Encryption as an abstract data-type: An extended abstract

Dale Miller

INRIA/Futurs & École polytechnique, France

Abstract

At the Dolev-Yao level of abstraction, security protocols can be specified using multisets rewriting. Such rewriting can be modeled naturally using proof search in linear logic. The linear logic setting also provides a simple mechanism for generating nonces and session and encryption keys via eigenvariables. We illustrate several additional aspects of this direct encoding of protocols into logic. In particular, encrypted data can be seen naturally as an abstract data-type. Entailments between security protocols as linear logic theories can be surprisingly strong. We also illustrate how the well-known connection in linear logic between bipolar formulas and general formulas can be used to show that the asynchronous model of communication given by multiset rewriting rules can be understood, more naturally as asynchronous process calculus (also represented directly as linear logic formulas). The familiar proof theoretic notion of interpolants can also serve to characterize communication between a role and its environment.

1 Introduction

When the topic of specifying and reasoning about security protocols attracts the attention of programming language researchers, it is common to find them turning to process calculi, such as CSP or the spi-calculus [27,18], automata, and even typed λ-calculus [29]. Proof search (that is, the foundations of logic programming), however, has a number of properties that are attractive when one attempts to specify and analyze security protocols. We list a few of these advantages here and devote the rest of this paper to develop these benefits further.

Formal analysis of security protocols is largely based on a set of assumptions commonly referred to as the Dolev-Yao model [10], an abstraction that supports symbolic execution and reasoning. It has been observed in various places (for example, [9,4]) that this abstract can be realized well using multiset rewriting. Given that it is well-known that proof search in linear logic provides a declarative framework for specifying multiset rewriting [12,15,4], a

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

rather transparent start at representing security primitives is available with proof search in linear logic.

Proof theory comes with a primitive, declarative, and well understood notion of "newness" and "freshness" via the technical devices of *eigenvariable* (newness in proofs) [13] and λ-abstraction [7] (newness in terms). It is appealing to try to use these devices rather than the more *ad hoc* approach to freshness, say, in the BAN logic [2]. Proof search has been used successfully to specify transitions in the π-calculus [26]: *ad hoc* provisos needed to make certain that names and scope were maintained correctly can be encoded in a meta-logic by mapping names to term-level abstractions (λ-abstractions) and to proof-level abstractions (eigenvariables) [25].

Proof search in linear logic has been studied extensively and proof systems exist that give strong normal forms to how search can actually proceed [1,23]. As a result, significant information is known about the structure of cut-free proofs: since these are the "traces" of protocols and attackers, this structural information is a great aid in reasoning about possible computation paths. The normal forms also provide blueprints for building high-level interpreters for linear logic specifications. Furthermore, the foundations of proof search is a current topic of active research [14] and we can expect that advances there will find applications to this particular domain of computer science.

The proof search paradigm comes equipped with notions of abstract datatypes [21] and higher-order predicate abstractions [28], all features that we shall draw upon in this paper. Given that these abstraction mechanism all result from aspects of logic, there is no problem in understanding the interaction between abstractions and other aspects of the logic (for example, with multiset rewriting). The problems surrounding "feature interaction" are not present in well designed logics.

Finally, logic comes with a built-in notions of entailment and of equivalence and it is interesting to see if these notions can be useful in reasoning about computation. This notion of logical equivalence would, for example, replace the notion of structural equivalence in process calculus [22]. When one allows rich forms of higher-order quantification, logical entailment is able to relate surprising different specifications.

2 Multiset rewriting in proof search

To model multiset rewriting we shall use a subset of linear logic called *process clauses* in [22]. The closed formula $\forall \bar{x}[G \multimap H]$ is a *process clause* (*clause* for short) if G and H are formulas composed of $\bot$, $⅋$, and $\forall$, and all free variables of G are free in H. Here, $⅋$ encodes the multiset constructor and $\bot$ encodes the empty multiset, H is the *head* of the clause, and G is the *body* of the clause. We will follow the common practice from the logic programming literature of writing the head of a clause first by reversing the sense of the implication: that is, the clause above is written as $\forall \bar{x}[H \circ\!- G]$. Kanovich [16,17] introduced a similar set of formulas he calls *linear Horn clauses*: these are essentially process clauses in contrapositive form (replace $\parr$, $\forall$, $\bot$, and $\multimapinv$ with $\otimes$, $\exists$, $\mathbf{1}$, and $\multimap$, respectively). We avoid using Kanovich's term here since it is contrary to the usual use of the term *Horn clause* in the logic programming literature where eigenvariables are not part of proof search.

Using simple linear logic equivalences, process clauses can be written in normal form as

$$\forall x_1 \ldots \forall x_i [a_1 \parr \cdots \parr a_m \multimapinv \forall y_1 \ldots \forall y_j [b_1 \parr \cdots \parr b_n]]$$

where $i, j, n, m \geq 0$ and $a_1, \ldots, a_m, b_1, \cdots, b_n$ are atoms. It is in this form that we generally present process clauses.

In this abstract, we choose sequents of the form $\Sigma : \Psi \longrightarrow \Gamma$ where Σ is the signature (declaration of eigenvariables) and Ψ and Γ are multisets of formulas (linear/bounded maintenance). To illustrate multiset rewriting using the right-hand context, consider the rewriting rule

$$a, b \Rightarrow c, d, e,$$

which specifies that a multiset should be rewritten by removing one occurrence each of a and b and then have one occurrence each of c, d, and e added.

