A Meta-notation for Protocol Analysis*

I. Cervesato N.A. Durgin P.D. Lincoln J.C. Mitchell A. Scedrov

Computer Science Lab
SRI International
Menlo Park, CA
lincoln@csl.sri.com

Computer Science Dept.
Stanford University
Stanford, CA 94305-9045
{iliano, nad, jcm}@cs.stanford.edu

Mathematics Dept.
University of Pennsylvania
Philadelphia, PA
andre@cis.upenn.edu

Abstract

Most formal approaches to security protocol analysis are based on a set of assumptions commonly referred to as the "Dolev-Yao model." In this paper, we use a multiset rewriting formalism, based on linear logic, to state the basic assumptions of this model. A characteristic of our formalism is the way that existential quantification provides a succinct way of choosing new values, such as new keys or nonces. We define a class of theories in this formalism that correspond to finite-length protocols, with a bounded initialization phase but allowing unboundedly many instances of each protocol role (e.g., client, server, initiator, or responder). Undecidability is proved for a restricted class of these protocols, and PSPACE*-completeness is claimed for a class further restricted to have no new data (nonces). Since it is a fragment of linear logic, we can use our notation directly as input to linear logic tools, allowing us to do proof search for attacks with relatively little programming effort, and to formally verify protocol transformations and optimizations.*

1 Introduction

In the literature on security protocol design and analysis, protocols are commonly described using an informal notation that leaves many properties of a protocol unspecified. For example, a short challenge-response section of a protocol might be written like this:

$$\begin{array}{ll} A \longrightarrow B: & \{n\}_K \\ B \longrightarrow A: & \{f(n)\}_K \end{array}$$

In this notation, a message of the form $\{x\}_y$ consists of a plaintext x encrypted with key y. In this example protocol, Alice chooses a random number n and sends its encryption to Bob. There is no specific indication of how Bob determines what to send in response, but we can see that Bob returns a message that contains the encryption of $f(n)$. By analogy with familiar protocols, we might assume that he decrypts the message he receives to determine n, then applies f to n and returns the result to Alice (encrypted with the same key).

As written, the protocol description only gives an intended trace or family of traces involving the honest principals. There is no standard way of determining the initial conditions or assumptions about shared information, nor can we see how the principals will respond to messages that differ from those explicitly written. For example, in the case at hand, we must explain in English that K is assumed to be a shared key and that n is generated by Alice. Otherwise, it is a perfectly reasonable interpretation of the two lines above that Alice and Bob initially share a number n. In this case, Alice might send $\{n\}_K$ to Bob, with Bob returning $\{f(n)\}_K$ to Alice only if he receives precisely $\{n\}_K$. While the two readings of the protocol give the same sequence of messages when no one interferes with network transmission, the effects are different if an intruder intercepts the message from Alice to Bob and replaces it with another message. For this reason, the notation commonly found in the literature does not provide a precise basis for security protocol analysis.

Most formal approaches to protocol analysis are based on a relatively abstract set of modeling assumptions, commonly referred to as the "Dolev-Yao model," which appear to have developed from positions taken by Needham and Schroeder [26] and a model presented by Dolev and Yao [11]. In this approach, messages are composed of indivisible abstract values, not sequences

*Partially supported by DoD MURI "Semantic Consistency in Information Exchange" as ONR Grant N00014-97-1-0505, and by NSF Grants CCR-9509931, CCR-9629754, and CCR-9800785 to various authors.

of bits, and encryption is modeled in an idealized way. Although the same basic modeling assumptions are used in theorem proving [27], model-checking methods [18, 20, 25, 28, 29] and symbolic search tools [17], there does not appear to be any standard presentation of the Dolev-Yao model as it is currently used in a variety of projects. One goal of this paper is to identify the modeling assumptions using the simplest formalism possible, so that the strengths and weaknesses of the Dolev-Yao model can be analyzed, apart from properties of logics or automated tools in which the model is commonly used.

While we began with the idea of creating a new formalism for this purpose, we naturally gravitated toward some form of rewriting, so that protocol execution could be carried out symbolically. In addition to rewriting to effect state transitions, we also needed a way to choose new values, such as nonces or keys. While this seems difficult to achieve directly in standard rewriting formalisms, the proof rules associated with existential quantification appears to be just what is required. Therefore, we have adopted a notation, first presented in [24], that may be regarded as either an extension of multiset rewriting (see, e.g., [3, 4]), with existential quantification, or a Horn fragment of linear logic [14]. A similar fragment of linear logic is used in [16] to represent real-time finite-state systems. Two other efforts using linear logic to model the state-transition aspect of protocols (but not existential quantification for nonces) are [8, 9].

Using this formalism, it is relatively straightforward to characterize the Dolev-Yao intruder and associated cryptographic assumptions. The formalism also seems appropriate for analyzing the complexity of protocol problems, and as a potential intermediate language for systems or approaches that might combine several different protocol analysis tools. We develop a format for presenting finite-length protocols, as the disjoint union of a set of initialization rules and sets of independent transition rules for each protocol participant. Using this form of protocol theory, we show that secrecy is an undecidable property even if data constructors, message depth, message width, number of distinct roles, role length, and depth of encryption are bounded by constants. If any of these restrictions are lifted, prior results, folklore, or a small amount of thought can be used to show undecidability, but we show even for the very small fragment with only nonces secrecy is undecidable. Finally, we have used a linear logic tool, *LLF* [5] in two ways. The first is to search executions of a protocol and intruder for protocol flaws. While symbolic search by a logic programming tool is not as efficient as optimized search by tools such as Murφ[10], this method does have the advantage that the input is substantially easier to prepare. The second use of LLF is to formally verify proofs of protocol optimizations. This provides a basis for simplifying search-based analysis and theorem-proving analysis of protocols.

2 Multiset rewriting with existential quantification

The notation we use involves *facts* and *transitions*. Our facts are first-order atomic formulas, and transitions are given by rewrite rules containing a precondition and postcondition. One important property of this formalism is that in applying a rule to a collection of facts, each fact that occurs in the precondition of the rule is removed. This gives us a direct way of representing state transitions, and provides the basis for the connection with linear logic. Another key property is that the postconditions of a rule may contain existentially quantified variables. Following the standard proof rules associated with existential quantification (in natural deduction or sequent-style systems), this provides a mechanism for choosing new values that are distinct from any other in the system.

More formally, our syntax involves terms, facts and rules. If we want to represent a system in this notation, we begin by choosing a vocabulary, or *first-order signature*. This is a standard notion from many-sorted algebra or first-order logic (see, e.g., [13, section 4.3].) As usual, the *terms* over a signature are the well-formed expressions produced by applying functions to arguments of the correct sort. A *fact* is a first-order atomic formula over the chosen signature. This means that a fact is the result of applying a predicate symbol to terms of the correct sorts. A *state* is a multiset of facts (all over the same signature).

A state transition is a *rule* written using two multisets of facts, and existential quantification, in the following syntactic form:

$$F_1, \dots, F_k \longrightarrow \exists x_1 \dots \exists x_j . G_1, \dots, G_n$$

The meaning of this rule is that if some state S contains facts $F_1, \dots F_k$, then one possible next state is the state S' that is similar to S, but with:

- facts $F_1, \dots F_k$ removed,
- $G_1, \dots G_m$ added, where $x_1 \dots x_j$ are replaced by new symbols.

While existential quantification does not semantically imply there exist "new" values with certain properties, standard proof rules for manipulating existential quantifiers require introduction of fresh symbols (sometimes called Skolem constants), as described below.

If there are free variables in the rule $F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$, these are treated as universally quantified throughout the rule. In an application of a rule, these variables may be replaced by any terms. To give a quick example, consider the following state, S, and rule, R:

$$\begin{array}{rcl} S & = & \{P(f(a)), P(b)\} \\ R & = & (\,P(x) \longrightarrow \exists z.\, Q(f(x), z)\,) \end{array}$$

One possible next state is obtained by instantiating the rule R to $P(f(a)) \longrightarrow \exists z.\, Q(f(f(a)), z)$. Applying this rule, we choose a new value, c, for z and replace $P(f(a))$ by $Q(f(f(a)), c)$. This gives us the state

$$S' = \{Q(f(f(a)), c), P(b)\}$$

The importance of existential quantification, for security protocols, is that it provides a direct mechanism for choosing a new value that is different from other values used in the execution of a system. Since many protocols involve choosing fresh nonces, fresh encryption keys, and so on, existential quantification seems like a useful primitive for describing security protocols.

