m stands for the internal memory of the intruder). At any

\[
\begin{align*}
\text{axiom:} & \quad I \models_{\Sigma} T, \Delta \mid \epsilon \mid \text{nil}; \quad \text{anti:} & \quad I \models_{\Sigma} \bot, \Delta \mid \epsilon \mid \theta; \\
\text{par:} & \quad I \models_{\Sigma} G_1 \triangleright G_2, \Delta \mid \epsilon \mid \theta, \quad \text{if} \ I \models_{\Sigma} G_1, G_2, \Delta \mid \epsilon \mid \theta; \\
\text{forall:} & \quad I \models_{\Sigma} x. G, \Delta \mid \epsilon \mid \theta, \quad \text{if} \ I \models_{\Sigma, e} G[e/x], \Delta \mid \epsilon \mid \theta, \quad \text{with} \ c \not\in \Sigma \\
\text{atomic multiset:} & \quad I \models_{\Sigma} A \mid \epsilon \mid \theta, \quad \text{if there exist} \ B \in I \ (\text{variant}), \ B' \leq B, \ A' \preceq A, \ |B'| = |A'|, \ C = B' \mid B', \ \text{and} \ \theta = \text{m.g.a.}(B', A')_{fV(A, C)}.
\end{align*}
\]

Figure 3: Satisfiability judgement for LOq goals with \( \mathcal{G} \), \( \top \), \( \bot \), and \( \forall \).
given instant of time, we can think of the current state of a
given system as a multiset of atoms representing principals and
messages currently on the network, and the intruder
knowledge. Following [11], we represent the environment in
which protocol execution takes place by means of: a protocol
theory, which includes rules for every protocol role (typically,
one rule for every step of the protocol), and an intruder the-
ory, which formalizes the set of possible actions of a mal-
cious intruder who tries to break the protocol. In addition,
it is possible to have additional rules for the environment.
Rules assume the general format

\[ F_1 \not\in \ldots \not\in F_n \rightarrow \forall X_1 \ldots \forall X_k. (G_1 \not\in \ldots \not\in G_m) \]

where \( F_i \), \( G_i \), are atomic formulas (representing e.g. prin-
cipals or messages) and \( X_i \) are variables. As explained in
Section 2, the standard semantics for the universal quanti-
fier requires new values to be chosen before application of
a rule. We use this behaviour to encode nonce generation
during protocol runs. As a result, we get for free the as-
sumption (required by the Dolev-Yao model) that nonces are
not guessable. In the following we will use these nota-
tional conventions: free variables inside a rule are always
implicitly universally quantified, and variables are written as upper-case identifiers.

As far as the specification of the initial states is concerned, we allow a partial specification of the initial states. This strategy is more flexible in that it may help us to find ad-
ditional hypotheses under which a given attack might take
place. As a general rule, the partial specification of the ini-
tial states we have chosen requires every principal to be in
his/her initial state (represented by the term init) at the
beginning of protocol execution.

Finally, we conclude this section by collecting together
some rules which are common to all the examples presented
in Section 5. In particular, we have two rules for the envi-
ronment:

\[
\begin{align*}
\epsilon_1 & \quad \bot \rightarrow \forall ID. (pr(ID, init)) \\
\epsilon_2 & \quad pr(Z, S) \rightarrow pr(Z, S) \not\in \forall pr(Z, init)
\end{align*}
\]

The first one allows the non-deterministic creation of new
principals (we use the universal quantifier to generate new
identifiers for them), whereas the second rule allows creation
of a new instance of a given principal (this allows a principal
to start another execution of a given protocol with a new and
possibly different role). Both rules can be fired at run-time,
i.e., during the execution of a given protocol. Thus, we will
always work in an open environment with multiple sessions
running in parallel between several agents. We use the term
init to denote the initial state of any given principal. We
also have the following two rules for the intruder theory:

\[
\begin{align*}
t_1 & \quad pr(Z, S) \rightarrow pr(Z, S) \not\in m(Z) \\
t_2 & \quad \bot \rightarrow \forall N. (m(N))
\end{align*}
\]

The first one allows the intruder to store the identifier of any
principal, whereas the second rule formalizes the capability of
the intruder to generate new values (e.g. nonces). Be-
fore explaining what kind of properties we can reason about
using the bottom-up evaluation scheme, let us describe the
specification of our main case-study.