To rewrite the right-hand multiset using the rule above, simply backchain over the clause $c \parr d \parr e \multimap a \parr b$: the full details of such a backchaining looks like the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma : \Psi \longrightarrow c, d, e, \Gamma}{\Sigma : \Psi \longrightarrow c, d \parr e, \Gamma}}{\Sigma : \Psi \longrightarrow c \parr d \parr e, \Gamma} \qquad \dfrac{\dfrac{}{\Sigma; \cdot \xrightarrow{a} a} \quad \dfrac{}{\Sigma; \cdot \xrightarrow{b} b}}{\Sigma; \cdot \xrightarrow{a \parr b} a, b}}{\Sigma; \Psi \xrightarrow{c \parr d \parr e \multimap a \parr b} a, b, \Gamma}}{\Sigma : \Psi, c \parr d \parr e \multimap a \parr b \longrightarrow a, b, \Gamma}$$

Left-introduction rules are applied to the formula on the sequent arrow (instead of general formulas in the left-hand context). The sub-proofs on the right are responsible for deleting from the right-hand context one occurrence each of the atoms a and b while the subproof on the left is responsible for inserting one occurrence each of the atoms c, d, and e and continuing the computation. Thus, the proof fragment above will be simply written as

$$\dfrac{\Sigma : \Psi \longrightarrow c, d, e, \Gamma}{\Sigma : \Psi, c \parr d \parr e \multimap a \parr b \longrightarrow a, b, \Gamma.}$$

We interpret this fragment of a proof as a rewriting of the multiset a, b, Γ to the multiset c, d, e, Γ.

Backchaining over the clause

$$\forall x_1 \ldots \forall x_i [a_1 \parr \cdots \parr a_m \multimapinv \forall y_1 \ldots \forall y_j [b_1 \parr \cdots \parr b_n]]$$

is equivalent to the following simple inference rule

$$\frac{\Sigma, y_1, \ldots, y_j : \Psi \longrightarrow \theta b_1, \ldots, \theta b_n, \Gamma}{\Sigma : \Psi \longrightarrow \theta a_1, \ldots \theta a_m, \Gamma}$$

Here, θ is a substitution mapping the variables $x_1, \ldots, x_n$ to Σ-terms and the variables $y_1, \ldots, y_m$ are not declared in Σ (otherwise we can use α-conversion on the clause before backchaining).

Example 2.1 Consider the problem of Alice wishing to communicate a value to Bob. The clause

$$\forall x[(a\ x) \parr b \multimapinv a' \parr (b'\ x)]$$

illustrates how one might synchronize Alice's role $(a\ x)$ with Bob's role b. In one, atomic step, the synchronization occurs and the value x is transfered from Alice, resulting in her continuation a', to Bob, resulting in his continuation $(b'\ x)$. If a server is also involved, one can imagine the clause being written as

$$\forall x[(a\ x) \parr b \parr s \multimapinv a' \parr (b'\ x) \parr s],$$

assuming that the server's state is unchanged through this interaction. As in most examples in this abstract, we shall assume the familiar Prolog convention of writing top-level implications in specification clauses in their reverse direction.

As this example illustrates, synchronization between roles is easy to specify and can trivialize both the nature of communication and the need for security protocols entirely. (For example, if such secure communications is done atomically, there is no need for a server s in the above clause.) While the clause in this example might *specify* a desired communication, it cannot be understood as capturing more low-level and realistic aspects of communications. In distributed settings, synchronization actually only takes place between roles and networks. Our main use of multiset rewriting will involve more restricted clauses with weaker assumptions about communications: these will involve synchronizations between roles and network messages (a model for asynchronous communications) and not between roles and other roles (a model of synchronous communications). Clauses such as those displayed above, however, can serve as *specifications* of what a given protocol might be shown to satisfy.

3 Encoding security protocols

The following encoding of security protocols follows to a large extent that given in [4,5] for the system MSR.

The type i is used to encode messages. Primitive objects, such as integers, strings, and nonces, are all constructors of i. The tupling operator $\langle \cdot, \cdot \rangle$, for pairing data together, has type $i \to i \to i$. Expressions such as $\langle \cdot, \cdot, \ldots, \cdot \rangle$

denote pairing associated to the right. One additional constructor for i is presented in Section 4.

A *network message* is encoded as an atomic formula of the form $\mathsf{N}(t)$, where $\mathsf{N}(\cdot)$ is a predicate of type $i \to o$ and t (of type i) is the actual data of the message. (Following [6], we use o to denote the type of formulas.) As we shall see, the state of the network will be a multiset of such atomic formulas.

The roles (e.g., Alice and Bob) and the formulas that describe them are rather complicated in MSR: we provide some definitions inspired by the definitions given in [4,5]. A *role identifier* is a symbol, say, ρ. For some number $n \geq 1$ and for $i = 1, \ldots, n$, the pair ρ_i of an identifier and an index is a *role state predicate*. These state predicates are used to encode internal states of a role as a protocol progresses. A *role state atom* is an atomic formula of the form $\rho_i(x_1, \ldots, x_m)$ where $x_1, \ldots, x_m$ are distinct variables and ρ_i is a role state predicate. A *role clause* is a process clause

$$\forall x_1 \ldots \forall x_i [a_1 \parr \cdots \parr a_m \multimapinv \forall y_1 \ldots \forall y_j [b_1 \parr \cdots \parr b_n]]$$

where $i, j, n, m \geq 0$. Here, the *head* of such as clause is the formula $a_1 \parr \cdots \parr a_m$ and the *body* is $\forall y_1 \ldots \forall y_j [b_1 \parr \cdots \parr b_n]$. Role clauses also have the following restrictions: all the atoms $a_1, \ldots, a_m, b_1, \ldots, b_n$ are either network messages or a role state atoms such that (1) there is at most one role state atom in the head and at most one in the body; (2) if there is a role state atom in the head, say, $\rho_i(\bar{t})$ and a role state atom in the body, say, $\rho'_j(\bar{s})$, then ρ and ρ' must be the same role identifier and $i < j$. In other words, a role clause only involves a single role (and possibly network messages) and when moving from the head to the body, the index of the role must increase. As a consequence of the restrictions on role clauses, roles cannot synchronize with other roles directly and one role cannot evolve into another role. Condition (1) allows for process creation (no role state atom in the head) and process deletion (no role state atom in the body). Condition (2) above implies is that all agents have finite runs [4,5]. A final restriction on role clauses is that all variables free in the body of the clause must be free in the role state atom in the head of the clause.

