Electronic Notes in Theoretical Computer Science 40 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume40.html 44 pages

Typed Multiset Rewriting Specifications of Security Protocols

Iliano Cervesato [1,2]

Advanced Engineering and Sciences Division
ITT Industries, Inc.
Alexandria, VA 22303-1410 — USA

Abstract

The language *MSR* has successfully been used in the past to prove undecidability results about security protocols modeled according to the Dolev-Yao abstraction. In this paper, we revise this formalism into a flexible specification framework for complex crypto-protocols. More specifically, we equip it with an extensible typing infrastructure based on dependent types with subsorting, which elegantly captures and enforces basic relations among objects, such as between a public key and its inverse. We also introduce the notion of memory predicate, where principals can store information that survives role termination. These predicates allow specifying complex protocols structured into a coordinated collection of subprotocols. Moreover, they permit describing different attacker models using the same syntax as any other role. We demonstrate this possibility and the precision of our type system by presenting two formalizations of the Dolev-Yao intruder. We discuss two execution models for this revised version of *MSR*, one sequential and one parallel, and prove that the latter can be simulated by the former.

1 Introduction

The language *MSR* [8,15,9] is a formalism designed to give simple specifications of authentication protocols within the Dolev-Yao abstraction of computer security [20,14]. It models a protocol as a set of *parametric multiset rewriting rules* augmented with *existential quantification* as a device to guarantee the freshness of newly generated data. Indeed, *MSR* is an acronym for MultiSet Rewriting. As usual, *multisets* (also called *bags*) are algebraic structures that differ from sets by not collapsing multiple occurrences of an object (they can alternatively been seen as unordered sequences). Roles, i.e. the sequence of

[1] Partially supported by NSF grant INT98-15731 "Logical Methods for Formal Verification of Software" and NRL under contract N00173-00-C-2086.
[2] Email: iliano@itd.nrl.navy.mil

©2001 Published by Elsevier Science B. V.

messages sent or received by each principal participating in a protocol run, are expressed as multiset rewriting rules communicating through dedicated *role state predicates* that pass control and data to the next rule in the sequence. Rules operate on *states*, defined as multisets of first-order atomic formulas modeling the messages in transit, the role state predicates currently present, and *persistent information* which establishes immutable relations such as between a key and its inverse.

MSR has successfully been used to prove that certifying the correctness of an authentication protocol is in general undecidable [8,15]. We have moreover showed its substantial equivalence in expressive power to extensions of popular specification languages for security protocols, such as strand spaces [16], and further studied their similarities under the microscope of linear logic [6]. On the practical side, *MSR* fuels the *CAPSL* authentication protocol verification tool in the form of the underlying *CIL* intermediate language [13].

In spite of these achievements, *MSR* is a very low level language, poorly suited for the direct specification of security protocols. In particular, the slack constraints on the persistent information predicates present in a rule makes designing automated support procedures difficult and reasoning about specifications error-prone. These predicates, together with an elementary form of typing, describe associations among objects, such as between a key and the principal who owns it; this could conceivably be used to catch such errors as attempted accesses to unauthorized keys. Another shortcoming concerns its rigid rule format that accounts for lengthy specifications (and the consequent model-checking overhead [12]). Finally, the language defined in [8,15,9] lacks the expressiveness needed to describe but the simplest of authentication protocols. In particular, any protocol structured in a collection of subprotocols lies outside the scope of this language.

In this paper, we propose a thorough redesign of *MSR* intended to establish it as a usable specification framework for complex security protocols. The major innovations include the adoption of a flexible yet powerful typing methodology that subsumes persistent information predicates, and the introduction of memory predicates that significantly widen the range of applicability of this formalism. The detailed definition of this language also allows for a precise description of its execution semantics. We call this formalism *Typed MSR*, or just *MSR* when no ambiguity can arise. It fully subsumes our earlier presentations of this language [8,15].

The type annotations of our new language, drawn from the theory of dependent types with subsorting, enable precise object classifications for example by distinguishing keys on the basis of the principals they belong to, or in function of their intended use. Therefore, the public key of any two principals can be assigned a different type, in turn distinct from their digital signature keys. Protocol specifications, called protocol theories in *MSR*, are strongly typed, and we have devised algorithms for statically catching type violations, e.g. the use of a shared key to perform public-key encryption [4]. Our typing

infrastructure can point to more subtle errors, such as a principal trying to encrypt a message with a key that does not belong to him. A procedure for enforcing such access control policies is analyzed in [3].

Memory predicates allow a principal to remember information across role executions. Their presence opens the doors to the specification of protocols structured as a collection of coordinated subprotocols. This novel possibility is exemplified in [5], where we formalize the Neuman-Stubblebine repeated authentication protocol [21], which lies outside the reaches of our previous version of *MSR*. A particularly interesting application of memory predicates is given by their role in the specification of the intruder model against which a protocol is to be analyzed. Indeed, our enhanced version of *MSR* allows expressing an attacker as a distributed protocol that communicates through dedicated memory predicates. We exemplify this technique by giving two specifications of the generic Dolev-Yao intruder. We should stress that these specifications lie completely within the the language of our present version of *MSR*, while the attacker corresponded to a set of rules that did not fully follow the syntax of protocols in [8,15,9].

The detailed definition of *MSR* presented in this paper enables an unambiguous description of its execution semantics. We propose two models: one based on the sequential application of rules, while the second allows independent rules to fire in parallel. We prove that the latter mode can be emulated by sequential execution. Although this result is not particularly surprising in itself, the ability to give a completely formal proof further establishes the merit of the precise definition of our language.

This paper is organized as follows: in Section 2, we define the term language of *MSR*. Section 3 introduces states and their components and Section 4 presents rules, roles and protocol theories. The sequential and parallel execution models of *MSR* are discussed and correlated in Section 5. Section 6 formalize two variants of the Dolev-Yao intruder model in our language. Finally, Section 7 concludes this paper and outlines directions of future work.

2 Typed Messages

In Section 2.1, we describe the messages, or more generally terms, that form the core of *MSR*. Then in Section 2.2, we introduce the typing infrastructure that allow us to make sense of these terms. We will need them in the definition of protocol rules in Section 4. We conclude in Section 2.3 by pointing out the major differences with respect to the messages used in previous work on *MSR* [8,9].

2.1 Messages

Messages are obtained by applying a number of message forming constructs, discussed below, to a variety of *atomic messages*. The atomic messages we

will consider in this paper are principal identifiers, keys, nonces, and raw data (i.e. pieces of data that have no other function in a protocol than to be transmitted). We formalize our notion of atomic message by means of the following grammatical productions:

$$\begin{array}{rcll} \textit{Atomic messages: } a & ::= & \mathsf{A} & \textit{(Principal)} \\ & | & \mathsf{k} & \textit{(Key)} \\ & | & \mathsf{n} & \textit{(Nonce)} \\ & | & \mathsf{m} & \textit{(Raw datum)} \end{array}$$

Here and in the rest of the paper, A, k, n, and m will range over principal names, keys, nonces, and raw data respectively. We will sometimes also use B to denote a principal. Although we will limit the discussion in this paper to these kinds of atomic messages, it should be noted that others can be accommodated by extending the appropriate definitions.

The *message constructors* we will consider consist of concatenation, shared-key encryption, and public-key encryption. Altogether, they give rise to the following definition of a *message*, or more properly a *term*.

$$\begin{array}{rcll} \textit{Messages: } t & ::= & a & \textit{(Atomic messages)} \\ & | & x & \textit{(Variables)} \\ & | & t_1\, t_2 & \textit{(Concatenation)} \\ & | & \{t\}_{\mathsf{k}} & \textit{(Symmetric-key encryption)} \\ & | & \{\!\{t\}\!\}_{\mathsf{k}} & \textit{(Asymmetric-key encryption)} \end{array}$$

We will use the letter t, possibly sub- and/or super-scripted, to range over terms. Observe that we use a different syntax for shared-key and public-key encryption. We could have identified them, as it is done in many approaches. We choose instead to distinguish them to show the flexibility and precision of our technique.