4.1 Millen’s ffgg protocol

Although an artificial protocol, Millen’s ffgg protocol [30]
provides an example of a parallel session attack, which re-
quires running at least two processes for the same role. It
has been proved (see [30]) that no serial attacks exist, i.e.,
the protocol is secure if processes are serialized. The proto-
col is described informally as follows.

1. \( A \rightarrow B \rightarrow A \rightarrow : N_1, N_2 \)
2. \( A \rightarrow B \rightarrow : \{N_1, N_2, S\} K_s \}
3. \( A \rightarrow B \rightarrow : \{N_1, X, Y, N_1\} K_s \}
4. \( B \rightarrow A \rightarrow : N_1, X, \{X, Y, N_1\} K_s \)

\( N_1 \) and \( N_2 \) stand for nonces, created by principal \( B \) and
included in message 2. The \( m \% m' \) notation, introduced in
[24], used in message 3 represents a message which has
been created by the sender according to format \( m \), but is interpreted as \( m' \) by the receiver. In this case, the intuition
is that upon receiving message 3, \( B \) checks that the first
component does correspond to the first of the two nonces
previously created, while no check at all is performed on the
second component of the message. In message 3, \( S \) stands for
a secret, of the same length as a nonce, which is in possession
of \( A \). The security property one is interested to analyze is
whether the secret S can be disclosed to a malicious intruder.

We have implemented the ff gg protocol through the spec-
ification shown in Figure 4, while the intruder theory is pre-
SENTED in Figure 5.

The specification consists of a set of protocol rules (rules
\( p_1 \) through \( p_4 \) in Figure 4) and an intruder theory (rules \( t_1 \)
through \( t_4 \) in Figure 5). We remind the reader that the four
rules \( \epsilon_1 \), \( \epsilon_2 \), \( t_1 \) and \( t_2 \) discussed in Section 4 are in addition
to the present rules.

Protocol rules directly correspond to the informal descrip-
tion of the ff gg protocol previously presented. We have fol-
lowed the conventions outlined in Section 4 to model the
internal state of principals. In particular, we have a term
init denoting the initial state of a principal, and the con-
structors \( step_1, step_2, step_3 \) and \( step_4 \) to model the different
steps of a protocol run. At every step, each principal needs
to remember the identifier of the other principal he/she is

\[
\begin{align*}
\text{Figure 4: Specification of the ff gg protocol}
\end{align*}
\]
executing the protocol with. In addition, at step 2 the responder stores the first nonce created (in order to be able to perform the required check, see rule \( p_4 \)), and at step 3 the initiator of the protocol remembers the secret \( S \). We have modeled the secret \( S \) using the universal quantifier, as for nonces. In this way, we can get for free the requirement that the secret initially is only known to the principal who possesses it. Finally, we have term constructors \( \text{plain}(\ldots) \) and \( \text{enc}(\ldots) \) (to be precise, we should say a family of term constructors, we find it convenient to overload the same symbol with different arities) to distinguish plain messages from encrypted messages.

The intruder theory is made up of rules \( i_1 \) through \( i_8 \) in Figure 5. It is an instance of the general Dolev-Yao intruder theory (see e.g. [11]). Let us discuss it in more detail. Rules \( i_1 \) through \( i_4 \) are decomposition rules, whereas rules \( i_5 \) through \( i_8 \) are composition rules. We have four rules for each of the two different kinds (composition and decomposition) of messages, dealing with the different formats of messages used in the \( ffgg \) protocol. For instance, rule \( i_1 \) deals with decomposition of plain messages with one component, whereas rule \( i_5 \) deals with decomposition of messages with two plain components and one encrypted component, and so on. Clearly, the intruder cannot further decompose encrypted components, which are stored exactly as they are, whereas plain messages are decomposed into their atomic constituents. The intruder theory we have presented is an instance of the general Dolev-Yao intruder theory, in that intruder rules have been tailored to the particular form of messages used in the specific protocol under consideration, an optimization often taken by verification methods [21]. This hypothesis can be relaxed (as we did for the analysis of Needham-Schroeder protocol, see Section 5). The present specification is sufficient for our purposes.

### 5. VERIFICATION OF SECURITY PROPERTIES

In order to understand how to apply our bottom-up evaluation strategy for the analysis of security protocols, we first need the following general observation. Several practical examples of safety properties present the following interesting feature: their negation can be represented by means of the upward closure of a collection of minimal violations, e.g., as for mutual exclusion properties of communication protocols [1]. Thanks to this property, it often becomes possible to finitely represent infinite collections of unsafe configurations. Symbolic procedures can be applied then in order to saturate the set of predecessor states (by iteratively applying a transition relation backwards). Using this method and assuming that a fixpoint is eventually reached, it is possible then to establish which initial states lead to violations of the property.