The way that existential quantification is used in our formalism is based on the existential elimination rule from natural deduction. This proof rule is commonly written as follows.

$$(\exists\ \mathbf{elim}) \qquad \frac{\exists x.\phi \qquad \begin{array}{c}[y/x]\phi \\ \vdots \\ \psi\end{array}}{\psi} \quad \begin{array}{l} y \text{ not free in any} \\ \text{other hypothesis}\end{array}$$

If we have an existentially quantified axiom, $\exists x.\phi$, then this rule says that if we wish to prove some formula ψ, we can choose a new symbol y for the "x that is presumed to exist" and proceed to derive ψ from $[y/x]\phi$. The side condition "y not free in any other hypothesis in the proof of ψ" means that the only hypothesis in the proof of ψ that can contain y is the hypothesis $[y/x]\phi$.

As a means of explaining the Dolev-Yao intruder and encryption models using our notation, we begin with an overly simplified form of the Needham-Schroeder public-key protocol [26]. Without encryption, the core part of the Needham-Schroeder protocol proceeds as follows:

$$\begin{array}{ccccc} A & \longrightarrow & B & : & N_a \\ B & \longrightarrow & A & : & N_a, N_b \\ A & \longrightarrow & B & : & N_b \end{array}$$

where N_a and N_b are fresh nonces, chosen by Alice (A) and Bob (B), respectively.

We can describe this simplified protocol in our notation using the predicates A_i, B_i, N_i for $0 \le i \le 3$, with the following intuitive meaning:

$A_i(\,\ldots\,)$	Alice in local state i, with the indicated data
$B_i(\,\ldots\,)$	Bob in local state i, with indicated data
$N_i(\,\ldots\,)$	Network has message i, with indicated data

The data associated with the state of some principal, or a network message, will depend on the particular state or message. Each principal begins in local state 0, with no data. Therefore, predicates A_0 and B_0 are predicates with no arguments. When Alice chooses a nonce, she moves into local state 1. Therefore, predicate A_1 is a predicate of one argument, intended to be the nonce chosen by Alice. Similarly, predicate B_1 has two arguments, the data received from Alice in message one of the protocol and the nonce chosen by Bob for his response.

Using these predicates, we can state the protocol using four transition rules:

$$\begin{array}{rcl} A_0() & \longrightarrow & \exists x. A_1(x), N_1(x) \\ B_0(), N_1(x) & \longrightarrow & \exists y. B_1(x,y), N_2(x,y) \\ A_1(x), N_2(x,y) & \longrightarrow & A_2(x,y), N_3(y) \\ B_1(x,y), N_3(y) & \longrightarrow & B_2(x,y) \end{array}$$

Each rule corresponds to an action by a principal. In the first rule, Alice chooses a nonce, sends it on the network, and remembers the nonce by moving into a local state that retains the nonce value. In the second step, Bob receives a message on the network, chooses his own nonce, transmits it and saves it in his local state. In the third step, Alice receives Bob's message and replies, while in the fourth step Bob receives Alice's final message and changes state.

In Table 1 is a sample trace generated from these rules, beginning from state A_0, B_0. Spacing is used to separate the facts that participate in each step from those that do not.

There are two main parts of the Dolev-Yao model as commonly used in protocol analysis. The first is the set of possible intruder actions, applied nondeterministically throughout execution of the protocol. The

$$
\begin{array}{rrcll}
B_0(), & A_0() & \longrightarrow & A_1(\mathsf{nA}), N_1(\mathsf{nA}), & B_0() \\
A_1(\mathsf{nA}), & B_0(), N_1(\mathsf{nA}) & \longrightarrow & B_1(\mathsf{nA}, \mathsf{nB}), N_2(\mathsf{nA}, \mathsf{nB}), & A_1(\mathsf{nA}) \\
B_1(\mathsf{nA}, \mathsf{nB}), & A_1(\mathsf{nA}), N_2(\mathsf{nA}, \mathsf{nB}) & \longrightarrow & A_2(\mathsf{nA}, \mathsf{nB}), N_3(\mathsf{nB}), & B_1(\mathsf{nA}, \mathsf{nB}) \\
A_2(\mathsf{nA}, \mathsf{nB}), & B_1(\mathsf{nA}, \mathsf{nB}), N_3(\mathsf{nB}) & \longrightarrow & B_2(\mathsf{nA}, \mathsf{nB}), & A_2(\mathsf{nA}, \mathsf{nB})
\end{array}
$$

second is a "black-box" model of encryption and decryption. We explain the intruder actions here, with the encryption model presented in Section 2.4.

The protocol adversary or "intruder" may nondeterministically choose among the following actions at each step:

- Read any message and block further transmission,
- Decompose a message into parts and remember them,
- Generate fresh data as needed,
- Compose a new message from known data and send.

By combining a read with resend, we can easily obtain the effect of passively reading a message without preventing another party from also receiving it.

In general, the intruder processes data in three phases. The first is to read and decompose data into parts. The second is to remember parts of messages, and the third is to compose a message from parts it remembers. We illustrate the basic form of the intruder actions using one unary network-message predicate N_1 and one binary network-message predicate N_2. Using predicates D for decomposable messages and M for the intruder "memory", the basic rules for intercepting, decomposing and remembering messages are

$$
\begin{array}{lcl}
N_1(x) & \longrightarrow & D(x) \\
N_2(x, y) & \longrightarrow & D(x, y) \\
D(x, y) & \longrightarrow & D(x), D(y) \\
D(z) & \longrightarrow & M(z)
\end{array}
$$

While the predicate D may appear to be an unnecessary intermediary here, a protocol with more complicated messages will lead to more interesting ways of destructuring messages. As noted in [7], it is important in proof search to separate the decomposition phase from the composition phase, which is accomplished here using separate D and C predicates. The rules for composing messages from parts are written using the C, for "composable", predicate as follows:

$$
\begin{array}{lcl}
M(x) & \longrightarrow & C(x), M(x) \\
C(x) & \longrightarrow & N_1(x) \\
C(x), C(y) & \longrightarrow & C(x, y) \\
C(x, y) & \longrightarrow & N_2(x, y)
\end{array}
$$

The rule for generating new data is

$$\longrightarrow \ \exists x. M(x)$$

The reason we need the last transition rule (which can be applied any time without any hypothesis) is that the intruder may need to choose new data in order to trick an honest participant in a protocol. This is illustrated in the following attack on the simplified (and obviously insecure) form of the Needham-Schroeder protocol.

For the simplified example at hand, we can compose rules to eliminate the D and C predicates as follows:

$$
\begin{array}{lcl}
N_1(x) & \longrightarrow & M(x) \\
M(x) & \longrightarrow & N_1(x), M(x) \\
N_2(x, y) & \longrightarrow & M(x), M(y) \\
M(x), M(y) & \longrightarrow & N_2(x, y), M(x), M(y) \\
N_3(x) & \longrightarrow & M(x) \\
M(x) & \longrightarrow & N_3(x), M(x)
\end{array}
$$

This reduces the number of steps in the trace, shown in Table 2, which has actions of the honest participants in the left column and actions of the intruder indented. For simplicity, duplicate copies of $M(\ )$ facts are not shown, since these have no effect on the execution of the protocol or intruder.

In this attack, the intruder intercepts messages between A and B, replacing data so that the two principals have a different view of the messages that have been exchanged. Specifically, the intruder replaces Alice's nonce nA by a value n chosen by the intruder. When Bob responds to the altered message, the intruder intercepts the result and replaces n by nA so that Alice receives the message she expects.

The commonly used "black-box" model of encryption may be written in our multiset notation using the following vocabulary. For concreteness, we discuss public-key encryption. Symmetric or private-key encryption can be characterized similarly. We assume that plaintexts have sort plain and ciphertexts have sort cipher.