Again, other constructors, for example digital signatures and hash functions, can easily be accommodated by extending the appropriate definitions. We refrain from doing so since their inclusion would lengthen the discussion without introducing substantially new concepts.

Type tuples (discussed in Section 3) and protocol rules (see Section 4) rely on objects that may contain variables to be instantiated during type-checking (this aspect is formalized in [4]) and execution, respectively. A *parametric message* allows variables where terms could appear in the messages of Section 2.1.

We will write A (or B), k, n and m, variously decorated, for atomic constants or variables that are principals, keys, nonces and raw data respectively. Whenever the object we want to refer to cannot be but a constant (mostly in the execution rules of Section 5), we will use the corresponding seriffed letters: A (or B), k, n and m. Instead, the letters x, y and z will stand for terms that must be variable. In some circumstances, we will need to refer to objects that

can be either variables or atomic message constants, but not composite terms. We call these terms *elementary* and denote them with the letter e, variously decorated.

Since most of the object we will be dealing with in the rest of the paper contain variables, we will use the wording "term" (or "message") to refer to "parametric terms" (or "messages"). Whenever we will be working with terms that do not contain variables, we will talk about "ground" terms (or messages), unless clear from the context.

2.2 Types

While types played a very modest role in the original definition of *MSR* [8,9], they stand at the core of the extension presented in this paper. Through typing, we can enforce basic well-formedness conditions (e.g. that only keys be used for encrypting a message), as described in detail in [4]. Types also provide a statically checkable way to ascertain complex desiderata such as, for example, that no principal may grab a key he/she is not entitled to access. This aspect is thoroughly analyzed in [3]. The central role of types in our present approach is witnessed by the fact that they subsume and integrally replace the "persistent information" of the original *MSR* [9].

The typing machinery that best fits our goals is based on the type-theoretic notion of *dependent product types with subsorting*. Rather than delving into the depth of the definitions and properties of this formalism, we will introduce only the facets that we will use, and only to the extent we will need them. In particular, we do not conduct an in-depth discussion of this type theory; we will even stay away from the most exotic aspects of its syntax. Readers who wish to further their understanding of this formalism are invited to consult [11,18,1,22].

Types are syntactic constructions that are used to classify other syntactic expression, such as terms. By doing so, they give them a *meaning*, saying for example that an object we interpret as a key is not a nonce. Whenever a key is used where a nonce is expected, something has gone wrong since the meaning of this term has been violated. The types we will use in this paper are summarized in the following grammar:

$$
\begin{array}{rlll}
\textit{Types: } \tau & ::= & \mathsf{principal} & \textit{(Principals)} \\
 & | & \mathsf{nonce} & \textit{(Nonces)} \\
 & | & \mathsf{shK}\ A\ B & \textit{(Shared keys)} \\
 & | & \mathsf{pubK}\ A & \textit{(Public keys)} \\
 & | & \mathsf{privK}\ k & \textit{(Private keys)} \\
 & | & \mathsf{atm} & \textit{(Atomic messages)} \\
 & | & \mathsf{msg} & \textit{(Messages)}
\end{array}
$$

In the sequel, τ, possibly variously decorated, will stand for a type. Needless to say, the types "principal" and "nonce" are used to classify principals and

nonces respectively. The next three productions allow distinguishing between shared keys, public keys and private keys. Dependent types offer a simple and flexible way to express the relations that hold between keys and their owner or other keys. Given principals "A" and "B", a shared key "k" between "A" and "B" will have type "shK A B". Here, the type of the key *depends* on the specific principals "A" and "B". Similarly, a constant "k" is given type "pubK A" to indicate that it is a public key belonging to "A". We use dependent types again to express the relation between a public key and its inverse. Continuing with the last example, the inverse of "k" will have type "privK k", from which it is easy to establish that it belongs to principal "A".

The type "atm" is intended to classify atomic terms, which clearly include raw data, but also principal names, nonces and all the above key types. This statement implies, for example, that a nonce "n" has both type "nonce" and "atm". We accommodate this form of hierarchical multiple classification by imposing a *subsorting* relation between types. We formalize this relation by means of the following judgment:

$$\tau :: \tau' \qquad\qquad \tau \textit{ is a subsort of } \tau'$$

In this paper, the subsorting relation requires that the types of principals, nonces and keys be subtypes of atm. Its extension is expressed by means of the following rules as far as atomic terms are concerned:

$$\frac{}{\mathsf{principal} :: \mathsf{atm}}\ \mathbf{ss_pr} \qquad\qquad \frac{}{\mathsf{nonce} :: \mathsf{atm}}\ \mathbf{ss_nnc}$$

$$\frac{}{\mathsf{shK}\ A\ B :: \mathsf{atm}}\ \mathbf{ss_shK} \qquad \frac{}{\mathsf{pubK}\ A :: \mathsf{atm}}\ \mathbf{ss_pbK} \qquad \frac{}{\mathsf{privK}\ k :: \mathsf{atm}}\ \mathbf{ss_pvK}$$

These rules are parametric since, for example, **ss_shK** establishes that "shKAB :: atm" holds for any choice of the principals "A" and "B".

The type msg is used to classify generic messages. Since nonces, keys, and principal identifiers are routinely part of messages, we make atm a subsort of msg:

$$\frac{}{\mathsf{atm} :: \mathsf{msg}}\ \mathbf{ss_atm}$$

Again, the types and the subsorting rules above should be thought of as a reasonable instance of our approach rather than the approach itself. Other schemas can be specified by defining appropriate types and how they relate to each other. For example, digital signatures could be accommodated by introducing dedicated dependent types akin to "pubK A" and "privK k". In another scenario, an application may find it convenient to see each of the key-related types above as a subtype of a universal key type, say "key", in turn a subsort of msg. As a final example, we may want to define distinct types for long-term keys and have them not be a subsort of msg, prohibiting in this way the transmission of long-term secrets as parts of messages; more complex key

stratifications coud be captured as well. These possibilities are exemplified in [4,5].

In the first part of this section, we have introduced terms and the types intended to classify them. We will now present the typing rules that will allow us to establish whether an expression built according to the syntax of terms can be considered a ground message (more in general whether a given expression free of variables has a certain type).

The rules below systematically reduce the typability of a composite term to the validity of its subterms. This is adequate for constructors, but atomic messages need to be treated specially unless we are willing to write new rules for them each time we model a protocol. Independence of rules from actual atomic messages is achieved through the notion of a *signature*. A signature, written Σ, is a finite sequence of declarations that map atomic messages to their type. More formally,

$$\begin{array}{rll} \textit{Signatures: } \Sigma ::= & \cdot & \textit{(Empty signature)} \\ & \mid \Sigma,\ a : \tau & \textit{(Atomic message declaration)} \\ & \mid \textit{(\ldots)} & \end{array}$$

The dots in the last line express the fact that this definition is incomplete: we will extend it in the next section.

We assume that each atomic message in a signature is declared exactly once. We will also often elide the leading "·" from a non-empty signature, and promote the extension operator "," to denote signature union. This operation is defined only if the resulting sequence is itself a signature (in particular it should not contain multiple declarations for the same constant).