This observation can be applied in our setting in order to specify interesting security properties. As an example, in the \( ffgg \) protocol we consider a configuration unsafe if there exist at least two honest principals, say \( alice \) and \( bob \), who have run the protocol to completion (i.e., they have completed, respectively, step 4 and step 3) and the secret \( S \) has been disclosed to the intruder (i.e., it is eventually stored in the intruder’s internal memory). In our setting a configuration is represented as a multiset of atomic formulas. In order to symbolically represent all possible configurations in which a violation might occur, we can use then the \( LO_\forall \) clause in Figure 6. Every top-down derivation leading from an initial goal (state) to an instance of the axiom \( \top \), obtained by applying the rule \( u \) will represent a possible attack to the protocol security. It is important to note that, by the admissibility of weakening, the previous \( LO_\forall \) rule can be used to represent unsafe configurations for any number of principals involved in sessions running in parallel with the session carried over by \( alice \) and \( bob \). Exploring all possible top-down derivations however corresponds to an exhaustive search of the state space of the specification and it would force us to fix a given initial configuration.

Contrary, by evaluating bottom-up the \( LO_\forall \) program obtained by merging the protocol and intruder theory with

![Figure 5: Intruder theory for the \( ffgg \) protocol](image)

![Figure 6: A logical representation of an infinite set of unsafe configuration for the \( ffgg \) protocol](image)

---

\( i_1: n(\text{plain}(X)) \rightarrow m(\text{plain}(X)) \)

\( i_2: n(\text{plain}(X, Y)) \rightarrow m(\text{plain}(X)) \not\Rightarrow m(\text{plain}(Y)) \)

\( i_3: n(\text{enc}(X, Y, Z, W)) \rightarrow m(\text{enc}(X, Y, Z, W)) \)

\( i_4: n(\text{plain}(X, Y), \text{enc}(U, V, W, Z)) \rightarrow m(\text{plain}(X)) \not\Rightarrow m(\text{plain}(Y)) \not\Rightarrow m(\text{plain}(U, V, W, Z)) \)

\( i_5: m(\text{plain}(X)) \rightarrow m(\text{plain}(X)) \not\Rightarrow n(\text{plain}(X)) \)

\( i_6: m(\text{plain}(X)) \not\Rightarrow m(\text{plain}(Y)) \not\Rightarrow m(\text{plain}(X)) \not\Rightarrow m(\text{plain}(Y)) \not\Rightarrow n(\text{plain}(X, Y)) \)

\( i_7: m(\text{enc}(X, Y, Z, W)) \rightarrow m(\text{enc}(X, Y, Z, W)) \not\Rightarrow m(\text{enc}(X, Y, Z, W)) \not\Rightarrow m(\text{enc}(X, Y, Z, W)) \not\Rightarrow m(\text{enc}(X, Y, Z, W)) \not\Rightarrow n(\text{plain}(X, Y), \text{enc}(U, V, W, Z)) \)

---

Given that sets of unsafe configurations are encoded via logical axioms, computing the backward reachability set of a transition relation amounts to evaluating the corresponding logic program bottom-up.
the symbolic representation of unsafe states like the clause \( u \), we obtain the same effect using backward reachability for a complex specification (with quantification and so on) carried over in a completely open environment. Furthermore, if a fixpoint is reached (this is not guaranteed in general) we can derive conditions on the initial states under which unsafe configurations will not be reached. In other words we can establish the following connection between bottom-up evaluation (i.e., the semantics \( \mathcal{F}(P) \) defined in Section 3) of an LOV specification of an authentication protocol and verification of security properties. Let \( \mathit{Init} \) be a collection of multisets of ground atomic formulas (the initial states of a protocol), \( T_p \) be the LOV theory encoding a protocol \( P, T_i \) the LOV intruder theory, and let \( U \) be a collection of LOV clauses \( A_1 \leftarrow T, \ldots, A_k \leftarrow T \) (the minimal violations of a security property \( S \)). Furthermore, let \( T = T_p \cup T_i \cup U \).