A *role theory* is a linear logic formula of the form

$$\exists x_1 \ldots \exists x_r \; [C_1 \otimes \cdots \otimes C_s],$$

where $r, s \geq 0$, $C_1, \ldots, C_s$ are role clauses, where $x_1, \ldots, x_r$ are variables of type i or $i \to i$, and whenever C_i and C_j have the same role state predicate in their head then $i = j$. This latter condition implies that agents in protocols are deterministic. This is a condition that can easily be relaxed within linear logic if non-deterministic agents are of interest. Fortunately, this restriction does not restrict the class of intruders (testers) that we consider briefly in Section 7 since intruders can be grouped together in multisets and interleaving provides non-determinism.

Existential quantification like that surrounding role theories are used in logic programming to provide for abstract data-types and here they will serve as local constants shared by certain role clauses. In particular, shared keys between, say Alice and a trusted server, will be existentially quantified in this way with a variable of type $i \to i$. The use of existential quantifier at type $i \to i$ is explained next.

4 Encryption as an abstract data-type

Encryption keys will be encoded using function symbols of type $i \to i$. Since such keys will need to be given scope, they will be quantified either existentially over role theories or universally in role clauses. Using higher-order quantification over data constructors is the usual way to specify *abstract data-types* within logic programming [21]. Since we will be allowing quantification of higher-order type, proof search will be slightly more complicated than if we restricted ourselves to only first-order quantification. For example, the (meta-level) equations for α, β, and η conversions are assumed, although no other equations on the type i are assumed. (As is customary with typed λ-terms, η is assumed since there seems to be no good reason to distinguish, say, the encryption key k form the expression $(\lambda w.kw)$.) As discussed in [24], higher-order quantification can add greatly to the expressive strength of specification, but when done carefully, it does not need to add to the complexity of proof search. The remaining constructor for the type i is $\cdot^\circ$ of type $(i \to i) \to i$: this constructor is used to coerce an encryption key back into a data item, and in this way, a role can place a key into a network message. (We will not introduce application $app : i \to (i \to i)$, the dual operation to $\cdot^\circ$, since the expression $(app\ k^\circ\ x)$ will be written simply as $(k\ x)$.) This approach to "encryption as an abstract data-type" is a departure from MSR.

Consider the following specification that contains three occurrences of encryption keys.

$$\begin{array}{lrl} \exists k_{as} \exists k_{bs} [& a_1\langle M, S\rangle & \multimapinv a_2 S \parr \mathsf{N}(k_{as}\ M). \\ & b_1 T \parr \mathsf{N}(k_{bs}\ M) & \multimapinv b_2\langle M, T\rangle. \\ & s_1() \parr \mathsf{N}(k_{as}\ P) & \multimapinv \mathsf{N}(k_{bs}\ P). \quad] \end{array}$$

(Here as elsewhere, quantification of capital letter variables is universal with scope limited to the clause in which the variable appears.) In this example, Alice (a_1, a_2) communicates with Bob (b_1, b_2) via a server (s_1). To make the communications secure, Alice uses the key k_{as} while Bob uses the key k_{bs}. The server is deleted immediately after it translates one message encrypted for Alice to a message encrypted for Bob. The use of the existential quantifiers helps establish that the occurrences of keys, say, between Alice and the server and Bob and the server, are the only occurrences of that key. Even if more

$$\begin{array}{ll}
\text{Message 1} & A \longrightarrow S\text{: } A, B, n_A \\
\text{Message 2} & S \longrightarrow A\text{: } \{n_A, B, k_{AB}, \{k_{AB}, A\}_{k_{BS}}\}_{k_{AS}} \\
\text{Message 3} & A \longrightarrow B\text{: } \{k_{AB}, A\}_{k_{BS}} \\
\text{Message 4} & B \longrightarrow A\text{: } \{n_B\}_{k_{AB}} \\
\text{Message 5} & A \longrightarrow B\text{: } \{n_B, Secret\}_{k_{AB}}
\end{array}$$

Fig. 1. A conventional presentation of the Needham-Schroeder protocol.

$$\begin{array}{rcl}
\multicolumn{3}{l}{\exists k_{as} \exists k_{bs} \{} \\
a_1(S) & \multimap & \forall na.\ a_2(na, S) \parr \mathsf{N}(\langle alice, bob, na \rangle). \\
a_2(N, S) \parr \mathsf{N}(k_{as}\langle N, bob, K^\circ, En\rangle) & \multimap & a_3(N, K, S) \parr \mathsf{N}(En). \\
a_3(Na, Key, S) \parr \mathsf{N}(Key\ Nb) & \multimap & a_4() \parr \mathsf{N}(Key\ \langle Nb, S\rangle). \\
b_1() \parr \mathsf{N}(k_{bs}\ \langle Key^\circ, alice\rangle) & \multimap & \forall nb. b_2(nb, Key) \parr \mathsf{N}(Key\ nb). \\
b_2(Nb, Key) \parr \mathsf{N}(Key\langle Nb, S\rangle) & \multimap & b_3 S. \\
s_1 \parr \mathsf{N}(\langle alice, bob, N\rangle) & \multimap & \forall k. \mathsf{N}(k_{as}\langle N, bob, k^\circ, k_{bs}\langle k^\circ, alice\rangle\rangle). \\
 & & \}
\end{array}$$

Fig. 2. Encoding the Needham-Schroeder protocol.

principals are added to this system, these occurrences are still the only ones for these keys. Thus, the existential quantifier helps in determining the static or lexical scope of key distribution. Of course, as protocols are evaluated (that is, a proof is searched for), keys may extrude their scope and move freely onto the network. This dynamic notion of scope extrusion is similar to that found in the π-calculus [26] and is modeled here similar to an encoding of the π-calculus into linear logic found in [22].