Given the notions introduced so far, it is fairly easy to define a meaningful type system for messages and the other types we have described. In order to accomplish this goal, we will rely on the following message typing judgment:

$$\Sigma \vdash t : \tau \qquad\qquad \textit{Term } t \textit{ has type } \tau \textit{ in signature } \Sigma$$

All composite terms have type msg, given that their constituent submessages are correctly typed. This implies that the subterms of a concatenation $(t_1\, t_2)$ are themselves messages. On the other hand, the plaintext part t of an encrypted message $\{t\}_k$ should have type msg but k should be a shared key between two principals. Terms encrypted with public keys, of the form $\{\!\{t\}\!\}_k$, are handled similarly. This intuition is formally captured in the following typing rules for messages:

$$\frac{\Sigma \vdash t_1 : \mathsf{msg} \quad \Sigma \vdash t_2 : \mathsf{msg}}{\Sigma \vdash t_1\, t_2 : \mathsf{msg}}\ \textbf{mtp_cnc}$$

$$\frac{\Sigma \vdash t : \mathsf{msg} \quad \Sigma \vdash k : \mathsf{shK}\ A\ B}{\Sigma \vdash \{t\}_k : \mathsf{msg}}\ \textbf{mtp_ske}$$

$$\frac{\Sigma \vdash t : \mathsf{msg} \quad \Sigma \vdash k : \mathsf{pubK}\ A}{\Sigma \vdash \{\!\{t\}\!\}_k : \mathsf{msg}}\ \textbf{mtp_pke}$$

These rules are parametric, as witnessed by the numerous meta-variables they contain, but also hypothetical in the sense that the validity of the consequent relies on the validity of their premises.

The next rule reduces verifying that a term t has type τ to checking that it has type τ' for some subsort of τ. In this way, we can for example use a nonce or a key where an object of type msg is expected. The formal rule is as follows:

$$\frac{\Sigma \vdash t : \tau' \quad \tau' :: \tau}{\Sigma \vdash t : \tau}\ \textbf{mtp_ss}$$

The last term typing rule deals with elementary messages components. An atomic message a has a type τ if the signature at hand contains the declaration "$a : \tau$". The validity of the type τ in Σ is independently checked by verifying the validity of a signature [4].

$$\frac{}{(\Sigma, a : \tau, \Sigma') \vdash a : \tau}\ \textbf{mtp_a}$$

This concludes the presentation of the typing rules needed in this paper. It should be noted that these rules do not guarantee that the signature declarations obey the typing policy: for example, nothing forces the arguments A and B of the type in the rightmost premise of rule **mtp_ske** to be principals. Although well-typed signatures are an essential component of a correct specification, we do not need to require them in this paper. These omitted typing rules can be found in [4] together with procedures designed to validate every element of an *MSR* protocol specification.

For the convenience of the reader, we collect all the rules presented in this paper in Appendix A.

2.3 Changes

We will now compare the above term infrastructure with the notion of message used in earlier versions of *MSR* [8,9]. The main differences concern the types and the way they are used.

(i) Structurally, the original presentation of *MSR* referred to so-called *simple types*, i.e. types that do not depend on terms. These types would permit distinguishing objects into keys, nonces, messages, etc. (the taxonomy was open-ended), but would not allow any finer classification [7]. Subsorting introduced some flexibility. Most of the information that is now captured through types and especially their dependencies on terms was then expressed by means "persistent predicates" that cluttered protocol specifications and complicated reasoning about them. These entities have been dropped.

(ii) Typing played a rather inessential role in [8,9]: constants were given types, but rules themselves did not carry such information. Furthermore, no type system was explicitly presented to validate terms or rules. Typing is central to the language proposed in this paper: every object is typed. Indeed, our approach allows not only verifying basic well-formedness conditions (e.g. that no message is encrypted with a nonce) [4], but types provide a statically checkable way to enforce complex requirements such as, for example, that no principal uses a key he/she is not entitled to access [3].

(iii) In this paper, we make a syntactic distinction between shared-key and public-key encryption. Although not present in [8,9], we could have easily distinguished the two forms of encryption in the earlier version of *MSR*.

3 Message Predicates and States

As we will see shortly, a state is a collection of specialized atomic first-order formulas. States are a fundamental concept in *MSR*. Indeed, they are the central constituent of the snapshots of a protocol execution. They are the objects transformed by rewrite rules to simulate message exchange and information update. Finally, together with execution traces, they are the hypothetical scenarios on which protocol analysis is based.

In this section, we will formalize the concept of state in *MSR*, together with the constructions needed to implement it. In Section 3.1 we introduce message tuples and a notion of type for them. Then in Section 3.2, we discuss the message predicates that appear in a state, and in Section 3.3 we define states.

3.1 Message Tuples and Dependent Types

As we will see shortly, message predicates are atomic first-order formulas with zero or more terms as their arguments. In this section, we are concerned with formalizing the concept of tuple and presenting suitable types for them.

A *message tuple* is an ordered sequence of terms. It is trivially defined as follows:

$$\begin{array}{rcll} \textit{Message tuples } \vec{t} & ::= & \cdot & \textit{(Empty tuple)} \\ & | & t,\ \vec{t} & \textit{(Tuple extension)} \end{array}$$

We will often omit the leading $\cdot$ whenever a tuple is not empty.

It is tempting to define the type of a tuple as the sequence of the types of its components. Therefore, if A is a principal name and $\mathsf{k_A}$ is a public key for A, the tuple $(\mathsf{A}, \mathsf{k_A})$ would have type "$\mathsf{principal} \times \mathsf{pubK\ A}$" (the *Cartesian product* symbol "$\times$" is the standard constructor for type tuples). This construction allows us to associate a generic principal with A's public key: if B is another principal, then $(\mathsf{B}, \mathsf{k_A})$ will have this type as well. We will often need stricter associations, such as between a principal and its *own* public key. In order

to achieve this, we will rely on the notion of *dependent type tuple*. In this example, the tuple $(\mathsf{A}, \mathsf{k_A})$ will be attributed type "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$", where the variable A in "$\mathsf{principal}^{(A)}$" records the name of the principal at hand and forces the type of the key to be "$\mathsf{pubK}\ A$" for this particular A: therefore $(\mathsf{A}, \mathsf{k_A})$ is a valid pair of type "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$", but $(\mathsf{B}, \mathsf{k_A})$ is now ill-typed since $\mathsf{k_A}$ has type "$\mathsf{pubK\,A}$" rather than the expected "$\mathsf{pubK\,B}$". [3]

We do attribute a type to a term tuple by collecting the type of each constituent message, but we label these objects with variables to be used in later types that may depend on them. A *dependent type tuple* is therefore an ordered sequence of types parameterized as follows:

$$\begin{array}{llll} \textit{Type tuples}\ \vec{\tau} & ::= & \cdot & \textit{(Empty tuple)} \\ & \mid & \tau^{(x)} \times \vec{\tau} & \textit{(Type tuple extension)} \end{array}$$

In the second line of this definition, the notation $^{(x)}$ on the left of the Cartesian product symbol *binds* the variable x in the type tuple $\vec{\tau}$ to its right. Similarly for example to a quantifier, $\tau^{(x)}$ is a *binder*. Variables that are not bound in this way are said to be *free*. We will often be interested in *closed* type tuples, all of whose variables are bound. The *scope* of a binder is the expression over which its binding action spans. The scope of a type tuple component extends to the entire type tuple to its right.

Given a dependent tuple type $\tau^{(x)} \times \vec{\tau}$, we will drop the label $^{(x)}$ whenever the variable x does not occur (free) in $\vec{\tau}$. The resulting simplified notation, $\tau \times \vec{\tau}$, will help writing more legible specifications when possible. As for term tuples, we will omit the leading "$\cdot$" whenever convenient.

3.2 Message Predicates

The predicates that can enter a state or a rewrite rule are of three kinds:

- First, the predicate $\mathsf{N}(_)$ implements the contents of the *public network* in a distributed fashion: for each (ground) message t currently in transit, the state will contain a component of the form $\mathsf{N}(t)$.
- Second, active roles rely on a number of *role state predicates*, generally one for each rule in them, of the form $\mathsf{L}_l(_, \ldots, _)$, where l is a unique identifying label. The arguments of this predicate record the value of known parameters of the execution of the role up to the current point.
- Third, a principal A can store data in private memory predicates of the form $\mathsf{M}_A(_, \ldots, _)$ that survives role termination and can be used across the execution of different roles, as long as the principal stays the same.