$B_0(), A_0()$	Initial configuration
$\longrightarrow A_1(\mathsf{nA}), N_1(\mathsf{nA}), B_0()$	Alice chooses nonce and sends
$\longrightarrow A_1(\mathsf{nA}), B_0(), M(\mathsf{nA})$	Intruder intercepts message nA
$\longrightarrow A_1(\mathsf{nA}), B_0(), M(\mathsf{nA}), M(n)$	Intruder generates fresh value n
$\longrightarrow A_1(\mathsf{nA}), N_1(n), B_0(), M(\mathsf{nA}), M(n)$	Intruder sends n to Bob
$\longrightarrow B_1(n, \mathsf{nB}), N_2(n, \mathsf{nB}), A_1(\mathsf{nA}), M(\mathsf{nA}), M(n)$	Bob receives, generates nonce, replies
$\longrightarrow A_1(\mathsf{nA}), B_1(n, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Intruder intercepts message with n
$\longrightarrow A_1(\mathsf{nA}), N_2(\mathsf{nA}, \mathsf{nB}), B_1(n, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Intruder sends message with nA
$\longrightarrow A_2(\mathsf{nA}, \mathsf{nB}), N_3(\mathsf{nB}), B_1(n, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Alice receives and responds
$\longrightarrow B_2(n, \mathsf{nB}), A_2(\mathsf{nA}, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Bob changes to final state, indicating successful completion of protocol

- Additional sorts: $\mathsf{e\_key}, \mathsf{d\_key}$
- Predicate: $\mathsf{Key\_pair}(\mathsf{e\_key}, \mathsf{d\_key})$
- Function: $\mathsf{enc} : \mathsf{e\_key} \times \mathsf{plain} \rightarrow \mathsf{cipher}$

We could also include a decryption function $\mathsf{dec} : \mathsf{d\_key} \times \mathsf{cipher} \rightarrow \mathsf{plain}$. However, it seems simpler to write protocols using pattern-matching (encryption on the left-hand-side of a rule) to express decryption.

Example The core Needham-Schroeder protocol with encryption begins with each principal generating and publishing a key pair. Therefore the "local state 0" predicate for each principal will contain a key pair. For example, Bob's initial actions can be stated as the following rules:

$$\begin{array}{rcl} & \longrightarrow & \exists k : \mathsf{e\_key}.\ \exists k' : \mathsf{d\_key}.\ B_0(k), \\ & & \mathsf{Key\_pair}(k, k') \\ B_0(k) & \longrightarrow & \mathsf{Announce}(k), B_0(k) \end{array}$$

The first rule (without hypotheses) lets Bob generate a key pair, while the second announces the public key so that other principals can choose to communicate with Bob. Alice similarly chooses and publishes her public key. After doing so, she chooses a principal to communicate with from the set of announced public keys and transmits a message.

$$\begin{array}{rcl} A_0(k), \mathsf{Announce}(k') & \longrightarrow & \exists x.\ A_1(k, k', x), \\ & & N_1(\mathsf{enc}(k', \langle x, k \rangle)), \\ & & \mathsf{Announce}(k') \end{array}$$

The following transition rule then allows Bob to decrypt the message from Alice.

$$\begin{array}{rcl} B_0(k), N_1(\mathsf{enc}(k, \langle x, k' \rangle)) & \longrightarrow & \exists y.\ B_1(k, k', x, y), \\ & & N_2(\mathsf{enc}(k', \langle x, y \rangle)) \end{array}$$

A complete presentation with slightly more general initialization steps is given in Appendix A, for the interested reader.

Intruder To model the encryption capabilities of the intruder, we add a decomposition and a composition rule to the intruder model. The decomposition rule allows the intruder to decrypt a message (or part of a message) when the decryption key is known.

$$\begin{array}{l} D(\mathrm{enc}(k,x)),\ \mathsf{Key\_pair}(k, k'),\ M(k') \\ \quad \longrightarrow \quad D(x),\ \mathsf{Key\_pair}(k, k'),\ M(k') \end{array}$$

The composition rule allows the intruder to encrypt a message with any encryption key known to the intruder.

$$M(k),\ C(x) \longrightarrow C(\mathrm{enc}(k,x)),\ M(k)$$

3 Bounded protocols

It is relatively straightforward to use the multiset rewriting framework summarized in the preceding section to describe finite-state and infinite-state systems. Using function symbols, it is possible to describe computation over unbounded data types. In particular, it is easy to encode counter machines or Turing machines, implying that implication is undecidable. However, the principal authentication and secrecy protocols of interest are all of bounded length (see [6] for a relevant survey).

In order to study finite-length protocols more carefully, we identify the syntactic form of a class of well-founded protocol theories, called simply *well-founded theories* in this paper.

Some preliminary definitions involve the ways that a fact may be created, preserved, or consumed by a rule. While multiple copies of some facts may be needed in some derivations, we are able to eliminate the need for multiple copies of certain facts.

Definition 1. A rule $l \rightarrow r$ in a theory $\mathcal{T}$ *creates* P facts if some $P(\vec{t})$ occurs more times in r than in l. A rule $l \rightarrow r$ in a theory $\mathcal{T}$ *preserves* P facts if every $P(\vec{t})$ occurs the same number of times in r and l. A rule $l \rightarrow r$ in a theory $\mathcal{T}$ *consumes* P facts if some fact $P(\vec{t})$ occurs more times in l than in r. A predicate P in a theory $\mathcal{T}$ is *persistent* if every rule in $\mathcal{T}$ which contains P either creates or preserves P facts.

To give an example, a rule of form

$$P(\vec{x}) \rightarrow P(\vec{y})$$

does not preserve P facts, since it can be used to create a fact $P(\vec{t})$ and consume a fact $P(\vec{s})$.

Since a persistent fact is never consumed by any rule, there is no need to generate more than one copy of a particular fact – as long as that fact is never needed twice by a single rule. By simple transformation, it is possible to eliminate the need for more than one copy of any persistent fact. For example, a rule of form:

$$P(\vec{x}), P(\vec{y}), \ldots \rightarrow Q(\vec{x}, \vec{y}), P(\vec{x}), P(\vec{y}), \ldots$$

(with P a persistent predicate) can be replaced by rules of form:

$$\begin{array}{lcl} P(\vec{x}) & \rightarrow & P_1(\vec{x}), P(\vec{x}) \\ P(\vec{x}) & \rightarrow & P_2(\vec{x}), P(\vec{x}) \\ P_1(\vec{x}), P_2(\vec{y}), \ldots & \rightarrow & Q(\vec{x}, \vec{y}), P_1(\vec{x}), P_2(\vec{y}), \ldots \end{array}$$

where P_1 and P_2 are persistent predicates.

Definition 2. A rule $l \rightarrow r$ in a theory $\mathcal{T}$ is a *single-persistent rule* if all predicates that are persistent in theory $\mathcal{T}$ appear at most once in l. A theory $\mathcal{T}$ is a *uniform theory* if all rules in $\mathcal{T}$ are single-persistent rules.

Since any theory can be rewritten as a uniform theory, we will assume that all theories discussed from this point are uniform theories.

Definition 3. Let $\mathbf{P}$ be a set of predicates, each persistent in a uniform theory $\mathcal{T}$. Two states S and S' are *P-similar* (denoted $S \simeq_P S'$) if, after removing all duplicate persistent P facts from each state, they are equal multisets.

Lemma 1. *Given a uniform theory $\mathcal{T}$, $\mathbf{P}$ the set of predicates persistent in $\mathcal{T}$, σ a substitution, and R a rule that creates only persistent facts, if $S \xrightarrow{\mathcal{T}} T$ is a derivation which invokes σR more than once, then there exists a derivation $S \xrightarrow{\mathcal{T}} T'$ that invokes σR only once, with $T \simeq_P T'$.*

In many protocols, there is an implicit or explicit initialization phase that distributes keys or establishes other shared information. We incorporate this into our formal definitions by letting a protocol theory consist of an initialization theory, together with the disjoint union of bounded subtheories that characterize the behavior of each protocol agent (role). In order to bound the entire protocol, we must assume that the initialization theory is bounded, and that initialization can be completed prior to the execution of the protocol steps proper.

Definition 4. A rule $R = l \rightarrow r$ *enables* a rule $l' \rightarrow r'$ if there exist σ, σ' such that some fact $P(\vec{t}) \in \sigma r$ is also in $\sigma' l'$. A theory $\mathcal{T}$ *precedes* a theory $\mathcal{R}$ if no rule in $\mathcal{R}$ enables a rule in $\mathcal{T}$.

In particular, if a theory $\mathcal{T}$ precedes a theory $\mathcal{R}$, then no predicates that appear in the left hand side of rules in $\mathcal{T}$ are created by rules that are in $\mathcal{R}$.