Example 4.1 The Needham-Schroeder Shared Key protocol presented in Figure 1 [30] and the rough translation of it into linear logic given in Figure 2 provides another example. Notice that two shared keys are used in this example and that the server creates a new key that is placed within data and is then used by Alice and Bob to communicate directly. It is a simple matter to show that this protocol implements the specification (taken from Example 2.1):

$$\forall x[a_1(x) \parr b_1() \parr s_1() \multimap a_4() \parr b_3(x)].$$

The linear logic proof of this starts with the multiset $a_1(c)$, $b_1()$, $s_1()$ on the right of the sequent arrow (for some "secret" eigenvariable c) and then reduces this back to the multiset $a_4() \parr b_3(c)$ simply by performing a simple "execution" of the logic program in Figure 2. Notice that the $\forall$ used in the bodies of clauses in this protocol are used both for nonce creation (at type i) and encryption key creation (at type $i \rightarrow i$).

Example 4.2 Consider the following two clauses for Alice.

$$aK^\circ \parr \mathsf{N}(K\ M) \multimapinv a'M. \qquad (3.1)$$

$$a \parr \mathsf{N}(K\ M) \multimapinv a'M. \qquad (3.2)$$

In the first case, she possesses an encryption key and uses it to decrypt a network message. In the second case, it appears that she is decrypting a message without knowing the key, an inappropriate behavior, of course. Technically, this clause is not a proper role clause (since there is a variable M that is not free in the role state predicate in the head a). In any case, it is interesting to consider (3.2) for a moment. Notice that (3.2) is logically equivalent (and, hence, operationally indistinguishable using proof search) to both of the formulas

$$\forall M \forall X[a \parr \mathsf{N}(X) \multimapinv a'M] \quad \text{and} \quad \forall X[a \parr \mathsf{N}(X) \multimapinv \exists M.a'M].$$

This last clause clearly illustrates that Alice is not actually decoding an existing message but is simply guessing (using $\exists$) at some data value M, and continues with that guess as $a'M$. If one thinks operationally instead of declaratively about proof search involving clause (3.2), we would consider possible unifiers for matching the pattern $(K\ M)$ with a network message, say, $(k\ s)$, for two constants k and s. Unification yields exactly the following three different unifiers:

$$[M \mapsto (k\ s), K \mapsto \lambda w.w] \quad [M \mapsto s, K \mapsto k] \quad [M \mapsto M, K \mapsto \lambda w.(k\ s)]$$

Thus, M can be bound to either $(k\ s)$ or s or any term: in other words, M can be bound to any expression of type i.

By using higher-order quantification, logical entailment is strengthened and can help in reasoning about role clauses and theories.

Example 4.3 Consider the two clauses

$$a_1 \multimapinv \forall k.\mathsf{N}(k\ m) \quad \text{and} \quad a_1 \multimapinv \forall k.\mathsf{N}(k\ m').$$

Each of these clauses specify that Alice can take a step that generates a new encryption key and then outputs a message (either m or m') using that encryption key. Since Alice has no continuation, no one, not even Alice will be able to decode this message. It should be the case that these two clauses are "operationally" similar since they both generate a "junk message." In fact, it is an easy matter to show that these two clauses are logically equivalent. A proof that the first implies the second contains a subproof of the sequent

$$\forall k.\mathsf{N}(k\ m') \longrightarrow \forall k.\mathsf{N}(k\ m),$$

and this is proved by introducing an eigenvariable, say c, on the right and the term $\lambda w.(c\ m)$ on the left.

5 Abstracting internal states

The following example illustrates that using existential quantification over *predicates* (in particular, role state predicates) allows interesting rewriting of the structure of role theories.

Example 5.1 [Reducing n-way to 2-way synchronization] General n-way synchronization ($n \geq 2$) can be rewritten using 2-way synchronization by the introduction of new, intermediate, and hidden predicates as is allowed in role theories. For example, the following two formulas are logically equivalent.

$$\exists l_1 \exists l_2. \left\{ \begin{array}{c} a \parr b \multimapinv l_1 \\ l_1 \parr c \multimapinv l_2 \parr e \\ l_2 \multimapinv d \parr f \end{array} \right\} \quad \dashv\vdash \quad a \parr b \parr c \multimapinv d \parr e \parr f$$

The clause on the right specifies a 3-way synchronization and the spawning of 3 new atoms whereas the formula on the left is limited to rewriting at most two atoms into at most 2 atoms. The proof of the forward entailment in linear logic is straightforward while the proof of the reverse entailment involves the two higher-order substitutions of $a \parr b$ for $\exists l_1$ and $d \parr f$ for $\exists l_2$. As long as we are using logical entailment, these two formulas are indistinguishable and can be used interchangeably in all contexts. If instead we could observe possible failures in the search for proofs, then it is possible to distinguish these formulas: consider the search for a proof of a sequent containing a and b but not c. Since linear logic does not observe such failures, this kind of observation cannot be internalized.

Existential quantification over program clauses can also be used to hide predicates encoding roles. In fact, one might argue that the various restrictions on sets of process clauses (no synchronization directly with atoms encoding roles, no role changing into another role, etc) might all be considered a way to enforce locality of predicates. Existential quantification can, however, achieve this same notion of locality, but much more declaratively.