[3] Our dependent type tuples are usually called weak dependent sums in the type theoretic community, and the standard notation for the dependent type tuple we have written as "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$" is "$\Sigma A : \mathsf{principal}.\, \mathsf{pubK}\ A$". We believe that our syntax is likely to be more clear to the target audience of this paper.

The reader familiar with our previous work on *MSR* will have noticed a number of differences with respect to the definitions given in [8,9]. Memory predicates are indeed new. They are intended to model situations that need to maintain data private across role executions: for example, this allows a principal to remember its Kerberos ticket, or the trusted-third-party of a fair exchange protocol to avoid fraudulent recoveries from aborted transactions. Another difference with respect to earlier work is the absence of a dedicated predicate retaining the intruder's knowledge. This can however be easily implemented using memory predicates, as we will see in Section 6.

Every protocol relies on a public network. Therefore, we will hardwire the network predicate $\mathsf{N}(_)$ in our language. Local state and memory predicates are different: they are defined on a per-protocol basis. This is similar to principals and keys. We therefore maintain generality by declaring them as part of the signature. We can now complete the definition of a signature as follows:

$$
\begin{array}{lll}
\textit{Signatures } \Sigma ::= & \cdot & \textit{(Empty signature)} \\
 & \mid \Sigma,\ a : \tau & \textit{(Atomic message declaration)} \\
 & \mid \Sigma,\ \mathsf{L}_l : \vec{\tau} & \textit{(Local state predicate declaration)} \\
 & \mid \Sigma,\ \mathsf{M}__ : \vec{\tau} & \textit{(Memory predicate declaration)}
\end{array}
$$

3.3 States

A *multiset* is a collection of objects whose multiplicity counts, but whose order is irrelevant. We can see them either as unordered sequences, or as sets that allow several instances of the same element to coexist (and prohibit collapsing them). A *state* is a finite multiset of ground state predicates. The syntax of states is partially formalized by means of the following grammar:

$$
\begin{array}{lll}
\textit{States: } S ::= & \cdot & \textit{(Empty state)} \\
 & \mid S,\ \mathsf{N}(t) & \textit{(Extension with a network predicate)} \\
 & \mid S,\ \mathsf{L}_l(\vec{t}) & \textit{(Extension with a role state predicate)} \\
 & \mid S,\ \mathsf{M}_\mathsf{A}(\vec{t}) & \textit{(Extension with a memory predicate)}
\end{array}
$$

As for signatures, we will omit the leading "·" and interpret the extension construct "," as a union operator. It will be convenient to break from the rigid format imposed by the above grammatical productions, and abstract from the order of the component predicates of a state, treating them as a multiset.

Protocol rules transform states. They do so by identifying a number of component predicates, removing them from the state, and adding other, usually related, state elements. The antecedent and consequent of a rewrite rule embed therefore substates. However, in order to be applicable to a wide array of states, rules usually contain variables that are instantiated at application time. This calls for a parametric notion of states and message predicates.

For the most part, this reduces to admitting variables in the embedded terms. However, role state predicates need to be created on the spot in order to avoid interferences. We achieve this by introducing variables, denoted L, that are instantiated to actual role state predicates during application. This makes our language weakly second-order, although we could easily reduce it to the first order by interpreting a role state predicate L_l as the symbol L indexed by a label l that is kept as a variable in rules. We however opt for the more direct solution since it does not have any drawback, and allows for slightly simpler definitions.

3.4 Changes

In the final part of this section, we will analyze the main differences between the notion of state and related concepts defined above and the analogous entities from the original definition of *MSR* [8,7].

Typing constitutes a minor distinction at this level since both versions of *MSR* rely on typed predicate names, although little use of this aspect was made in [8,7]. Our present usage of dependent type tuples is mostly a consequence of our adoption of dependent types to classify terms. The major differences regard instead the introduction of memory predicates and the elimination of persistent information and distinguished intruder knowledge predicates in states.

(i) In earlier versions of *MSR*, a principal had no way to remember data across role executions: role state predicates are local to a role instance, and network messages are not intended for storing private information. Memory predicates survive role termination. As already mentioned, this is useful when modeling scenarios that require remembering information across role executions: for example, this allows a principal to remember its Kerberos ticket, or the trusted-third-party of a fair exchange protocol to avoid fraudulent recoveries from aborted transactions. Another novel important use of memory predicate is in modeling protocols consisting of a number of subprotocols: these predicates allow subprotocols to call each other and share data.

(ii) There is no trace in the above definitions of the persistent information that formed the immutable portion of the state of *MSR* in [8,7]. The functionality of those predicates is now performed by the strong typing policy of our updated framework, and in particular by our reliance on dependent types.

(iii) Finally, we should point out the absence of a dedicated predicate intended to hold the intruder's knowledge. This aspect of our earlier work can now be realized transparently through memory predicates, as we will witness in Section 6.

4 Multiset Rewriting Theories

In the past, crypto-protocols have often been presented as the temporal sequence of messages being transmitted during a "normal" run. Recent proposals champion a view that places the involved parties in the foreground. A protocol is then a collection of independent *roles* that communicate by exchanging messages, without any reference to runs of any kind. A role has an owner, the principal that executes it, and specifies the sequence of messages that he/she will send, possibly in response to receiving messages of some expected form. A role can therefore be seen as a reactive system.

MSR adopts and formalizes this perspective. It represents protocols as a set of syntactic entities that we also call roles. A role is itself given as a parameterized collection of multiset rewrite rules that encode the expected message receptions and the corresponding transmission. Rule firing emulates receiving (and accepting) a message and/or sending a message, the smallest execution steps.

This section defines rules, roles and protocol theories. More specifically, in Section 4.1, we introduce rules and their constituents. Roles are defined in Section 4.2. Then, in Section 4.3, we turn our attention to protocol theories, which are our representation of protocols. Finally, in Section 4.4 we anticipate the dynamic notion of an active role. Roles and protocol theories will be further discussed in Sections 5 which focuses on execution. Section 4.5 examines how these definitions differ from our original presentation of *MSR*.

4.1 Rules

The core of a rule has the form "$lhs \to rhs$". Rules are the basic mechanism that enables the transformation of a state into another, and therefore the simulation of protocol execution: whenever the antecedent "lhs" matches part of the current state, this portion may be substituted with the consequent "rhs" (after some processing).

It is convenient to make protocol rules parametric so that the same rule can be used in a number of slightly different scenarios (e.g. without fixing interlocutors or nonces). A typical rule will therefore mention variables $x_1, \ldots, x_n$ that will be instantiated to actual terms during execution. Typed universal quantifiers can conveniently express this fact so that rules assume the form "$\forall x_1 : \tau_1. \ldots \forall x_n : \tau_n. (lhs \to rhs)$". This idea is more precisely captured by the following grammar:

$$\begin{array}{lll} \textit{Rule: } r ::= & lhs \to rhs & \textit{(Rule core)} \\ & \mid \forall x : \tau.\, r & \textit{(Parameter closure)} \end{array}$$

The universal quantifiers used in rules bind the variables they are applied to. Free variables can occur in the construction of a rule, but roles themselves should have all their variables bound (this is enforced by the typing rules):

they are closed. The scope of all the binders in the above productions spans over a whole rule.

If we think of variables as place holders, binders dictate which place holders must be instantiated with the same term (or predicate name). Variable names implement this association. Besides a possibly mnemonic function, the names of variables themselves have little importance and can be replaced with arbitrary strings as long as no conflict with a free variable is introduced (an effect called *variable capture*: renaming would cause a free variable to become bound). It will be convenient (e.g. when applying a substitution) to allow the implicit capture-free renaming of bound variables in a predicate sequence (this is known as α-*conversion* in type-theoretic circles).