Definition 5. A theory $\mathcal{A}$ is a *well-founded principal theory* if it has an ordered set of predicates, called the *principal role states* and numbered $A_0, A_1, \ldots, A_k$ for some k, such that each rule $l \rightarrow r$ contains exactly one state predicate $A_i \in l$ and one state predicate $A_j \in r$, with $i < j$. We call the first role state, A_0, an *initial role state.*

By defining a principal theory in this way, we ensure that each application of a rule in $\mathcal{A}$ advances the state forward. Each instance of a principal role can only result in a finite number of steps in the derivation.

Definition 6. A theory $\mathcal{S} \subset \mathcal{T}$ is a *bounded sub-theory* if all rules R in $\mathcal{S}$ that create a fact fall into one of the following categories:

1. The facts created by R contain existentials.
2. The facts created by R are initial role states of $\mathcal{T}$.
3. The facts created by R are persistent in $\mathcal{T}$.

Definition 7. A theory $\mathcal{P}$ is a *well-founded protocol theory* if $\mathcal{P} = \mathcal{I} \uplus \mathcal{A} \uplus \mathcal{B} \uplus \mathcal{C} \ldots$ where $\mathcal{I}$ is a bounded sub-theory (called the *initialization theory*) and $\mathcal{A}, \mathcal{B}, \mathcal{C} \ldots$ are a finite number of well-founded principal theories, with $\mathcal{I}$ preceding $\mathcal{A}, \mathcal{B}, \mathcal{C} \ldots$.

The structure of protocol theories allows derivations to be broken down into two stages – the initialization stage, and the non-initialization (protocol run) stage. Any derivation in the theory can be reordered to contain all initialization steps before any non-initialization steps.

Lemma 2. *Given a well-founded protocol theory $\mathcal{P} = \mathcal{I} \uplus \mathbf{A}$, where $\mathcal{I}$ is the initialization theory, and $\mathbf{A}$ is the disjoint union of one or more principal theories, if $S \xrightarrow{\mathcal{P}} T$ is a derivation over $\mathcal{P}$, then there is a derivation $S \xrightarrow{\mathcal{I}} S'$ and $S' \xrightarrow{\mathbf{A}} T$, where all rules from $\mathcal{I}$ are applied before any rules from $\mathbf{A}$.*

One motivation for using multiset rewriting for protocol analysis is that this framework allows us to use essentially the same theory for all adversaries for all protocols. In this subsection, we specify the properties of intruder theories that are needed to bound the number of intruder steps needed to produce a given message. As explained in [7], the actions of the standard intruder can be separated into two phases, one in which messages are decomposed into smaller parts, and one in which these parts are (re)assembled into a message that will be sent to some protocol agent. An additional detail is that we sometimes need to postpone decomposition of a specific fact until a later time, such as until a decryption key becomes available.

In determining the *size* of a fact, we count the predicate name, each function name, and each variable or constant symbol. For example, fact $P(A, B)$ has size 3, and fact $P(f(A, B), C)$ has size 5.

Definition 8. A rule $R = l \to r$ is a *composition rule* if the non-persistent facts in r have larger size than the facts in l. A rule $R = l \to r$ is a *decomposition rule* if the non-persistent facts in r have smaller size than the facts in l.

For example,

$$C(A), C(B) \to C(\langle A, B\rangle)$$

is a composition rule, and

$$D(\langle A, B\rangle) \to D(A), D(B)$$

is a decomposition rule.

For the intruder theories we will consider, we allow persistent facts to appear in both the left and right hand sides. So, in general a decomposition rule is of form:

$$D(\langle A, B\rangle), \vec{P}(\dots) \to D(A), D(B), \vec{P'}(\dots)$$

where $\vec{P}$ and $\vec{P'}$ are sets of persistent predicates, with $\vec{P} \subseteq \vec{P'}$ (and similarly for composition rules).

We also need to introduce more complicated decomposition rules, which we call "Decomposition rules with Auxiliary facts". These are pairs of rules of form:

$$D(t), \vec{P}(\dots) \to \vec{P'}(\dots), A(t)$$

and

$$A(t), \vec{Q}(\dots) \to \vec{Q'}(\dots), D(t')$$

where $\vec{P} \subseteq \vec{P'}$, $\vec{Q} \subseteq \vec{Q'}$, and $size(t') < size(t)$. Here, A represents an Auxiliary fact (which can appear only in a pair of rules of this form) which is used to amortize the decomposition of $D(t)$ into $D(t')$ across the two rules. Appendix A.3 shows an example of when this type of decomposition rule is needed, in order to allow for decrypting an old fact with a newly learned encryption key.

Definition 9. A theory $\mathcal{T}$ is a *two-phase* theory if its rules can be divided into three disjoint theories, $\mathcal{T} = \mathcal{I} \uplus \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{I}$ is a bounded sub-theory preceding $\mathcal{C}$ and $\mathcal{D}$, $\mathcal{C}$ contains only composition rules, $\mathcal{D}$ contains only decomposition rules, and no rules in $\mathcal{C}$ precede any rules in $\mathcal{D}$.

Definition 10. A *normalized derivation* is a derivation where all rules from the decomposition theory are applied before any rules from the composition theory.

As also shown in [7] in a slightly different context, all derivations in a two-phase theory can be expressed as normalized derivations.

Lemma 3. *If a theory $\mathcal{T}$ is two-phase, and we limit the size of terms, and we limit the number of times each existential is instantiated, then there are a finite number of normalized derivations in the theory.*

Definition 11. Given a well-founded protocol theory $\mathcal{P} = \mathcal{I} \uplus \mathbf{A}$ and a two-phase intruder theory $\mathcal{M}$, a *standard trace* is a derivation which has all steps from the initialization theory $\mathcal{I}$ first, then interleaves steps from the principal theories $\mathbf{A}$ with normalized derivations from the intruder theory $\mathcal{M}$.

Theorem 1. *Let $\mathcal{P}$ be any well-founded protocol theory and $\mathcal{M}$ be any two-phase intruder theory. If we bound the number of uses of each existential, and we bound the size of each term, then the set of standard traces of $\mathcal{P} \cup \mathcal{M}$ is finite.*

4 Intractability

There are many undecidable properties of arbitrary protocols. If we consider all possible systems that are definable using Horn logic as protocols, then undecidability follows easily from the undecidability of Horn-clause logic. In particular, it is possible to give finite descriptions of infinite-state systems using function symbols. For example, if we have $0 : \mathsf{nat}$ and $\mathsf{suc} : \mathsf{nat} \rightarrow \mathsf{nat}$, then we can write expressions for arbitrarily many natural numbers. It is straightforward to go from there to various undecidability results based on counter machines. However, this kind of intractability result ignores the fact that most protocols used in practice have a fixed number of steps and communicate data of bounded complexity. If we restrict our attention to finite-length protocols, of the form identified in the previous section, then it does not seem possible to write protocols whose behavior is as complicated as arbitrary Turing machines. However, as shown in this section, there are nontrivial lower bounds if we combine protocols with an intruder. Throughout this section, we will restrict our attention to runs in which all terms have fewer than some fixed number of symbols and the protocol principals have bounded activity.

Even without generating new data, determining a security property may require exponentially-many runs of a protocol and the decision problem is PSPACE-Complete. The first result, that there are protocols without nonces where the shortest insecure run is exponential, is illustrated by the following family of protocols, one for each integer k. The protocol for integer k assumes that a private symmetric key K is shared between principals $A, B_1, ..., B_k$ and C. (The same effect can occur achieved in a public key protocol, by first running secure key exchange steps.) In Table 3 we show the protocol for $k = 4$.

In this artificial protocol, A sends C a message containing $(0, 0, 0, 0)$ but C will only respond to a message with $(1, 1, 1, 1)$. However, principals $B_1, \ldots, B_k$ implement a k-bit increment on encrypted tuples. Therefore, if an intruder routes the initial message from A through $2^k - 1$ B principals in repeated runs of the protocol, C will expose the secret key. It is easy to see that unless an exponential number of messages are sent, the key K remains secret.

If we formulate security as a decision problem without requiring the faulty trace as output, it is easy to show that the problem is PSPACE-hard by straightforward reduction from linear-bounded Turing machines. An interesting aspect of the protocol above is that it shows that a protocol can be secure against polynomial-time attack, but considered insecure under Dolev-Yao assumptions.