**Proposition 5 (Ensuring Security).** The protocol \( P \) is secure w.r.t. the intruder with capabilities \( T_i \), initial configurations \( \mathit{Init} \), and the property \( S \), if and only if \( \mathit{Init} \cap \{F(T)\} = \emptyset \).

We remark that an if and only if condition holds in the previous proposition, i.e. the bottom-up evaluation algorithm is correct and complete. As a corollary, we get the following property (only if direction in the previous proposition), useful for debugging purposes.

**Proposition 6 (Proving Insecurity).** If there exists \( A \) such that \( A \in \mathit{Init} \cap \{F(T)\} \), then there exists an attack that leads from the initial configuration \( A \) to an unsafe configuration \( B \in \{U\} \).

### 5.1 Some Practical Experiments

A simple prototype which implements the bottom-up evaluation procedure for LOV programs has been implemented in SML as described in [10]. Running our bottom-up evaluation algorithm on the \( \mathrm{ffgg} \) specification, we automatically find a violation to the security property of Figure 6.

As in traditional model checking, counterexamples traces can be automatically generated whenever a violation is found. In particular, the trace corresponding to the above attack is shown in Figure 7 (we only post-processed the output of
our verification tool to show the trace in a more human-readable form. The trace in Figure 7 corresponds to an L0V derivation which leads from an initial state to a state violating the security property of Figure 6. The attack is exactly the parallel session one described in [30]. We note that this attack is also an example of a type flaw attack, in that it relies on the secret $S$ that this attack is also an example of a type flaw.

In order to let the reader better understand the connection between bottom-up evaluation used by our verification algorithm and the top-down derivation shown in Figure 7, we present below some of the steps performed by the bottom-up evaluation algorithm. In the following we follow the same syntactical notations as in Figure 7. Bottom-up evaluation starts from axiom $i$, i.e. we assert the following provable multiset:

$$m_1) \text{alt}_{bob,S}, bob_1, m(S),$$

where $S$ is a free variable. Different clauses are applicable at this point. Among them, decomposition rule $i_1$. We can apply a variant of $i_4$, let it be

$$n(X', Y', \{V', W', Z'\}_{K}) \vdash m(X') \not\equiv m(Y')$$

$m_1$, in the following manner: unify $S$ with $Y'$ (hence unifying $n(S)$ in the multiset with $m(Y')$ in the clause body) and consider the other two atoms in the body (i.e. $m(X')$ and $m(\{V', W', Z'\}_{K}))$ as being implicitly contained in $m_1$ (remember that interpretations are to be considered upward-closed). By applying the resulting clause backwards (i.e. the body is replaced by the head) we get the multiset

$$m_2) \text{alt}_{bob,S}, bob_1, n(X', S, \{V', W', Z'\}_{K}).$$

Multiset $m_2$ is accumulated into the current set of provable goals (other multisets can be obtained by applying the remaining program clauses). Now, consider the application of a variant of protocol rule $p_4$, let it be

$$(B'')_{A, N_1, N_2} \not\equiv n(\{N_1'', X'', Y''\}_{K_2}) \vdash (B'')_{A, N_1, N_2} \not\equiv n(\{N_1'', X'', Y'', N_1''\}_{K_2})$$

to $m_2$, in the following way: unify $N_1''$ with $X''$, $X''$ with $S$ and $V'$, $Y''$ with $W'$, and $B''$ with $U$ (thus unifying the atom $n(\{N_1'', X'', Y'', N_1''\}_{K_2})$ with the atom $n(X', S, \{V', W', Z'\}_{K_2}))$. Furthermore, assume the atom $(B'')_{A, N_1, N_2}$ to be implicitly contained in $m_2$. We get the multiset

$$m_3) \text{alt}_{bob,S}, bob_1, (B'')_{A, N_1, N_2}, n(\{N_1'', X'', Y''\}_{K_2})$$

which is in turn accumulated as a provable goal. We invite the reader to observe the correspondence between the bottom-up construction we are sketching and the top-down construction illustrated in Figure 7. Notice that the sequence of rules we are applying is the same but in the reversed order, i.e. axiom $u$, then rules $i_4$ and $p_4$, and so on (clearly, we are illustrating only one among the possible bottom-up derivations). Furthermore, every atom that we described as implicitly contained in the current multiset corresponds to one of the atoms in the top sequent of Fig. 7. In other words, the bottom-up computation starts from a multiset representing the minimal violations of the security property under consideration (i.e., axiom $u$), whereas any additional atom that turns out to be involved in the proof (see top-sequent in Fig. 7) is (implicitly) added, so to say, in a lazy manner as the bottom-up construction proceeds. Variable bindings can also be (implicitly) enforced during the bottom-up construction. For instance, the atom $(B'')_{A, N_1, N_2}$ (which we assumed to be implicitly contained in $m_3$) corresponds to the atom $bob_2$ in the top sequent of Fig. 7. Eventually, variable $B''$ (which is contained, e.g. in $m_3$) will be unified with $bob$, and similarly, $A''$ will be unified with $al$.