The *left-hand side*, or *antecedent*, of a rule is a finite collection of parametric message predicates, and is therefore given by the following grammar for predicate sequences:

$$
\begin{array}{llll}
\textit{Predicate sequences: lhs} & ::= & \cdot & \textit{(Empty predicate sequence)} \\
& | & \textit{lhs},\ \mathsf{N}(t) & \textit{(Extension with a network predicate)} \\
& | & \textit{lhs},\ L(\vec{e}) & \textit{(Extension with a role state predicate)} \\
& | & \textit{lhs},\ \mathsf{M}_A(\vec{t}) & \textit{(Extension with a memory predicate)}
\end{array}
$$

Observe that rule antecedents and in general predicate sequences differ from states (see Section 3.3) mainly by the limited instantiation of role state predicates: in a rule, these objects consist of a role state predicate variable applied to as many elementary terms as dictated by its type. Recall that elementary terms are either variables or atomic message constants. Network and memory predicates will in general contain parametric terms, although not necessarily raw variables as arguments.

The *right-hand side*, or *consequent*, of a rule consists of a predicate sequence possibly prefixed by a finite string of fresh data declarations such as nonces or short-term keys. We rely on the existential quantification symbol to express data generation. We have the following grammar:

$$
\begin{array}{llll}
\textit{Right-Hand sides: rhs} & ::= & \textit{lhs} & \textit{(Sequence of message predicates)} \\
& | & \exists x : \tau.\, \textit{rhs} & \textit{(Fresh data generation)}
\end{array}
$$

The notion of fresh and bound variable discussed earlier applies also here. Notice that the scope of these quantifiers is limited to the right-hand side of the current rule. Later rules can refer to the values created by these variables by introducing universal quantifiers of the proper type: synchronization is ensured by their occurrence in the role state predicates. We have shown in [6] that, when encoding of *MSR* in logic, our marker for fresh data is indeed rendered by an existential quantification.

4.2 Roles

Role state predicates record information accessed by a rule. They are also the mechanism by which a rule can enable the execution of another rule in the same role. Relying on a fixed protocol-wide set of role state predicates is dangerous since it could cause unexpected interferences between different instances of a role executing at the same time. Instead, we make role state predicates local to a role by requiring that fresh names be used each time a new instance of a role is executed. As in the case of rule consequents, we achieve this effect by using existential quantifiers: we prefix a collection of rules ρ that should share the same role state predicate L by a declaration of the form "$\exists L : \vec{\tau}$", where the typed existential quantifier expresses the fact that L should be instantiated with a fresh role state predicate name of type $\vec{\tau}$. Again, a translation of *MSR* in logic interprets these declarations as existential quantifications [6].

With this insight, the following grammar defines the notion of rule collection:

Rule collections: $\rho ::= \cdot$		*(Empty role)*
	$\mid \exists L : \vec{\tau}.\, \rho$	*(Role state predicate parameter declaration)*
	$\mid r,\ \rho$	*(Extension with a rule)*

It should be observed that this definition allows for role state predicate parameters declarations and rules to be interleaved in a rule collection. We will however generally divide a collection in a *preamble* where all roles state parameters are declared, and a *body* that lists the rules that constitute a role.

A *role* is given as the association between a *role owner* A and a collection of rules ρ. Some roles, such as those implementing a server or an intruder, are intrinsically bound to a few specific principals, often just one. We call them *anchored roles* and denote them as

$$\rho^{\mathsf{A}}$$

Here, the role owner A is an actual principal name, a constant. Other roles can be executed by any principal. In these cases A must be kept as a parameter bound to the role. We use the following syntax to represent these *generic roles*:

$$\rho^{\forall A}$$

where the implicitly typed universal quantification symbol implies that A should be instantiated to a principal before any rule in ρ is executed, and sets the scope of the binding to ρ. Observe that in this case A is a variable. With a slight abuse of notation, we will sometimes refer to roles of either kind with the letter ρ, variously subscripted.

4.3 Protocol Theories

A *protocol theory*, written $\mathcal{P}$, is a finite collection of roles:

$$\begin{array}{lll} \textit{Protocol theories: } \mathcal{P} ::= & \cdot & \textit{(Empty protocol theory)} \\ & \mid \mathcal{P},\ \rho^{\forall A} & \textit{(Extension with a generic role)} \\ & \mid \mathcal{P},\ \rho^{A} & \textit{(Extension with an anchored role)} \end{array}$$

It should be observed that we do not make any special provision for the intruder. The adversary is expressed as one or more roles in the same way as the more legitimate message exchange in a protocol. We will illustrate how this is achieved for the standard Dolev-Yao intruder and for a variant of it in Section 6.

4.4 Active Roles

As we will see in Section 5, several instances of a given role, possibly stopped at different rules, can be present at any moment during execution. We record the role instances currently in use, the point at which each is stopped, and the principal who is executing them in an *active role set*. These objects are finite collections of *active roles*, i.e. partially instantiated rule collections, each labelled with a principal name. The following grammar captures their macroscopic structure:

$$\begin{array}{lll} \textit{Active role sets: } R ::= & \cdot & \textit{(No active role)} \\ & \mid R,\ \rho^{\mathsf{A}} & \textit{(Extension with an instantiated role)} \end{array}$$

The notation ρ^{A} is reminiscent of anchored roles. Active roles are actually more liberal in that some of the role state predicate symbols as well as their arguments may be instantiated. Intuitively, ρ^{A} results from instantiating the contents of some role, with A is its elected owner.

4.5 Changes

It should be noted that the above syntax of rules is much more liberal than the original definition of *MSR* [8,9]. We dedicate the remainder of this section to commenting on these differences. We begin by noting structural changes (items i–iii), then move to persistent information (item iv), memory predicates (item v), and network predicates (item vi). We then continue with a number of remarks concerning the role state predicates (items vii–x), and conclude with active roles (item xi).

(i) A protocol theory was defined in [8,9] as a collection of rules for which the graph generated by their role state predicates was acyclic (see item x below for further details). Each connected component corresponded to a role. Besides this constraint, the rules within a role were independent. This aspect is mostly maintained in our current formulation, but

our definition of role allow threading rules by using role state predicate declarations as a sequencing device. This option is seldom used since this effect can be achieved more simply by appropriately using role state predicates.

(ii) In [8,9], all variables were implicitly universally quantified at the head of a rule, unless marked in the consequent as fresh data. Here, these variables are explicitly introduced by typed universal quantifiers. Explicit quantification simplifies rule analysis as far as type-checking [4] and access control [3] are concerned. More importantly, it declares the type of variables, which is necessary for the correct execution of a protocol (more on this aspect in 5).

(iii) The only quantifiers present in [8,9] appeared in rule consequents and had the purpose of introducing variables to be instantiated with fresh data. The adoption of dependent types makes this mechanism more powerful. Our current schema allows for example a server to generate a key to be shared among two specific principals.

(iv) As already observed, our current formulation does not make use of the persistent predicates of [8,9]. As we said, the information once conveyed by these objects is not entirely captured within the type system of *MSR*.

(v) Memory predicates are a novel feature. As already mentioned, they are useful whenever data should be remembered across role executions, which happens, for example, when a subprotocol can be executed repeatedly after some initialization phase.

(vi) Any number of network predicates can appear in both the antecedent and the consequent of a rule. The original definition allowed at most once such predicate per rule, either in the left-hand side (in "receive" rules), or in the right-hand side (for "send" rules). Denker et al. [12] showed that, whenever some simple conditions are met, a sequence of contiguous message reception rules followed by a sequence of contiguous transmission rules can be soundly collapsed into a single rule of the proposed form. This is clearly a conservative extension of our previous proposal. Besides reducing the length of the specification of a protocol, the practical value of merging rules lies in the fact that it yields a huge overhead reduction when model-checking a protocol.

(vii) In [8,9], each rule contained exactly one role state predicate on each side, with the exception of the single "role generation rule" that activated the role. The productions above instead allow zero, one, or several such predicates on either side. This definition simplifies the grammatical specification of roles. However, we will make a very limited use of this added expressive power. Having more than one role state predicate in the left-hand or right-hand side of a rule does not appear particularly useful since the arguments of these predicates collect data known to a principal in the current thread of execution of a role. The antecedent of

the first rule in a role cannot sensibly mention any role state predicate since its left-hand side could not match any state. The antecedent of the other rules typically will have such a predicate for synchronization, although this is not enforced. For the same reason, the consequent of a rule will generally contain a role state predicate. The natural exception to this practice regards the final rules of a role, which do not need to make provisions for further synchronization.