4.2.1 Restricted protocol form

In general, it is convenient to write the steps of a protocol agent A in the form

$$A_i(\ldots), N_j(\ldots), P(\ldots), Q(\ldots), \ldots$$
$$\rightarrow \vec{\exists} \ldots . A_k(\ldots), N_k(\ldots), P(\ldots), Q(\ldots), \ldots$$

where $P(\ldots), Q(\ldots), \ldots$ are persistent facts appearing on the left and right of the rule. However, for the purpose of proving a stronger negative result, we restrict our attention to a simpler form of protocol step in this section. Specifically, we say a protocol theory is in *restricted form* if the constituent principal theories consist only of rules of the form

$$A_i(\ldots), N_j(\ldots) \rightarrow \vec{\exists} \ldots . A_k(\ldots), N_\ell(\ldots)$$

with one principal role state and one network message on the left and one principal role state and one network message on the right. As with all finite-length protocols, we assume the states of agent A are given by a finite list of predicates $A_1, \ldots, A_a$. We also assume that set of possible network messages are given by a finite list of predicates $N_1, \ldots, N_n$, with $i < k \leq a$ and $j < \ell \leq n$ in each rule of the form above.

Theorem 2. *Secrecy is undecidable for finite-length protocols of restricted form. More specifically, there is no algorithm for deciding whether a given protocol, run in combination with the standard intruder, allows the intruder to gain access to a given initial secret.*

The proof involves representing existential Horn theories as protocols, as described below. While space limitations prevent us from presenting the proof in detail, we will sketch the main ideas.

4.2.2 Representation of existential Horn theories

Given a set of existential Horn formulas, of the form described in Appendix B and repeated below, we can construct a protocol so that, when combined with the standard intruder theory, the intruder memory may contain formulas representing all consequences of the theory.

In order to do this, we must use the intruder in an essential way. Specifically, each agent can only execute

$$
\begin{array}{ll}
A \longrightarrow C: & (0,0,0,0)_K \\
C \longrightarrow A: & \text{if sent } (1,1,1,1)_K \text{then respond } K \\
B_1 \longrightarrow A: & \text{if sent } (x_1,x_2,x_3,0)_K \text{then respond } (x_1,x_2,x_3,1)_K \\
B_2 \longrightarrow A: & \text{if sent } (x_1,x_2,0,1)_K \text{then respond } (x_1,x_2,1,0)_K \\
B_3 \longrightarrow A: & \text{if sent } (x_1,0,1,1)_K \text{then respond } (x_1,1,0,0)_K \\
B_4 \longrightarrow A: & \text{if sent } (0,1,1,1)_K \text{then respond } (1,0,0,0)_K
\end{array}
$$

$k = 4$

a finite sequence of steps. Therefore, we use a separate agent for each Horn clause. The role of the intruder is to convert the final message sent by one agent to an initial message received by another agent. As a result of intruder actions, a datum may pass through an unbounded number of protocol steps.

At the same time, in order to represent the Horn theory faithfully, we cannot give the intruder complete access to all to atomic formulas used in a Horn clause. In particular, we cannot let the intruder combine data from different messages. For example, if one agent sends a message representing $P(a, b)$, we cannot allow the intruder to intercept this message and replace it with $P(b, a)$. However, it is easy to prevent this form of interference if we encrypt atomic formulas with a shared private key.

Putting these two ideas together, we represent an existential Horn clause theory by a protocol with one agent per clause. The agent A for a clause

$$
\begin{array}{l}
\forall x_1 \ldots \forall x_i [(\alpha_1 \wedge \ldots \wedge \alpha_k) \\
\implies \exists y_1 \ldots \exists y_j (\beta_1 \wedge \ldots \wedge \beta_\ell)]
\end{array}
$$

is

$$
\begin{array}{l}
A_0, N_{a_0}(\lceil \alpha_1 \wedge \ldots \wedge \alpha_k \rceil) \\
\rightarrow \exists y_1 \ldots \exists y_j .\, A_1, N_{a_1}(\lceil \beta_1 \wedge \ldots \wedge \beta_\ell \rceil)
\end{array}
$$

where the encoding $\lceil \alpha_1 \wedge \ldots \wedge \alpha_k \rceil$ of a conjunction of atomic formulas as a single atomic formula is described below.

There are several ways to represent a conjunction of atomic formulas as a single network message, containing only one predicate symbol, hidden from the adversary. One way, intended to keep syntactic complication to a minimum, is to assume that for each k, we have a k-ary encryption function $\mathsf{Encrypt}_k$. For notational simplicity, we will suppress k in the description below. We also assume that for each sequence of predicates $P_1, \ldots, P_k$ that occurs together in the left or right hand side of is a given Horn clause, we have a constant symbol $P_1.P_2.\ldots.P_k$. (Although we have used a sequence of letters, numbers and subscripts to write out our name for this constant symbol, we assume it is an atomic constant symbol of the language.) Given these assumptions, we let

$$
\begin{array}{l}
\lceil P_1(t_{1,1}, \ldots t_{1,i_1} \wedge \ldots \wedge P_j(t_{j,1}, \ldots, t_{j,i_1}) \rceil \\
= \mathsf{Encrypt}(P_1.P_2.\ldots.P_k, t_{1,i_1}, \ldots, t_{j,1}, \ldots, t_{j,i_1})
\end{array}
$$

We assume that all encryption is done using the same key, the key is shared among honest participants, but not revealed to the intruder. (It suffices to have one principal executing several roles, using a private key.)

The final main idea in the representation of Horn theories is the way that a set of conjunctions may be combined to produce the conjunction of atomic formulas needed to apply another Horn clause. This is perhaps best illustrated by example.

Suppose that the existential Horn clause

$$\forall x \forall y [(P(x) \wedge Q(x, y) \wedge R(y)) \implies \exists z (P(z) \wedge Q(y, z)]$$

is part of the Horn theory we wish to represent by a protocol. In order to use this implication, the protocol must produce a message containing $\lceil (P(a) \wedge Q(a, b) \wedge R(b) \rceil$ for some a and b. However, the protocol agents that represent Horn clauses produce encodings of conjunctions of atomic formulas, and the atomic formulas here may come from different rules. Therefore, we need additional protocol agents that select atomic formulas out of conjunctions and combine them.

The process is very similar to the encoding of the intruder, except that protocol agents can manipulate encrypted values. For each conjunction form (including variables) that appears on the right-hand side of a Horn clause, such as $P(z) \wedge Q(y, z)$, we include *decomposition agents* of the form

$$A_0, N_0(\lceil P(z) \wedge Q(y, z) \rceil) \rightarrow A_1, N_1(\lceil P(z) \rceil)$$

and

$$B_0, N_0(\lceil P(z) \wedge Q(y, z) \rceil) \rightarrow B_1, N_1(\lceil Q(y, z) \rceil)$$

for predicates A_0, A_1, B_0, B_1 not used for other agents. We also need a *composition agent* for the left-hand-side of each original Horn clause. For the clause above, the agent will have states A_0, A_1, A_2 and A_3. At each step, the agent reads one of the atomic formulas in its target conjunction, sending out either a dummy message or, at the last step, a message containing the conjunction of atomic formulas needed. In order to assemble $\lceil(P(x) \wedge Q(x,y) \wedge R(y)\rceil$, we can use an agent A with the following steps:

$$\begin{array}{l}
A_0, N_0(\lceil P(x)\rceil) \rightarrow A_1(\lceil P(x)\rceil), N_1() \\
A_1(\lceil P(x)\rceil), N_2(\lceil Q(x,y)\rceil) \rightarrow \\
\qquad A_2(\lceil P(x) \wedge Q(x,y)\rceil), N_3() \\
A_2(\lceil P(x) \wedge Q(x,y)\rceil), N_4(\lceil R(y)\rceil) \rightarrow \\
\quad A_3(), N_5(\lceil P(x) \wedge Q(x,y) \wedge R(y)\rceil)
\end{array}$$

After this agent sends message N_5, the intruder can read the data $\lceil P(x) \wedge Q(x,y) \wedge R(y)\rceil$ contained in this message and forward it to the agent representing the Horn clause with hypothesis $P(x) \wedge Q(x,y) \wedge R(y)$.

In each of our examples, we have numbered the network messages $N_0, N_1, \ldots$, but it is possible to renumber them so that each message number has a fixed format.