Example 5.2 [Hiding role state predicates] The following two formulas are logically equivalent:

$$\exists\ a_2, a_3. \left\{ \begin{array}{l} a_1 \parr \mathsf{N}(m_0) \multimapinv a_2 \parr \mathsf{N}(m_1) \\ a_2 \parr \mathsf{N}(m_2) \multimapinv a_3 \parr \mathsf{N}(m_3) \\ a_3 \parr \mathsf{N}(m_4) \multimapinv a_4 \parr \mathsf{N}(m_5) \end{array} \right\} \quad \dashv\vdash$$

$$a_1 \parr \mathsf{N}(m_0) \multimapinv (\mathsf{N}(m_1) \multimapinv (\mathsf{N}(m_2) \multimapinv (\mathsf{N}(m_3) \multimapinv (\mathsf{N}(m_4) \multimapinv (\mathsf{N}(m_5) \parr a_4)))))$$

The changing of polarity that occurs when moving to the premise of a $\multimapinv$ flips expressions from output (e.g., $\mathsf{N}(m_1)$) to input (e.g., $\mathsf{N}(m_2)$), etc. Thus, by hiding intermediate roles state predicates, it is possible to rewrite a role theory into a different style of formula that seems quite natural.

6 Asynchronous and synchronous connectives

The observation that abstracting over internal states results in an equivalent syntax with nested $\circ\!-$ suggests an alternative syntax for roles. Consider the following syntactic categories of linear logic formulas:

$$H ::= A \mid \bot \mid H \parr H \mid \forall x.H$$

$$K_O ::= H \mid H \circ\!- K_I \mid \forall x.K_O \qquad K_I ::= H \circ\!- K_O \mid \forall x.K_I$$

Here, A denotes the class of atomic formulas encoding network messages and formulas belonging to the class H denote bundles of messages that are used as either input or output to the network. Formulas belonging to the classes K_I and K_O can have deep nesting of implications and that nesting changes phases from input to output and back to input. The reason to split the class of K formulas into input formulas (K_I) and output formulas (K_O) is to parallel the MSR formalism more closely: in MSR, after a agent does an input, it must do an output (even if there is not messages to output). That is, in MSR, every input action has a continuation but not every output action as a continuation.

A formula in the category K_I is called a *role formula*.

The connectives of linear logic can be classified as asynchronous connective ($\parr$, $\forall$, &, etc) and synchronous connective ($\otimes$, $\exists$, $\oplus$, etc). The dual of a connective in one class is a connective in the other. The formulas of MSR are examples of *bipolar*: these are formulas in which no asynchronous connective is in the scope of a synchronous connective. For example, $Q_1 \parr \cdots \parr Q_m \circ\!- P_1 \parr \cdots \parr P_n$ is logically equivalent to $Q_1 \parr \cdots \parr Q_m \parr (P_1^\perp \otimes \cdots \otimes P_n^\perp)$. Obviously, role formulas are, in general, not bipolars.

Proposition 6.1 *For every role theory in which only the predicate* $\mathsf{N}(\cdot)$ *is free, there is a collection of role formulas to which it is provably equivalent.*

Proof. This proposition is proved by showing how to remove the existentially quantified role state predicate with maximal index by generating the appropriate higher-order substitution (similar to those produced in Example 5.1). When no more quantified role state predicates remain, the resulting theory is the desired collection of role formulas. $\Box$

To illustrate an example of this style of syntax, consider first declaring local all role predicates in the Needham-Schroeder Shared Key protocol in Figure 2. This then yields the logically equivalent presentation in Figure 3. There, three formulas are displayed: the first represents the role of Alice, the second Bob, and the final one the server. (All agents in this Figure are written at the same polarity, in this case, in output mode: since Bob and the server essentially start with inputs, these two agents are negated, meaning they first output nothing and then move to input mode.) Andreoli's compilation method [1] applied to the formula in Figure 3 yields the formulas in Figure 2: the new constants introduced by compilation are the names used to denote role continuation.

$$
\begin{array}{ll}
\text{(Out)} & \forall na.\mathsf{N}(\langle alice,\ bob,\ na\rangle) \multimapinv \\
\text{(In)} & \quad (\forall Kab \forall En.\mathsf{N}(kas\langle na,\ bob,\ Kab^{\circ},\ En\rangle) \multimapinv \\
\text{(Out)} & \qquad (\mathsf{N}(En) \multimapinv \\
\text{(In)} & \qquad\quad (\forall NB.\mathsf{N}(KabNB) \multimapinv \\
\text{(Out)} & \qquad\qquad (\mathsf{N}(Kab(NB,\ secret)))))).
\end{array}
$$

$$
\begin{array}{ll}
\text{(Out)} & \bot \multimapinv \\
\text{(In)} & \quad (\forall Kab.\mathsf{N}(kbs(Kab^{\circ},\ alice)) \multimapinv \\
\text{(Out)} & \qquad (\forall nb.\mathsf{N}(Kab\ nb) \multimapinv \\
\text{(In)} & \qquad\quad (\mathsf{N}(Kab(nb,\ secret)) \multimapinv \\
\text{(Cont)} & \qquad\qquad b\ secret))).
\end{array}
$$

$$
\begin{array}{ll}
\text{(Out)} & \bot \multimapinv \\
\text{(In)} & \quad (\forall N.\mathsf{N}(\langle alice,\ bob,\ N\rangle) \multimapinv \\
\text{(Out)} & \qquad (\forall k.\mathsf{N}(kas\langle N,\ bob,\ k^{\circ},\ kbs(k^{\circ},\ alice)\rangle))).
\end{array}
$$