(viii) The role state predicate symbols allowed in [8,9] were constants rather than variables. We now prefer to make them unique to each instantiation of a rule in order to avoid the possibility of interferences (although we have no examples in which such a phenomenon is harmful).

(ix) With the exception of [7], our previous work did not refer to any argument of a role state predicate as a distinguished role owner. These early versions of *MSR* did not need to actively rely on role owners to fulfill their objectives, i.e. studying the decidability of discovering attacks in a protocol. Therefore, no particular emphasis was given to this notion. This information becomes crucial when trying to enforce access control, as described in [3].

(x) For any given role, it is interesting to study the graph whose nodes are its role state predicates and whose edges link the predicate occurring in the left-hand side to the predicate in the right-hand side of each rule. In [8,9,6] we required that this graph be an upper semi-lattice, and in particular that it had a single entry point and be acyclic. This was a crucial assumption for studying the complexity of protocols [8,15]. Here, we drop this condition from the syntax of a role, but our firing rules will de facto implement a similar constraint at execution time.

(xi) Our previous work [8,9] did not have a notion of active role: since role state predicate symbols were constants, rules could not be threaded within a role and each rule was necessarily an independent object that could be applied whenever its antecedent matched the current state. The more complex pattern proposed in this chapter require memorizing which roles are in current use, and how they are instantiated. This is the purpose of active role sets. It must be observed that these entities are closely related to the notions of residual strands studied in [9] as a part of an execution model for strand specifications of security protocols [16].

5 Protocol Execution

In this section, we will define the execution model of *MSR*. More precisely, we first outline the definition of the pervasive notion of substitution in Section 5.1. Then, we present the basic one-step rule application and its sequential iteration in Section 5.2. In Section 5.3, we extend this notion to allow concurrent rule firing. We prove the admissibility of parallel firing in Section 5.4. We

conclude in Section 5.5 with a discussion of the main changes with respect to previous versions of *MSR*.

5.1 Substitutions

In this section, we introduce the notion of substitution used in the execution rules below. We focus mainly on the syntax of this operation. Its fairly standard definition is displayed in Appendix A.3.

Given a variable x and an object O that may mention x as a parameter, we denote the *substitution* of a term t for x in O as $[t/x]O$. As far as the execution rules are concerned, the instantiating term t will always be ground, and therefore the implementation of substitution does not need to take particular provisions aimed at avoiding variable capture (i.e. the risk that a free variable in t may accidentally become bound in O). The parametric objects O we will be interested in are other terms t', term tuples $\vec{t}$, types τ, type tuples $\vec{\tau}$, predicate sequences lhs, right-hand sides rhs, rules r, and rule collections ρ. All together, we will encounter the following forms of substitution:

$$[t/x]t' \quad [t/x]\vec{t} \quad [t/x]\tau \quad [t/x]\vec{\tau} \quad [t/x]lhs \quad [t/x]rhs \quad [t/x]r \quad [t/x]\rho$$

Observe that, for simplicity, we are overloading the bracket notation to denote the application of the substitution operation to objects belonging to different syntactic categories. We only need to define the substitution of t for x in objects that may mention x as a free variable. This is why we do not include protocol theories, states and active roles in the above list.

Besides term variables, rule collections contain a second kind of parameter, namely role state predicate symbols, that we have denoted with the letter L, variously decorated. During execution, we will need to instantiate them to constants of the form L_l, where l is a label. We extend our syntax and write $[\mathsf{L}_l/L]O$ for the substitution of variable L with L_l in object O. This operation applies to left-hand sides lhs, consequents rhs, rules r and rule collections ρ:

$$[\mathsf{L}_l/L]lhs \qquad [\mathsf{L}_l/L]rhs \qquad [\mathsf{L}_l/L]r \qquad [\mathsf{L}_l/L]\rho$$

It should be observed that although L stands for a predicate symbol, it never needs to be instantiated to anything more complex than a constant. The higher-orderness of our notation is only apparent.

Again, the definition of these operations can be found in Appendix A.3.

5.2 Sequential Firing

Execution is concerned with the use of a protocol theory to move from a situation described by a state S to another situation modeled by a state S'. In this section, we will introduce judgments and rules describing the atomic execution steps that lead from S to S'. We will then show how they can be chained to perform multi-rule sequential applications.

Referring to the situation that the execution of a protocol has reached by means of a state is an oversimplification. Two more ingredients are required: first we need to know which roles can be used in order to continue the execution, at which point they were stopped, and how they were instantiated. This calls for an active role set. Second, it is very convenient to carry around a list of the constants in use in a signature: this allows us in particular to verify that instantiations are well-formed and well-typed. Situations are then formally defined as a triple consisting of a state S, an active role set R and a signature Σ. Such triples are called *snapshots*, and denoted as in the following grammatical production:

$$\textit{Snapshot: } C ::= [S]_{\Sigma}^{R}$$

We shall observe that no element in a snapshot contains free variables: Σ is clearly ground, and so is the state S; the active role set R will generally contain bound variables, but execution will always instantiate them to ground terms as it exposes them.

Given a protocol $\mathcal{P}$, we describe the fact that execution transforms a snapshot C into another snapshot C' in one step by means of the following judgment, where we have expanded the definition of C and C' to familiarize the reader:

$$\mathcal{P} \triangleright [S]_{\Sigma}^{R} \longrightarrow [S']_{\Sigma'}^{R'} \qquad\qquad \textit{One-step sequential firing}$$

This judgment is implemented by the next six rules that fall into three classes. We should be able to: first, make a role from $\mathcal{P}$ available for execution; second, perform instantiations and apply a rule; and third, skip rules (more on this later).

We first examine how to extend the current active role set R with a role taken from the protocol specification $\mathcal{P}$. As defined in Section 4.3, $\mathcal{P}$ can contain both anchored roles ρ^{A} and generic roles $\rho^{\forall A}$. This yields the following two rules, respectively:

$$\frac{}{(\mathcal{P}, \rho^{\mathsf{A}}) \triangleright [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,\rho^{\mathsf{A}}}}\,\textbf{exs_arole}$$

$$\frac{\Sigma \vdash \mathsf{A} : \mathsf{principal}}{(\mathcal{P}, \rho^{\forall A}) \triangleright [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,([\mathsf{A}/A]\rho)^{\mathsf{A}}}}\,\textbf{exs_grole}$$

Anchored roles can simply be copied to the current active role sets since their syntax meets the requirements for active roles. In order to make a generic role available for execution, we must assign it an owner. The premise of rule **exs_grole** selects a principal name A from the current signature Σ and instantiates ρ with it. Observe that this premise relies the typing judgment to make sure that A is defined and that it actually stands for a principal name. We could have alternatively looked it up in Σ directly.