5 Multiset rewriting and LLF

The multiset-rewriting notation used in this paper is the first-order Horn fragment of linear logic [14], with existential quantification. Specifically, each transition rule

$$A_1, \ldots, A_n \longrightarrow \exists\vec{x}.B_1, \ldots, B_m$$

can be written as a linear logic formula

$$A_1 \otimes \ldots \otimes A_n \multimap \exists\vec{x}.B_1 \otimes \ldots \otimes B_m$$

Under this correspondence, every derivation using multiset rewriting corresponds to a linear logic derivation, and conversely. This allows us to use linear logic tools for protocol analysis. In particular, we have used the linear logical framework *LLF* [5] to simulate the execution of protocols, detect attacks, and construct formal proofs about protocol transformations. Similar results could be obtained using other linear logic systems such as *Forum* [23] or *Lygon* [15], except that our proofs about protocol transformations rely on the proof-term representation of *LLF*, discussed below.

Since we will describe some aspects of the protocol proofs carried out using LLF, we give a brief overview of LLF. Although operators $\otimes$ and $\exists$ are not provided directly in LLF, it is possible to write rules as logically equivalent formulas using other linear logic connectives. Although the exact syntax is not overly important, we give an illustration since it provides an opportunity to illustrate an operational reading of linear logic formulas. As in other forms of logic, conjunction in hypotheses, $A \otimes B \multimap C$, is equivalent to nested implication, $A \multimap B \multimap C$. There is also a double-negation property of linear logic, with $(A \multimap \mathit{false}) \multimap \mathit{false}$ equivalent to A, that allows us to write existential quantification using negated universal quantification. In place of *false*, we use a propositional variable, which is implicitly universally quantified. This leads to a form of "continuation-passing" logic program, of the sort usually associated with double-negation in constructive logic. Using *loop* as our propositional variable, the formula $A_1 \otimes \ldots \otimes A_n \multimap \exists\vec{x}.B_1 \otimes \ldots \otimes B_m$ can be written as the clause

$$\begin{array}{ll}
 & A_1 \multimap \ldots \multimap A_n \\
\multimap & \forall\vec{x}.(B_1 \multimap \ldots \multimap B_m \multimap \mathit{loop}) \\
\multimap & \mathit{loop}
\end{array}$$

Following conventional logic-programming notation, we often write this clause with the outer implications reversed:

$$\begin{array}{ll}
\mathit{loop} & \circ\!- A_1 \\
 & \ldots \\
 & \circ\!- A_n \\
 & \circ\!- \forall\vec{x}.(B_1 \multimap \ldots \multimap B_m \multimap \mathit{loop})
\end{array}$$

This formula may be read operationally as follows: "in order to make an iteration, consume $A_1, \ldots, A_n$, generate new constants $\vec{c}$ and substitute them for the variables $\vec{x}$, then assert the facts $B_1, \ldots, B_m$, and finally try another rule (clause)".

The signature used to describe protocols uses different types for messages, keys, nonces, and so on. For example, we use type `key` for both public keys and principal names, which we identify. Nonces have type `atm`, for atomic message, while other messages have type `msg`. The function symbols `@` and `k2m` are used as type conversion functions from nonces and keys to messages, respectively. Composite messages are obtained from their parts using the infix `*` operator, and a message M encrypted with key K is written `crypt` M K. A key K is made public by a fact of the form `annKey` K. The inverse of a key K is specified as `inv` K.

Messages sent by a principal have the form `toNet` P N M, where P identifies the protocol, N is the message number within that protocol, and M is the message itself. Principals receive messages in packets of the form `fromNet` P N M. As we will see, both the network and the intruder can convert `toNet` assertions to `fromNet` assertions.

In actual LLF syntax, Alice's first step in the Needham-Schroeder protocol is written as follows:

```
nsA1: loop
      o- annKey B
      o- a0 A
      o- ({Na:atm}
            a1 A B (@ Na)
         -o toNet ns 1
            (crypt ((@ Na) * (@ (k2m A))) B)
         -o loop).
```

Here `nsA1` is a label, used to name this clause in traces, and {`Na:atm`} indicates a quantified variable `Na` of type `atm`.

The network simply converts every `toNet` assertion into a `fromNet` assertion, so that each message sent to the network can be read as message from the network by other principals. The intruder can intercept a message and decompose it into its atomic constituents (of the form `@` N or `k2m` K). The intruder then builds up a message from these fragments and possibly newly generated data. Finally it sends it off in the form of a `fromNet` message.

As an example equivalence proof, we describe an optimization of our Needham-Schroeder model and prove the equivalence between the "standard" model with network and intruder and an optimized version in which the network is eliminated. Since the intruder can simulate the network, by decomposing and then recomposing each message, the two models are equivalent, in the sense that for every trace of the system with network and intruder, there is a corresponding trace of the system without the network in which all honest parties see exactly the same sequences of messages, and conversely.

We state the equivalence using standard logical notation. Specifically, we write $\Gamma \vdash A$ if there is a derivation $\mathcal{D}$ of this judgment of linear logic. Let NS, I, N, and $Init$ be the multisets of linear logic formulas representing the steps of the Needham-Schroeder protocol, the intruder, the network, and the initial facts, respectively. For any pair of principals A,B in $Init$, let ϕ_{AB} be the formula

$$\phi_{AB} = \exists x_a, x_b . A_2(A, B, x_a, x_b) \otimes B_2(B, A, x_a, x_b)$$

stating that the protocol between A and B runs to completion. We use this formula as an example goal for proof search; similar equivalences hold for other goals.

Theorem 3 (Equivalence). *If there is a derivation $\mathcal{D}$ of $(NS, I, N, Init) \vdash \phi_{AB}$, then there is a derivation $\mathcal{D}'$ of $(NS, I, Init) \vdash \phi_{AB}$. Moreover, it is the case that $\mathcal{D}$ and $\mathcal{D}'$ contain the same protocol steps in the same order.*

While LLF is not a complete proof checker, there is a precise sense in which LLF can be used to develop machine-checkable proofs. In comparison with proof methods using more standard logics, e.g. [27], LLF has some advantages and disadvantages. One advantage is that the constructive nature of proofs is immediately apparent: from an LLF proof of the equivalence above, we obtain an algorithm that transform a trace of one system into a trace of the other. This is in part a result of the way we formalize the equivalence, and in part a consequence of the constructive nature of our fragment of linear logic. A disadvantage of LLF is that it supports only a restricted fragment of linear logic. Moreover, we must write proofs in a certain style in order for LLF to be able to check their correctness.

Overview of the proof We wish to prove that for every trace, possibly relying on the network to carry messages to principals, there exist an equivalent trace where the network is never used. We do this by formulating two versions of the protocol. Traces of the system including a separate network have type `loop1`, while traces of the system without a network have type `loop2`. Our definition of *equivalent* traces is that there is no noticeable difference from the point of view of any honest participant in the protocol: they send and receive the same messages and apply the same transitions. The LLF formalization of this proof involves definition of a binary predicate `net2intr` which relates a derivation possibly involving the network (of type `loop1`), and a derivation that is network free (of type `loop2`). The arguments of this predicate are LLF terms corresponding to derivations and therefore representing traces. The proof of equivalence proceeds by induction on the structure of the derivation using the network, using a nested induction on the structure of the messages whenever we encounter a network transmission.

LLF proof checking The LLF equivalence proof consists of clauses that formalize each inductive case of the proof. While LLF checks the type of each clause, type checking does not guarantee that the proof is correct since (a) the LLF implementation does not guarantee that each case is covered, and (b) it does not check whether uses of the induction hypothesis are well-founded. However, we can prove an adequacy theorem which shows that any type-checked proof of a certain form must be a correct proof.

In order to state this result, we write $\Delta \vdash_{LLF} M : A$ for the judgment that the LLF term M has type A with respect to the (intuitionistic and linear) declarations in Δ. Let Δ_{NS} be the complete set of declarations used in the equivalence proof.

Theorem 4 (Adequacy of representation). *Let A and B be principals in Init, and ϕ_{AB} defined as above*

- *There is a bijection between derivations $\mathcal{D}$ of $(NS, I, N, Init) \vdash \phi_{AB}$ and LLF terms M such $\Delta_{NS} \vdash_{LLF} M$: `loop1` is derivable,*
- *There is a bijection between derivations $\mathcal{D}'$ of $(NS, I, Init) \vdash \phi_{AB}$ and LLF terms M' such that $\Delta_{NS} \vdash_{LLF} M'$: `loop2` is derivable,*
- *For every term M such that $\Delta_{NS} \vdash_{LLF} M$: `loop1` is derivable, there exist terms M' and P such that $\Delta_{NS} \vdash_{LLF} M'$: `loop2` and $\Delta_{NS} \vdash_{LLF} P$: `net2intr` M M' are derivable.*

The terms M and M' above contain the same sequences of protocol steps, and the same `toNet` and `fromNet` tokens, in the same order.