Fig. 3. The roles of Alice, Bob, and a server

The style of specification given in Figure 3 is similar to that of process calculus: in particular, the implication $\multimapinv$ is syntactically similar to the dot prefix in, say, CCS. Universal quantification can appear in two modes: in output mode it is used to generate new eigenvariables (similar to the π-calculus restriction operator) and in input mode it is used for variable binding (similar to value-passing CCS). The formula $a \multimapinv (b \multimapinv (c \multimapinv (d \multimapinv k)))$ can denote processes described as

$$\bar{a} \mid\mid (b.\ (\bar{c} \mid\mid (d.\ \ldots))) \quad \text{or} \quad a.\ (\bar{b} \mid\mid (c.\ (\bar{d} \mid\mid \ldots)))$$

depending on which polarity it is being used. This formula and it's negation can also be written without linear implications as follows:

$$a \parr (b^{\perp} \otimes (c \parr (d^{\perp} \otimes \ldots))) \text{ resp, } \ a^{\perp} \otimes (b \parr (c^{\perp} \otimes (d \parr \ldots))).$$

Once a process with a continuation (that is, one that has an implication) has done an output (input), its continuation is an input (output) process. To see this mechanism in the proof search setting, consider a sequent $\Delta \longrightarrow \Gamma$ where Δ is a multset of K_I formulas and Γ are multisets of K_O formulas (here, we elide the signature associated to a sequent). The right-hand side of sequents involve asynchronous behavior (output) and left-hand side of sequents involve synchronous behavior (input). The two rules involving proof search with implications can be written as follows:

$$\frac{\Delta, K \longrightarrow \Gamma, H, \mathcal{A}}{\Delta \longrightarrow H \multimapinv K, \Gamma, \mathcal{A}} \qquad \frac{H \longrightarrow \mathcal{A}_1 \qquad \Delta \longrightarrow K, \mathcal{A}_2}{\Delta, H \multimapinv K \longrightarrow \mathcal{A}_1, \mathcal{A}_2}$$

Here, $\mathcal{A}$ denotes a multiset of atoms (i.e., network messages). Notice that we can assume that the left-introduction rule for $\circ\!-$ is only done when the right-hand side of the concluding sequent contains at most atomic formulas.

If the three formulas in Figure 3 are placed on the right-hand side of a sequent arrow (with no formulas on the left) then the role formula for Alice will output a message and move to the left-side of the sequent arrow (reading inference rules bottom up). Bob and the server output nothing and move to the left-hand side as well. At that point, the server will need to be chosen for a $\multimap$L inference rule, which will cause it to input the message that Alice sent and then move its continuation to the right-hand side. It will then immediately output another message, and so on.

Various equivalences familiar from the study of asynchronous communication are found in linear logic. For example, if one skips a phase, the two phases can be contracted as follows:

$$p \circ\!- (\bot \circ\!- (q \circ\!- k)) \equiv p \parr q \circ\!- k$$

$$p \circ\!- (\bot \circ\!- \forall x(q\ x \circ\!- k\ x)) \equiv \forall x(p \parr q\ x \circ\!- k\ x)).$$

While the nested presentation of roles is in some sense, more complicated syntax than the MSR (bipolar) format, this presentation certainly has its advantages over MSR. For example, there is only one predicate, namely $\mathsf{N}(\cdot)$, involved in writing out security protocols: role identifiers and role state predicates have disappeared. A role can now be seen as simply a formula and a role theory as simply an existentially quantified tensor of roles.

The following two examples illustrate a difference between the abstractions available in logic with those available in the π-calculus and the spi-calculus.

Example 6.2 [Comparison with the π-calculus] The π-calculus expression

$$(x)(\bar{x}m \mid x(y).Py)$$

is (weakly) bisimilar to the expression (Pm). This example is used to show that communication over a hidden channel provides no possible means for the environment to interact. A similar expression can be written as the following expression in linear logic:

$$\forall K[Km \parr (\forall x(Px \multimap Kx) \multimap \bot)].$$

Here, we have abstracted the *predicate* K: in a sense, we have abstracted the communication medium itself, and as such, the medium is available only for the particular purpose of communicating the message m from one process to another that is willing to do an input. This expression is logically equivalent to (Pm): the proof that (Pm) implies the displayed formula involves a use of equality (easy to add to the underlying logic in a number of ways) and the higher-order substitution $\lambda w.(w = m) \multimap \bot$ for K.

Example 6.3 [Comparison with the spi-calculus] In the spi-calculus, a "public" channel can be used for communicating. To ensure that messages are only "understood" by the appropriate parties, messages are encrypted with keys that are given specific scopes. For example, the expression

$$(k)(\bar{q}(\{m\}_k) \mid q(y).\text{let } \{x\}_k = y \text{ in } Px)$$

describes a process that is willing to output an encrypted message $\{m\}_k$ on a public channel q and to also input such a message and decode it. The key k is given a scope similar to that given in the π-calculus expression. The linear logic expression, call it B,

$$\forall k[\mathsf{N}(km) \parr (\forall x(Px \multimap \mathsf{N}(kx))) \multimap \bot]$$

is most similar to this spi-calculus expression: here, the network $\mathsf{N}(\cdot)$ corresponds to the public channel q. It is not the case, however, that B is logically equivalent to Pm since linear logic can observe that B can output something on the public channel, that is, $\forall y(\top \multimap \mathsf{N}(y)) \vdash B$ whereas it is not necessarily true that $\forall y(\top \multimap \mathsf{N}(y)) \vdash Pm$ is provable.

7 Interpolants and communications

We now illustrate how proof theory techniques can be applied to reasoning about security protocols.

Notice that role formulas, even collections of them, are not intended to be proved. In general, they are intended to evolve to other role formulas. To help analyze such evolution, we introduce *tests* which are essentially role formulas with an additional primitive for terminating proof search. In linear logic, the syntactic class of tests is written as

$$W ::= \top \mid H \mid H \multimapinv W \mid \forall x.W,$$

which is similar to the definition of K_O except that the additive true $\top$ is allowed as well. Following rather standard methods of using context to characterize a formula's meaning (as in, say, Kripke models or phase spaces), let $\|P\|$ denote the set of all multisets Γ of tests such that $\vdash \Gamma, P$ and define *testing equivalence* $P \simeq Q$ to hold when $\|P\| = \|Q\|$. Since $P \vdash Q$ implies $\|P\| \subseteq \|Q\|$, logically equivalent role formulas are testing equivalent. The converse is not the case.