Once a role has been activated by either of the above rules, chances are that it contains role state predicate parameter declarations that require to be instantiated with actual constants before any of the embedded rules can be applied. This situation, characterized by the fact that the active role under examination has the form $(\exists L : \vec{\tau}.\, \rho)^{\mathsf{A}}$, is implemented by the following rule, which generates a fresh constant L_l, adds a declaration for it in the current signature Σ, and replaces every occurrence of L in ρ with it. Notice that L_l shall be a new symbol that appears nowhere in the current snapshot (in particular it should not occur in Σ).

$$\frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(\exists L:\vec{\tau}.\,\rho)^{\mathsf{A}}} \longrightarrow [S]_{(\Sigma,\mathsf{L}_l:\vec{\tau})}^{R,([\mathsf{L}_l/L]\rho)^{\mathsf{A}}}}\ \textbf{exs_rsp}$$

Processing a role state predicate parameter declaration prefix may have the effect of exposing a protocol rule r. At this point, r can participate in an atomic execution step in two ways: we can either skip it (discuss below), or we can apply it to the current snapshot to obtain a new configuration. The latter option is implemented by the inference rule below, which makes use of the rule application judgment "$r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}$" to construct the state S' and the signature Σ' resulting from the application. We will describe this judgment shortly. The changes to the active role set are limited to discarding the used rule r.

$$\frac{r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(r,\rho)^{\mathsf{A}}} \longrightarrow [S']_{\Sigma'}^{R,(\rho)^{\mathsf{A}}}}\ \textbf{exs_rule}$$

Only the simplest of security protocols specify a purely linear sequence of actions. More complex systems allow various forms of branching or even more complex layouts. In a protocol theory, the control structure is mostly realized by the role state predicates appearing in a role. Branching can be modeled by having two rules share the same role state predicate parameter in their left-hand side. Roles, on the other hand, are defined as a linear collection of rules. Therefore, in order to access alternative role continuations, we may need to *skip* a rule, i.e. discard it and continue with the rest of the specification. The following two execution rules implement this scenario:

$$\frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(r,\rho)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R,(\rho)^{\mathsf{A}}}}\ \textbf{exs_skp} \qquad\qquad \frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(\cdot)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R}}\ \textbf{exs_dot}$$

The inference on the left skips a protocol rule. Rule **exs_dot** does some house-keeping by throwing away active roles that have been completely executed.

When successful, the application of a rule r to a state S in the signature Σ produces an updated state S' defined in the extended signature Σ'. This operation is defined by the following judgment:

$$r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'} \qquad\qquad \textit{Rule application}$$

It is implemented by two rules that discriminate on the structure of a protocol rule.

In order to apply a rule to the current state, we first need to appropriately instantiate the universal variables that may appear in it. Our next execution rule will therefore examine rules of the form $(\forall x : \tau.\, r)$. It instantiates x to some well-formed term t of type τ in the current signature:

$$\frac{\Sigma \vdash t : \tau \quad [t/x]r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}}{(\forall x : \tau.\, r) \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}}\ \textbf{exs_all}$$

Again, we make use of the typing judgment to ascertain that t has type τ in Σ. The attentive reader may be concerned by the fact that the construction of the instantiating term t is not guided by the contents of the state S. This is a very legitimate observation: the rule above provides an idealized model of the execution rather than the basis for the implementation of an actual simulator. We may want to think of the premise of this rule as a non-deterministic oracle that will construct the "right" term to successfully continue the execution. An operational model suited for implementation is the subject of current research.

We now consider execution steps resulting from the application of a fully instantiated rule of the form "$lhs \rightarrow rhs$". The antecedent lhs must be ground and therefore it has the structure of a legal state. This rule identifies lhs in the current state and replaces it with a substate lhs' derived from the consequent rhs. This latter operation is performed in the premise of this rule by the right-hand side instantiation judgment "$(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}$" that will be discussed shortly.

$$\frac{(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}}{(lhs \rightarrow rhs) \triangleright [S, lhs]_{\Sigma} \gg [S, lhs']_{\Sigma'}}\ \textbf{exs_core}$$

The right-hand side instantiation judgment used in the premise of rule **exs_core** generates a ground predicate sequence lhs from the consequent rhs of a fully instantiated rule r from an active role $(r, \rho)^{\mathsf{A}}$. The resulting changes to the current signature Σ are reflected in the updated signature Σ'. This judgment is defined as follows:

$(rhs)_{\Sigma} \gg (lhs)_{\Sigma'}$ *Right-hand side instantiation*

It is implemented by the following two rules, which instantiate the existentially quantified variables possibly wrapped around the core of rhs. If this parameter is already a predicate sequence, we simply return it, without making any change to the signature Σ:

$$\frac{}{(lhs)_{\Sigma} \gg (lhs)_{\Sigma}}\ \textbf{exs_seq}$$

The instantiation of an existentially quantified term variable x is handled in rule **exs_nnc** below. We generate a fresh term constant a of the appropriate

type, records this fact in the signature, and replaces every occurrence of x with a before examining the body of rhs.

$$\frac{([\mathsf{a}/x]rhs)_{(\Sigma,\mathsf{a}:\tau)} \gg (lhs)_{\Sigma'}}{(\exists x:\tau.\, rhs)_\Sigma \gg (lhs)_{\Sigma'}}\,\textbf{exs_nnc}$$

Observe that, in this rule, the generated constants should not appear anywhere in the current signature Σ (and therefore in the left-hand side of the judgment): this object is new.

We conclude this section by providing a judgment and the associated inference rules that allow us to chain the atomic steps presented above into multi-step sequential firings. This allows simulating executions consisting of the linear application of any number of basic steps between two snapshots of interest. We have the following judgment:

$$\mathcal{P} \triangleright C \;\longrightarrow^*\; C' \qquad\qquad \textit{Multi-step sequential firing}$$

The following rules define this judgment as the reflexive and transitive closure of the atomic step relation discussed above.

$$\frac{}{\mathcal{P} \triangleright C \;\longrightarrow^*\; C}\,\textbf{exs_it0} \qquad \frac{\mathcal{P} \triangleright C \;\longrightarrow\; C' \quad \mathcal{P} \triangleright C' \;\longrightarrow^*\; C''}{\mathcal{P} \triangleright C \;\longrightarrow^*\; C''}\,\textbf{exs_itn}$$

We conclude this section with a simple lemma that asserts that, as execution proceeds, the initial signature can only be extended. As a space saver, we write $\mathcal{P} \triangleright C \;\longrightarrow^{(*)}\; C'$ to indicate that we are considering the one-step ($\mathcal{P} \triangleright C \;\longrightarrow\; C'$) and the multi-step ($\mathcal{P} \triangleright C \;\longrightarrow^*\; C'$) versions of the sequential firing judgment at once.

Lemma 5.1 (*Signature Extension*)

Let Σ and Σ' be signatures, rhs the right-hand side of a rule, lhs a predicate sequence, r a rule, S and S' states, $\mathcal{P}$ a protocol theory, and R and R' active role sets.

(i) *If* $(rhs)_\Sigma \gg (lhs)_{\Sigma'}$, *then* $\Sigma' = (\Sigma, \Sigma'')$ *for some signature* Σ''.

(ii) *If* $r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}$, *then* $\Sigma' = (\Sigma, \Sigma'')$ *for some signature* Σ''.

(iii) *If* $\mathcal{P} \triangleright [S]^R_\Sigma \;\longrightarrow^{(*)}\; [S']^{R'}_{\Sigma'}$, *then* $\Sigma' = (\Sigma, \Sigma'')$ *for some signature* Σ''.

Proof: The proof proceeds by induction on the structure of a derivation of the judgments in each of the three parts of this lemma. These results are not mutually recursive and shall be proved in the given order. We present only a sketch of this very simple proof, reserving a detailed illustration of the technique used for more substantial examples in the sequel.

The first result (1) is immediate for rule **exs_seq** and follows by an appeal to the induction hypothesis for rule **exs_nnc**. The second point in this

lemma (2) relies on (1) for rule **exs_core** and the indurction hypothesis for rule **exs_all**. The last result (3), is immediate for rules **exs_arole**, **exs_grole**, **exs_rsp**, **exs_skp**, **exs_dot**, and **exs_it0**. It relies on (2) for rule **exs_rule** and follows by induction for rule **exs_itn**. □

5.3 Parallel Firing

Multi-step sequential firing simulates the execution of a protocol one rule at a time. However, even the simplest of protocols are inherently concurrent systems and allow independent roles to be executing at the same time. Although interleaving permits reducing such a behavior to the sequential case, it is interesting to provide a direct specification of this model. In this section, we will first discuss executions that can be obtained as the parallel composition of one-step sequential firings, and then generalize it to multi-step parallel firing. In Section 5.4, we will establish a formal relationship between the sequential and parallel models of execution.