We conclude the illustration of the bottom-up construction with an example of a clause involving universal quantification. Proceeding as above, eventually we get (a variant) of the multiset

$$m_4) \text{alt}_{bob,S}, bob_2, \{al_1, N_1', N_2', \text{bob}_2\}_{A, N_1', N_2'}, n(\{N_1', N_2', S\}_{K_2}).$$

Now, we can apply a variant of protocol rule $p_3$ (see the corresponding inference in Fig. 7), let it be

$$(A')_{N_1', N_2'} \not\equiv n(\{N_1', N_2'\}) \vdash \forall S'. ((A')_{N_1', N_2', S'} \not\equiv n(\{N_1', N_2', S'\}_{K_2}))$$

to $m_4$, by unifying $A'$ with $al$, $B'$ with $bob$, $S'$ with $S$ with $N_1'$ with $N_1'$ and $N_1'$ with $N_2'$. We get the following multiset:

$$m_5) \text{alt}_{bob,S}, bob_2, \{al_1, N_1', N_2', \text{bob}_2\}_{A, N_1', N_2'}, n(\{N_1', N_2'\}).$$

Notice that the bottom-up inference requires a new constant, let it be $c$, to be introduced in place of the universally quantified variable $S'$ in the body of the above clause. According to rule $forall$ for the satisfiability judgment (see Figure 3) a static check must be performed in order to ensure that the output multiset and unifier do not contain the constant $c$. This check is successfully passed, thus the above inference is perfectly legal. The bottom-up construction goes on in this way until the multiset $alt^{alt}_m, bob^{alt}_m$ (corresponding to the bottom sequent in Figure 7) is reached.

We conclude by mentioning that we have also performed some further experiments regarding Millen’s ffgg protocol, which we don’t discuss in detail. In particular, we wanted to ascertain the role of the two nonces $N_1$ and $N_2$ in the ffgg protocol. According to the informal notation for the protocol introduced at the beginning of this section, principal $B$ only checks that the first component of message $T$ is the nonce $N_1$, whereas no check is performed for the second component. We have verified that imposing the check on the second component, the ffgg protocol is safe w.r.t. the security property and the intruder theory we have presented, while removing all checks, as expected, introduces serial attacks.

We think that this example is a good illustration of the capabilities of our general framework. In fact, using the backward evaluation strategy championed in this paper, we are able to automatically find a parallel session attack, without enforcing any particular search strategy of our evaluation algorithm (i.e. the same algorithm can be used to find serial or parallel attacks). Furthermore, according to [30] the ffgg protocol can be generalized to protocols which only admit higher-order parallel attacks (i.e., attacks which take place only in presence of three or more concurrent roles for the same principal). Using the same algorithm, and the same protocol and intruder theories as before, we can automatically find such attacks, if any exists. This distinguishes our methodology from most approaches based on model-checking, which operate on a finite-state abstraction of a given protocol, and require the number of principals and the number of roles to be fixed in advance.
Another advantage of using backward reasoning is related to the generation of fresh nonces. Forward exploration needs to explicitly manage generation of fresh names. The backward application of LO rules allows us instead to observe only formulas defined over the signature of the original program. In fact, suppose that a rule body contains universally quantified variables that have to be matched with an interpretation computed during the bottom-up evaluation of a program. By the definition of the satisfiability judgement (Fig. 3) and of the $SP$ operator, we can restrict ourselves to a local top-down derivation in which we simplify the body of the clause (see rule for all and par of Fig. 3) and then we match the resulting multiset of formulas against the current interpretation (see rule atomic multiset of Fig. 3). In the end a static check is made in order to ensure that the output multiset and the resulting unifier do not contain the constants which have been introduced. In other words the effect of quantification in the body of a clause is simply that of restricting the set of possible predecessor configurations. Notice that, on the contrary, using top-down evaluation, the current signature is enriched by adding a new constant every time the rule for the universal quantifier is used.