In order to get a better handle on testing equivalence, it is valuable to find ways to simplify the structure of tests. As they stand now, we need to consider arbitrary implementations of roles that act as intruders. While intruders (tests) might do rather complicated internal actions, it is communications between the principals and the intruder via the network that should characterize the "external" meaning of a role.

Since data and encryption keys are encoded as constructors and eigenvariables, it would appear that we can apply the proof theoretic notion of *interpolant* [8] to help monitor communications. Classically, if $A \vdash B$ then an interpolant is a formula R such that $A \vdash R$ and $R \vdash B$ and all the non-logical symbols occurring in R occur in both A *and* B. In our linear logic setting, we can prove the following theorem. (For a related use of interpolants, see [22].)

Proposition 7.1 *Let Γ be a multiset of role formulas (principals) and let Δ be a multiset of tests (intruders) such that $\vdash \Gamma, \Delta$. This proof has an interpolant: that is, there is a formula R such the non-logical constants occurring in R occur in both Γ and in Δ, and is such that $\vdash \Gamma, R$ and $R \vdash \Delta$.*

Proof Outline. The proof is by induction on the structure of a cut-free proof of $\vdash \Gamma, \Delta$. The base case is the $\top$-R rule. In this case, the tester contains $\top$ and we also set the interpolant for this sequent to be $\top$. The other inference rules can be classified as either asynchronous or synchronous. The synchronous rules are $\forall L$ and $\multimap L$ and the other rules are asynchronous. All the asynchronous rules have exactly one premise and the interpolant associated to the premise is also associated to the conclusion. Now consider the two synchronous cases.

First notice that given the classification of formulas in the endsequent $\longrightarrow \Gamma, \Delta$ as being either principal or test, all formulas in all sequents of the (cut-free) proof can be similarly labeled as being subformulas of either a principal or a test.

Assume that the last inference rule of the proof is the following occurrence of $\multimap L$ and assume that R is an interpolant for the right premise.

$$\frac{H \longrightarrow \mathcal{A}_1 \qquad \Delta, \Gamma \longrightarrow K, \mathcal{A}_2}{\Delta, \Gamma, H \circ\!- K \longrightarrow \mathcal{A}_1, \mathcal{A}_2}$$

Here, Γ is a multiset of principal formulas and Δ is a multiset of tests. Let M_P be the $\parr$ of the principal atoms in $\mathcal{A}_1$ and let M_I be the $\parr$ of the test atoms in $\mathcal{A}_1$: if there are no such atoms of either classification, use $\perp$ as the formula. We now have two cases to consider: either the formula $H \circ\!- K$ comes from a principal or it comes from a test. In the first case, the interpolant associated to the bottom sequent is $M_I \parr R$. In the second case, the interpolant for the bottom sequent is $(M_P \circ\!- R)^\perp$.

Consider finally the $\forall L$ inference rule and assume that R is an interpolant for the premise.

$$\frac{Kt, \Gamma, \Delta \longrightarrow \mathcal{A}_P, \mathcal{A}_I}{\forall_\gamma x. Kx, \Gamma, \Delta \longrightarrow \mathcal{A}_P, \mathcal{A}_I}$$

Here, $\mathcal{A}_P$ is a multiset of principal atoms and $\mathcal{A}_I$ is a multiset of test atoms. Assume that the formula R is not an interpolant for the lower sequent since it contains occurrences of non-logical constants that do not occurr in both a principal formula and a test formula. In particular, let $c : \gamma$ be such a non-logical constant. Clearly, c has an occurrence in t. We must now distinquich two cases inorder to abstract away the occurrence of c in R as follows.

Case 1: The occurrence of the formula $\forall_\gamma x.Kx$ is a principal formula. In that case, we can conclude that t contains an occurrence of c, that no principal formula of the lower sequent contains c, and that some test forumula in the lower sequent also contains an occurrence of c. The new interpolant is then $\forall x.R[x/c]$ for some variable x of type γ. Proving the sequent $\forall_\gamma x.Kx, \Gamma \longrightarrow \mathcal{A}_P, \forall x.R[x/c]$ can be reduced to proving $Kt, \Gamma \longrightarrow \mathcal{A}_P, R$ by using a $\forall$-R (instantiating with c) and $\forall$-L rule (instantiating with t). Similarly, proving the sequent $\Delta, \forall x.R[x/c] \longrightarrow \mathcal{A}_I$ can be reduced to proving $\Delta, R \longrightarrow \mathcal{A}_I$ $\forall$-L rule (instantiating with c). Both of these two sequents are assumed provable using the inductive assumption.
Case 2: The occurrence of the formula $\forall_\gamma x.Kx$ is a test formula. In that case, we can conclude that t contains an occurrence of c, that no test formula of the lower sequent contains c, and that some principal forumula in the lower sequent also contains an occurrence of c. The new interpolant is then $\exists x.R[x/c]$ for some variable x of type γ. Proving the sequent $\forall_\gamma x.Kx, \Delta, \exists x.R[x/c] \longrightarrow \mathcal{A}_I$ can be reduced to proving $Kt, \Delta, R \longrightarrow \mathcal{A}_I$ by using a $\exists$-L (instantiating with c) and $\forall$-L rule (instantiating with t). Similarly, proving the sequent $\Gamma \longrightarrow \mathcal{A}_P, \exists x.R[x/c]$ can be reduced to proving $\Delta \longrightarrow \mathcal{A}_P, R$ using $\exists$-L rule (instantiating with c). Both of these sequents are assumed provable using the inductive assumption.

Repeat this abstraction for each possible constant c. The resulting abstraction is then an interpolant for the concluding sequent. In this way, one can build an interpolant inductively over the structure of the cut-free proof.