6 Conclusion

We believe that a logic-based formalism as described in this paper, with existential quantification for choosing new values, provides a useful notation for examining the standard Dolev-Yao protocol security assumptions. Using this formalism, we can describe all protocols we have encountered by formulating a rewrite rule (or several related rules) for each step of each principal. In addition, we have developed standard theories (sets of rewrite rules) for initialization steps such as generating and announcing public keys, and for the nondeterministic Dolev-Yao intruder. This gives us a set of conventions for describing protocols and provides the basis for developing general theorems and techniques for protocol analysis.

There are several advantages and intentional disadvantages of the formalism used in this paper. The intent of the design is that this formalism provides exactly the primitives needed to formalize protocols, no more. This makes it possible to analyze protocols without having to confront complications that might be inherent in a formalism but not intrinsic to protocols themselves. In particular, we are able to prove lower bounds on protocol analysis that seem faithful to the notion of finite-length protocol. Since the formalism is based on linear logic, we have a direct method for describing state changes. In contrast, axiomatizing protocol traces in higher-order logic requires rather complex conditions to guarantee that when a principal moves from one state to another, the previous state is no longer available for further use. In comparison with using CSP [18, 28, 29] or Murφ [25], our notation allows us to describe unbounded runs of the protocol simply, recovering bounded instances of th protocol by restricting the uses of existential quantification. Perhaps the closest formalism is spi-calculus [1]. However, there are several characteristics of the underlying pi-calculus, such as creation of new channels, that are absent from our system, making it simpler to prove meta-theoretic results. Finally, we should make clear that the formalism presented in this paper is not intended to be a logic for specifying and reasoning about protocols, only a notation for defining the behavior of a protocol in the face of network intruder.

We have found the linear-logic tool *LLF* useful for reasoning about protocols. As a first step toward developing an algorithmic meta-theory of protocols, we have proved correctness of a protocol/intruder optimization using this tool. We expect to develop and verify additional optimizations and transformations. In addition to meta-results, *LLF* also seems useful for searching the possible runs of a protocol and intruder. While we do not expect pure *LLF* to be more efficient or more comprehensive than tools such as the NRL analyzer [21] that have been refined over many years, there is some benefit in the simplicity of the tool. If we can improve our search-strategy language to make it more effective in common cases, then this may lead to better understanding of general techniques that could be adopted in a variety of settings.

Finally, although not discussed in this paper, we believe that there may be some advantage in developing translations between the notation used here and other standard protocol formalisms. We believe the translation into Murφ is straightforward and easy to implement, as is a translation into process calculus with ν operator [12]. On this basis, we believe there may be practical translations into CSP, for example, based on choosing bounds on the number of new values that are to be generated in any run. It also seems feasible to translate protocol definition languages such as CASPER [19] and CAPSL [22] into this formalism, making it possible to use our framework as an intermediate language in some kind of federated protocol analysis environment.

Acknowledgments: Thanks to Grit Denker, Jon Millen, Mourad Debbabi, Mark Mitchell, Vitaly Shmatikov and Ulrich Stern for helpful discussions.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] J.-P. Banâtre and D. L. Métayer. Computing by multiset transformation. *Communcations of the ACM*, 36:98–111, 1993.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] I. Cervesato and F. Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science — LICS'96*, pages 264–275, New Brunswick, NJ, July 1996. IEEE Computer Society Press.

[6] J. Clark and J. Jacob. A survey of authentication protocol literature. Web Draft Version 1.0 available from `http://www.cs.york.ac.uk/ jac/papers/drareviewps.ps`, 1997.

[7] E. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[8] K. Compton and S. Dexter. Proving authentication protocols in a fragment of linear logic. Manuscript, 1998.

[9] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

[10] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.

[11] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[12] N. Durgin and J. Mitchell. Analysis of security protocols. To appear in the Proceedings of the 1998 International Summer School of Marktoberdorf, 1999.

[13] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[14] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[15] J. Harland, D. Pym, and M. Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, pages 391–405, Munich, Germany, July 1996. Springer-Verlag LNCS 1101.

[16] M. Kanovich, M. Okada, and A. Scedrov. Specifying real-time finite-state systems in linear logic. In *COTIC '98: Second Workshop on Concurrent Constraint Programming for Time-Critical Applications and Multi-Agent Systems*, Nice, France, 1998. Electronic Notes in Theoretical Computer Science, Volume 16, Issue 1. http://www.elsevier.nl/locate/entcs.

[17] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

[18] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[19] G. Lowe. Casper: a compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 18–30, 1997.

[20] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: a comparison of two approaches. In *Proc. European Symposium On Research In Computer Security*. Springer Verlag, 1996.

[21] C. Meadows. The NRL protocol analyzer: an overview. *J. Logic Programming*, 26(2):113–131, 1996.

[22] J. Millen. CAPSL: Common authentication protocol specification language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[23] D. Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[24] J. Mitchell. Analysis of security protocols. Slides for invited talk at CAV '98, available at `http://www.stanford.edu/~jcm`, July 1998.

[25] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proc. IEEE Symp. Security and Privacy*, pages 141–151, 1997.

[26] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[27] L. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.

[28] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Soc Press, 1995.

[29] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[30] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.

Initialization Theory $\mathcal{I}$:

GOODGUY:		$\to$	$\exists k_e.k_d.GoodGuy(k_e,k_d), KP(k_e,k_d)$
BADKEY:		$\to$	$\exists k_e.k_d.KP(k_e,k_d), BadKey(k_e,k_d)$
ROLA:	$GoodGuy(k_e,k_d)$	$\to$	$GoodGuy(k_e,k_d), A_0(k_e)$
ROLB:	$GoodGuy(k_e,k_d)$	$\to$	$GoodGuy(k_e,k_d), B_0(k_e)$
ANNK:	$GoodGuy(k_e,k_d)$	$\to$	$AnnK(k_e), GoodGuy(k_e,k_d)$
ANNKB:	$BadKey(k_e,k_d)$	$\to$	$AnnK(k_e), BadKey(k_e,k_d)$

Protocol Theories $\mathcal{A}$ and $\mathcal{B}$:

A1:	$AnnK(k'_e), A_0(k_e)$	$\to$	$\exists x.A_1(k_e,k'_e,x), N_{S1}(enc(k'_e,\langle x,k_e\rangle)), AnnK(k'_e)$
A2:	$A_1(k_e,k'_e,x), N_{R2}(enc(k_e,\langle x,y\rangle))$	$\to$	$A_2(k_e,k'_e,x,y), N_{S3}(enc(k'_e,y))$
B1:	$B_0(k_e), N_{R1}(enc(k_e,\langle x,k'_e\rangle)), AnnK(k'_e)$	$\to$	$\exists y.B_1(k_e,k'_e,x,y), N_{S2}(enc(k'_e,\langle x,y\rangle)), AnnK(k'_e)$
B2:	$B_1(k_e,k'_e,x,y), N_{R3}(enc(k_e,y))$	$\to$	$B_2(k_e,k'_e,x,y)$

A Example: Needham-Schroeder Public Key Protocol

As an example, we give the full theory of the three-step core of the Needham-Schroeder public-key protocol.

$$\begin{array}{ccccl} A & \longrightarrow & B & : & \{A, N_a\}_{K_b} \\ B & \longrightarrow & A & : & \{N_a, N_b\}_{K_a} \\ A & \longrightarrow & B & : & \{N_b\}_{K_b} \end{array}$$

Note that this is a simplified version of the protocol, where the use of a trusted server to distribute the public keys is omitted. This results in a particularly straightforward representation for the Initialization and Protocol Theories.

The Initialization Theory is shown in Table 4. Here, the predicate *GoodGuy* indicates an uncompromised principal, parameterized by its encryption and decryption (public and private) keys. For simplicity, we identify the principal with its public key (i.e. where "A" appears in the protocol, we use the public key "K_a"). The GOODGUY rule allows for the creation of an unlimited number of principals, each with a unique key pair, denoted by the predicate *KP*.