In addition to Millen’s protocol we have used our method to study other more classical examples of authentication protocols. As an example we have analyzed the Needham-Schroeder protocol (discovering Lowe’s attack [23]), verified the corresponding corrected version, and analyzed the Otway-Rees protocol. We stress that our algorithm can be used either to find attacks of given protocols, or to prove that no attacks may exist (clearly, w.r.t. a given protocol theory and a given intruder theory). In other words, the bottom-up evaluation algorithm presented in Section 3 is correct and complete: no proof is found (w.r.t. to the given protocol and intruder theories), then no proof at all may exist.

All the experiments (performed on a Pentium II 233MHz under Linux 2.0.32, running Standard ML of New Jersey, Version 110.0.7) are summarized in Figure 8. The tag Invar indicates the use of invariant strengthening (the set of unsafe states is enriched with the negation of other invariants), a conservative technique that can speed up the fixpoint computation. Furthermore, Size denotes the number of multisets inferred during the evaluation, MSize the maximal number of multisets computed at any step, and Time the execution time in seconds. More details on these experiments can be found in [10] (available on the first author’s web page http://www.disi.unige.it/person/BozzanoM/).

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Invar</th>
<th>Steps</th>
<th>Size</th>
<th>MSize</th>
<th>Time</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Millen’s ffg</td>
<td></td>
<td>14</td>
<td>306</td>
<td>677</td>
<td>1335</td>
<td>attack</td>
</tr>
<tr>
<td>Millen’s ffg (corrected)</td>
<td></td>
<td>14</td>
<td>27</td>
<td>810</td>
<td>2419</td>
<td>yes</td>
</tr>
<tr>
<td>Needham-Schroder</td>
<td></td>
<td>13</td>
<td>294</td>
<td>323</td>
<td>45</td>
<td>attack</td>
</tr>
<tr>
<td>(strong correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham-Schroder</td>
<td></td>
<td>13</td>
<td>304</td>
<td>755</td>
<td>516</td>
<td>yes</td>
</tr>
<tr>
<td>(weak correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corrected Needham-Schroder</td>
<td></td>
<td>12</td>
<td>299</td>
<td>299</td>
<td>65</td>
<td>yes</td>
</tr>
<tr>
<td>(strong correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Otway-Rees</td>
<td></td>
<td>9</td>
<td>402</td>
<td>402</td>
<td>31</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>10339</td>
<td>10339</td>
<td>4272</td>
<td>attack</td>
</tr>
</tbody>
</table>

Figure 8: Analysis of authentication protocols: experimental results

6. CONCLUSIONS AND RELATED WORK

In this paper we have presented security protocols as a possible application field for our methodology based on a linear logic-based specification language and on a bottom-up evaluation strategy. Our verification procedure is tailored to study security violations which can be specified by means of minimality conditions. While this may rule out interesting properties, e.g. questions of belief [4], the proposed approach can be used to study secrecy and confidentiality properties. No artificial limit is imposed on the number of simultaneous sessions we are able to analyze.

We have performed some experiments on different authentication protocols which show that the methodology we propose can be effective either to find attacks or to validate existing protocols. We plan to overcome some current limitations of our approach, in particular we plan to refine and automatize the specification phase of protocols and of the intruder theory. Specifically, we want to study a (possibly automatic) translation between the usual informal description of protocols and our representation. As shown in the paper, a one-to-one translation (one rule for every step) could be enough, provided we have a way to store the information about the internal state of principals. For efficiency reasons, it could also be worth to investigate some optimizations, in particular to the intruder theory (concerning, e.g., the rules for composition and decomposition). We plan to use techniques like folding / unfolding to automatize this process.

Another topic we would like to investigate is typed multiset rewriting [13], which extends multiset rewriting with a typing theory based on dependent types with sub-sorting. Dependent types can be used to enforce dependency between an encryption key and its owner. The paper [13] also presents some extensions which increase the flexibility of multiset rewriting specifications, e.g. using memory predicates to remember information across role executions.

Finally, an open question is that of non-termination. In the few examples we have presented, our algorithm is always terminating, even without invariant strengthening. Although secrecy has been proved to be undecidable, even for finite-length protocols with data of bounded complexity [11], one may ask if a more restricted subclass of protocols exists, for which the verification algorithm presented here is always terminating.