A more careful reading of the structure of interpolants shows that they have the following structure:

$$M ::= \perp \mid A \mid M \parr M$$

$$R^+ ::= \top \mid \forall x.R^+ \mid M \multimapinv R^- \qquad R^- ::= M \multimapinv R^+ \mid \forall x.R^-.$$

Formulas in the R^+ syntactic category are contain the interpolants and are called *traces*. Clearly, traces are in fact simple tests (W-formulas).

Let $\|P\|_t$ denote the set of all traces R such that $\vdash P, R$ and define *trace equivalence*, $P \simeq_t Q$, as $\|P\|_t = \|Q\|_t$. Since traces are tests, $P \simeq Q$ implies $P \simeq_t Q$. The interpolation theorem provides a simple proof of the converse.

Proposition 7.2 *The relations $\simeq$ and $\simeq_t$ coincide.*

Proof. Assume that $P \simeq_t Q$ and that Γ is a set of testers such that $\vdash P, \Gamma$. By Proposition 7.1, we know that there is a trace R such that $\vdash P, R$ and $R \vdash \Gamma$. Since $P \simeq_t Q$, we know that $\vdash Q, R$ and by cut, we know that $\vdash Q, \Gamma$. Thus $P \simeq Q$. $\square$

Thus, the structure of attacks from intruders (testers) can be characterized by using the much simpler notion of (single-threaded) trace. This result is familiar also from process calculus and when interpreted as a theorem of concurrency, it is certainly not new. What is new here is showing that it is a simple consequence of the proof theoretic notion of interpolant.

8 Future Work

This paper shows that the logical foundations of MSR can be enriched and, at the same time, simplified, at least in the sense that some details that were addressed with various non-logical constants (continuation predicates, explicit encryption procedures, etc) can be replaced with logical concepts. As a result, reasoning about the meaning and correctness of protocols should be enhanced. Although some minor examples in the paper illustrate this possibility, much additional work remains to be done to validate this conjecture. I outline here some key topics to develop further.

The Dolev-Yao intruder. The notion of tester introduce in Section 7 is a natural way to provide meaning to a formula (in proof theory) or processes (in concurrency). The exact connection, however, between testing and the Dolev-Yao notion of intruder must be formally established.

Security properties. Generally, the reason to formalize a security protocol is as a first step to establish various kinds of security related properties, such as secrecy and authentication. One starting point for establishing such properties uses reachability analysis, and techniques in [19,20], which work directly on logic specifications of the form given here, might be one place to start.

Experimentation. Of course, serious experimentation with protocols and their formal properties should be explored. Given a modern logic programming language such as λProlog that incorporates λ-terms and their unification, eigenvariables, and proof search, it is an easy matter to implement interpreters that can execute and explore simple reachability questions. Designing a deductive system that formally supports reasoning about specifications that contains such computational elements is significantly more difficult.

New quantification. An extension to proof theory described in [25] proposes a new quantifier, $\nabla x.G$, which might be more appropriate for capturing the notion of scoping of new names within a security protocol. It seems likely that ∇ could be used to generate nonces while $\forall$ would continue to be used for encryption keys. The distinction between these quantifiers would not appear in the execution of the operational semantics of protocols but in reasoning about their equivalence and behavior.

Related formalisms. There are obviously many other paradigms to which this work can be related. Given that there is a well known connection between π-calculus style encodings and eigenvariables [22], it would seem quite natural to explore more fully possible connections with the spi-calculus [27]. Also, the connection between *strand spaces* [3,5,11] and cut-free proofs using role formulas seems likely be strong: one of the relations involved in defining a strand space should be that of subformula within a common role and the other relation should relate two atoms (messages) appearing together in the same initial sequent.

Acknowledgments. I wish to thank Iliano Cervesato, Alwen Tiu, and the anonymous reviewers of previous versions of this paper for their suggestions.

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Operating Systems Review*, 23(5):1–13, December 1989.

[3] I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in linear logic. In H. Veith, N. Heintze, and E. Clark, editors, *2000 Workshop on Formal Methods and Computer Security - FMCS'00*, July 2000.

[4] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *12th IEEE Computer Security Foundations Workshop*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[5] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop*, pages 35–51, Cambridge, UK, 3–5 July 2000. IEEE Computer Society Press.

[6] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[7] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[8] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269– 285, 1957.

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

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

[11] F. J. T. Fabrega, C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *J. of Computer Security*, 7(2,3):191–230, 1999.

[12] V. Gehlot and C. Gunter. Normal process representatives. *Fifth IEEE Sym. on Logic in Computer Science*, pages 200–207, Philadelphia, June 1990.

[13] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[14] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3), June 2000.

[15] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[16] Max Kanovich. Horn programming in linear logic is NP-complete. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 200–210. IEEE Computer Society Press, June 1992.

[17] Max Kanovich. The complexity of Horn fragments of linear logic. *Annals of Pure and Applied Logic*, 69:195–241, 1994.

[18] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, eds, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, *LNCS* 1005, pp. 147–166. 1996.

[19] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[20] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *TCS*, 294(3):411–437, 2003.

[21] D. Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, June 1989. MIT Press.

[22] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *1992 Workshop on Extensions to Logic Programming*, LNCS 660, pages 242–265. Springer-Verlag.

[23] D. Miller. Forum: A multiple-conclusion specification language. *TCS*, 165(1):201–232, September 1996.

[24] D. Miller. Higher-order quantification and proof search. In H. Kirchner and C. Ringeissen, editors, *Proc. of AMAST 2002*, LNCS 2422, pages 60–74, 2002.

[25] D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. To appear in the Proceedings of LICS03. July 2003.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[27] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 99.

[28] Gopalan Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[29] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop*, June 2001.

[30] P. Syverson and I. Cervesato. The logic of authentication protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.