The BADKEY rule provides a mechanism for specifying an unlimited number of compromised key pairs, which appear to belong to valid principals, but whose private keys are known to the intruder. The predicate *BadKey* denotes these compromised key pairs.

ROLA and ROLB allow an unlimited number of sessions to be started for any principal to act in the role of either "Alice" (the initiator) or "Bob" (the responder). A_0 and B_0 denote the initial role state for the A and B roles, respectively, parameterized by the public key (principal) acting in that role.

Since we are omitting the trusted server from this example, we accomplish key distribution by having the principals announce their keys. The ANNK rule accomplishes this for the *GoodGuy* participants, while the ANNKB rule does the same for the *BadKey* pairs. Note that both rules generate a predicate *AnnK* indicating a public key that is available for communication, so from this point the valid participants can't distinguish the good guys from the bad guys.

Note that in this initialization theory, all predicates are persistent except for initial role states A_0 and B_0.

The protocol theories, shown in Table 4, are derived directly from the specification of the Needham-Schroeder protocol. Theory $\mathcal{A}$ corresponds to the role of "Alice", and theory $\mathcal{B}$ corresponds to "Bob".

In rule A1, which corresponds to the first line of the protocol, a principal k_e, in its initial state A_0, decides to talk to another principal k'_e, whose key has been announced. A new nonce x is generated, along with a network message N_{S1} corresponding to the first message sent in the protocol, and the principal moves to the new state A1, remembering the values of x and k'_e. Note that since *AnnK* is persistent, it must also appear on the right hand side of the rule.

In step B1, corresponding to the second step of the protocol, a principal k_e, in the initial state B_0, responds to a message on the network which is of the expected format (i.e. encrypted with k_e's public key, and with the identity of a participant whose key has been announced, embedded inside). k_e generates another nonce, and replies to the message, moving to a new state B_1 where all the information (the two nonces and the two principals) is remembered.

Similarly, A2 corresponds to the third line of the protocol, and B2 corresponds to the implicit step where the responder actually receives the final message.

Note that sent messages are denoted by N_{Si} and received messages are denoted by a corresponding predicate N_{Ri}. The intruder theory, described in the next section, can be thought of as providing a network that, at a minimum, transforms N_{Si}'s to N_{Ri}'s, so the protocol can execute.

The Intruder Theory is shown in Table 5. Here the M predicate denotes persistent facts known to the intruder, while D and C represent non-persistent facts which can be decomposed and composed into other facts.

LRNKB is an initialization rule that allows the intruder to learn any Bad Keys. Since *BadKey* predicates are generated only by the initialization theory, we know from Lemma 1 that this rule only needs to be applied once per derivation, per *BadKey* fact.

The REC and SND rules are used to connect the intruder to the network being used by the participants. The REC rule intercepts a message from the network and saves it as a decomposable fact. The SND rule sends composed facts onto the network.

The COMP rule allow the user to compose small terms into larger ones, while the DCMP rule allows for decomposition of large terms into smaller ones..

LRN converts a decomposable fact into intruder knowledge, and USE converts intruder knowledge into a composable fact.

The ENC and DEC rules allow the intruder to decrypt a message if it knows the private key, and to generate encrypted message from known public keys.

Note that LRNA and DECA are decomposition rules with auxilliary facts that handle a special case for encrypted messages. If the message can't be decrypted because the key isn't currently known, LRNA remembers the decrypted message with the special "Auxilliary" predicate, A. The DECA rule allows Auxilliary messages to be decrypted at a later time, if the decryption key becomes known.

Finally, GEN allows the intruder to generate new facts (i.e. nonces) as needed.

This Intruder Theory can be divided into Composition and Decomposition rules, as shown in Table 5. So, this is a Two-Phase Intruder Theory, as described in Section 3.3.

B Horn Clauses with Existential Quantification

An *existential Horn clause* is a closed first-order formula of the form

$$\begin{aligned}&\forall x_1 \ldots \forall x_i [(\alpha_1 \wedge \ldots \wedge \alpha_k)\\ &\quad \Longrightarrow \exists y_1 \ldots \exists y_j (\beta_1 \wedge \ldots \wedge \beta_\ell)]\end{aligned}$$

where $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_\ell$ are first-order atomic formulas. Without restriction on the form of the atomic formulas, undecidability of the implication problem for existential Horn clauses follows immediately from the undecidability of Horn clauses without existential quantifiers. The problem of interest to us, however, is implication when the atomic formulas contain no function symbols.

The formulas we are interested in are a special case of database dependencies [2]. However, while database dependencies include existential quantification and do not involve function symbols, database dependencies also allow equality in the conclusions of Horn clauses. Since we do not use equality in our protocols (except by pattern matching in the hypotheses of rules), it is not immediately clear to us whether the following theorem is a consequence of standard results in database dependency theory.

Theorem 5. *The implication problem for existential Horn clauses without function symbols is undecidable. In particular, there is no algorithm for deciding whether a set of existential Horn clauses without function symbols implies a single atomic formula* $A(b_1, \ldots, b_k)$ *without function symbols or variables.*

This theorem has a straightforward direct proof based on axiomatizing a Cook's-theorem-style Turing machine tableau. Specifically, given any Turing machine, we write axioms of the form

$$\begin{aligned}&\forall x, y, z.[(\mathsf{Adj}(x, y) \wedge \mathsf{Adj}(y, z) \wedge\\ &\mathsf{Cont}(x, 0) \wedge \mathsf{Cont}(y, 1.q_i) \wedge \mathsf{Cont}(z, 1))\\ &\Longrightarrow \exists x', y', z'((\mathsf{Adj}(x', y') \wedge \mathsf{Adj}(y', z') \wedge\\ &\mathsf{Below}(x', x) \wedge \mathsf{Below}(y', y) \wedge \mathsf{Below}(z', z)) \wedge\\ &\mathsf{Cont}(x', 0.q_j) \wedge \mathsf{Cont}(y', 0) \wedge \mathsf{Cont}(z, 1))\end{aligned}$$

In this formula, the variables represent cells of the Turing machine tableau (i.e., cells of the tape at some stage

Initialization Rules:

LRNKB:	$BadKey(k_e, k_d)$	$\rightarrow$	$M(k_e), M(k_d), BadKey(k_e, k_d)$

I/O Rules:

REC:	$N_{Si}(x)$	$\rightarrow$	$D(x)$
SND:	$C(x)$	$\rightarrow$	$N_{Ri}(x)$

Decomposition Rules:

DCMP:	$D(\langle x, y\rangle)$	$\rightarrow$	$D(x), D(y)$
LRN:	$D(x)$	$\rightarrow$	$M(x)$
DEC:	$M(k_d), KP(k_e, k_d), D(enc(k_e, x))$	$\rightarrow$	$M(k_d), KP(k_e, k_d), D(x), M(enc(k_e, x))$
LRNA:	$D(enc(k_e, x))$	$\rightarrow$	$M(enc(k_e, x)), A(enc(k_e, x))$
DECA:	$M(k_d), KP(k_e, k_d), A(enc(k_e, x))$	$\rightarrow$	$M(k_d), KP(k_e, k_d), D(x)$

Composition Rules:

COMP:	$C(x), C(y)$	$\rightarrow$	$C(\langle x, y\rangle)$
USE:	$M(x)$	$\rightarrow$	$C(x), M(x)$
ENC:	$M(k_e), C(x)$	$\rightarrow$	$C(enc(k_e, x)), M(k_e)$
GEN:		$\rightarrow$	$\exists x.M(x)$

of the computation); a nice picture of the tableau we use appears in [30, page 255]. The constants 0 and 1 indicate symbols in these cells, and constants of the form $0.q_i$ or $1.q_j$ indicate that the cell contains a symbol 0 or 1 and is the location of the tape head, with machine in state q_i or q_j (respectively). A fact $\mathsf{Adj}(x, y)$ means that cell x is adjacent to y, $\mathsf{Below}(x', x)$ that cell x' is below cell x, and $\mathsf{Cont}(x, c)$ that the cell x has contents described by constant c, possibly giving the machine state in addition to the symbol contained in the cell. The atomic formula $A(b_1, \ldots, b_k)$ mentioned in the statement of the theorem can be an atomic formula that is derivable by a rule that requires, in its hypothesis, that the Turing machine is in a halting state.