We conclude by discussing some related work. A wide research area in security protocol analysis is related to rewriting. For instance, we mention [14], which specifies security protocols as rewriting theories which can be executed...
in the ELAN system. A similar approach is followed in [17], where the target executable language is instead Maude. Perhaps the more interesting work in this class is [21]. This work presents an automatic compilation process from security protocol descriptions into rewrite rules. The resulting specifications are then executed using the dafac theorem prover. As a difference with [17], which is based on matching, the execution strategy of [21] relies on narrowing and AC unification. Our approach, based on multiset unification, is clearly closer to the latter approach, although currently we do not support equational theories. All of the above approaches are limited to protocol debugging, therefore they can find attacks mounted on a given protocol, but they cannot be used to analyze correctness. Also, a crucial difference is that all the above works are based on a forward breadth-first-search strategy, while effectiveness of our verification algorithm strongly relies on a backward search strategy. Another approach which shares some similarity with ours is [16], where a specification for security protocols based on rewriting and encoded in a subset of intuitionistic logic is presented. The author uses universal quantification to generate nonces, like us, and embedded implication to store the knowledge of agents. This approach is still limited to protocol debugging.

An alternative approach to verifying security protocols is based on model checking. For instance, the FDR model checking tool was used by Lowe [23] to analyze the Needham-Schroeder public-key protocol. Other works which fall into this class are [25, 34]. All these approaches have in common the use of some kind of abstraction to transform the original problem into a finite-state model-checking problem, which is then studied by performing a forward reachability analysis. Using a finite-state approximation has the advantage of guaranteeing termination, however it only allows one to analyze a fixed number of concurrent protocol runs, an approach which is infeasible as this number increases. As a difference, we use a symbolic representation for infinite sets of states and a backward reachability verification procedure, which avoid putting limitations on the number of parallel sessions we are able to analyze.

Theorem proving techniques are used in [33], where protocols are inductively defined as sets of traces, and formally analyzed using the theorem prover Isabelle. Here, analysis is a semi-automatic process which can take several days. The NRL protocol analyzer [26] provides a mixed approach. It is based on protocol specifications given via Prolog rules, and enriched via a limited form of term rewriting and narrowing to manage symbolic encryption queries. Similarly to us, verification is performed by means of a symbolic model-checker which relies on a backward evaluation procedure which takes as input a set of insecure states. The analyzer needs to be fed with some inductive lemmas by the user, in the same way theorem provers need to be guided by the user during the proof search process.

In [8], the author proposes an optimized specification of security protocols based on an “attacker view” of protocol security, specified by means of Prolog rules, as in [26]. The approach is effective, and has been applied to prove correctness of a number of real protocols. The verification algorithm performs a backward depth-first search, which seems to be closely related to our evaluation strategy, and uses an intermediate code optimization using a technique similar to unfolding, which we plan to study as future work. On the other hand, we think that the multiset rewriting formalism which we use is more amenable to an automatic translation from the usual protocol notation. Ensuring faithfulness between the intended semantics of a protocol and its specification is necessary to prove correctness. Also, with respect to [8], we use a cleaner treatment for nonces, and we don’t have to use approximations (which may introduce false attacks) except for invariant strengthening, which can be controlled by the user.

Finally, we mention some works concerning the process of translation from the usual informal notation for protocols, which we plan to study as part of our future work. Existing approaches include Casper [24], a compiler from protocol specifications into the CSP process algebra, oriented towards verification in FDR, and CAPSL [29], a specification language which can be compiled into an intermediate language and used to feed tools like Maude [17] or the NRL analyzer [26]. Finally, [21] presents an automatic compilation process into rewriting rules which is able to manage infinite-state models.

Concerning the application of linear logic to verification, we would like to mention the work in [18], where phase semantics is used to prove properties of specification of concurrent constraint programs. The phase semantics for LO proposed by Andreoli could be the possible connection between the manual ‘semantic-driven’ method of [18] and our automated ‘syntactic-driven’ method that could be interesting to investigate.

The technical details of the bottom-up evaluation strategy for LO_2 programs is described in [7] (as practical example, in [7] we have studied a parameterized mutual exclusion protocol). The first author’s PhD thesis [10] also contains a detailed presentation and proofs for the results presented in this paper, and all the details of the experimental results mentioned in Section 5.1. Some preliminary results (e.g. Needham-Schroeder protocol) were also discussed in [9].

7. REFERENCES