We will rely on the following judgment to express the fact that snapshot C' is obtained from C by executing zero or more atomic steps in parallel:

$$\mathcal{P} \triangleright C \Longrightarrow C' \qquad\qquad \textit{One-step parallel firing}$$

The two rules below implement this judgment. The toptmost inference captures the degenerate situation where no basic step is applied: the current snapshot is returned as output. Rule **exp_par** expresses parallel execution: intuitively, we want to partition the current state description into partial snapshots, independently apply one basic step to each, and then merge the results together to obtain the output snapshot.

$$\frac{}{\mathcal{P} \triangleright C \Longrightarrow C}\,\textbf{exp_id}$$

$$\frac{\mathcal{P} \triangleright [S_1]_{\Sigma}^{R_1} \longrightarrow [S'_1]_{(\Sigma,\Sigma'_1)}^{R'_1} \quad \mathcal{P} \triangleright [S_2]_{\Sigma}^{R_2} \Longrightarrow [S'_2]_{(\Sigma,\Sigma'_2)}^{R'_2}}{\mathcal{P} \triangleright [S_1, S_2]_{\Sigma}^{(R_1,R_2)} \Longrightarrow [S'_1, S'_2]_{(\Sigma,\Sigma'_1,\Sigma'_2)}^{(R'_1,R'_2)}}\,\textbf{exp_par}$$

Technically, rule **exp_par** operates as follows: it splits the current state in two parts S_1 and S_2, and similarly partitions the active role set into R_1 and R_2. The left premise applies one atomic execution step to the partial snapshot $[S_1]_{\Sigma}^{R_1}$, producing $[S'_1]_{(\Sigma,\Sigma'_1)}^{R'_1}$ as a result, where Σ'_1 is the amount by which the current signature Σ has been extended during this step. Rather than having an unbounded number of premises, we call our parallel firing judgment recursively in the rightmost premise: in zero or more atomic steps, we transform the remainder of the initial snapshot, $[S_2]_{\Sigma}^{R_2}$, into $[S'_2]_{(\Sigma,\Sigma'_2)}^{R'_2}$ where again Σ'_2 is the resulting signature extension. The overall output snapshot is constructed by taking the multiset union of the returned states S'_1 and S'_2, juxtaposing the

output active role sets R'_1 and R'_2, and combining the resulting signatures (Σ, Σ'_1) and (Σ, Σ'_2) into $(\Sigma, \Sigma'_1, \Sigma'_2)$.

It should be noted that the input snapshots to the premises of rule **exp_par** have no state element or active role in common: they are independent. They both rely on the same signature Σ since unrelated state objects and active roles may in general refer to the same constants. The application of a basic step can at most extend the initial signature Σ (see the Signature Extension Lemma 5.1 and Lemma 5.2 below), which justifies expressing the signature output by the two premises as (Σ, Σ_i), for $i = 1, 2$. We can always choose the newly generated names so that Σ_1 and Σ_2 do not declare the same constant: under this assumption, $(\Sigma, \Sigma_1, \Sigma_2)$ is structurally a well-formed signature.

The last judgment we will consider iterates one-step parallel firings. It is expressed as follows:

$$\mathcal{P} \triangleright C \Longrightarrow^* C' \qquad \textit{Multi-step parallel firing}$$

Similarly to multi-step sequential firing, it is obtained by taking the reflexive and transitive closure of its one-step restriction:

$$\frac{}{\mathcal{P} \triangleright C \Longrightarrow^* C}\,\textbf{exp_it0} \qquad \frac{\mathcal{P} \triangleright C \Longrightarrow C' \quad \mathcal{P} \triangleright C' \Longrightarrow^* C''}{\mathcal{P} \triangleright C \Longrightarrow^* C''}\,\textbf{exp_itn}$$

We now extend the Signature Extension Lemma 5.1 to parallel firing. Again, we write $\mathcal{P} \triangleright C \Longrightarrow^{(*)} C'$ to indicate that we are considering the one-step ($\mathcal{P} \triangleright C \Longrightarrow C'$) and the multi-step ($\mathcal{P} \triangleright C \Longrightarrow^* C'$) versions of the parallel firing judgment at once.

Lemma 5.2 (*Signature Extension*)

Let Σ and Σ' be signatures, $\mathcal{P}$ a protocol theory, S and S' states, and R and R' active role sets. If $\mathcal{P} \triangleright [S]^R_\Sigma \Longrightarrow^{(*)} [S']^{R'}_{\Sigma'}$, then $\Sigma' = (\Sigma, \Sigma'')$ for some signature Σ''.

Proof: This result is proved by induction hypothesis on the given premise. An appeal to Lemma 5.1(2) and to the induction hypothesis is needed to justify the form of the premises of rule **exp_par**. □

5.4 Admissibility of Parallel Firing

We now turn to proving that parallel firing can be emulated by sequential execution. The proof of this property relies on a number of simple lemmas, the first of which, given below, states that term typing is invariant with respect to weakening: if a term is typable in a signature, it remains typable when considering additional declarations.

Lemma 5.3 (*Weakening Term Typing*)

Let (Σ, Σ'') *be a signature,* t *a term and* τ *a type. If* $(\Sigma, \Sigma'') \vdash t : \tau$, *then* $(\Sigma, \Sigma', \Sigma'') \vdash t : \tau$ *for any signature* Σ'.

Proof: The proof proceeds by induction on the structure of a derivation $\mathcal{T}$ of the given judgment. It is uncondionally valid when the last (and only) rule of $\mathcal{T}$ is **mpt_a**. It relies on immediate appeals to the induction hypothesis in the other cases (rules **mtp_ss**, **mtp_cnc**, **mtp_ske** and **mtp_pke**). □

We now come to the central lemma in the proof of the admissibility of the rules for parallel firing. It asserts that not only signatures, but also states and active role sets can be weakened in an execution judgment, without influencing its derivability. The first result in this lemma is needed in the proof of the remaining two.

Lemma 5.4 (*Weakening Execution*)

Let Σ *and* Σ'' *be signatures, rhs the right-hand side of a rule, lhs a predicate sequence,* $\mathcal{P}$ *be a protocol theory,* S *and* S'' *states, and* R *and* R'' *active role sets. For any signature* Σ', *state* S' *and active role set* R',

(i) *if* $(rhs)_\Sigma \gg (lhs)_{\Sigma,\Sigma''}$, *then* $(rhs)_{\Sigma,\Sigma'} \gg (lhs)_{\Sigma,\Sigma',\Sigma''}$.

(ii) *if* $\mathcal{P} \triangleright [S]^R_\Sigma \longrightarrow^{(*)} [S'']^{R''}_{\Sigma,\Sigma''}$, *then* $\mathcal{P} \triangleright [S, S']^{R,R'}_{\Sigma,\Sigma'} \longrightarrow^{(*)} [S', S'']^{R',R''}_{\Sigma,\Sigma',\Sigma''}$.

(iii) *if* $\mathcal{P} \triangleright [S]^R_\Sigma \Longrightarrow^{(*)} [S'']^{R''}_{\Sigma,\Sigma''}$, *then* $\mathcal{P} \triangleright [S, S']^{R,R'}_{\Sigma,\Sigma'} \Longrightarrow^{(*)} [S', S'']^{R',R''}_{\Sigma,\Sigma',\Sigma''}$.

Proof: We proceed by induction on the structure of a derivation for the antecedent of each result. In (1), the property is immediate for rule **exs_seq** and is obained by application of the induction hypothesis for rule **exs_nnc**. We proceed similarly in the case of (2), except that we make use of (1) for rule **exs_rule** and of Lemma 5.3 for rules **exs_grole** and **exs_all**. The treatment of (3) is again similar: it should be noted that the state and active role sets extensions (S' and R' respectively) can be split arbitrarily in the premises of rule **exp_par**. □

We can now prove our admissibility result: an arbitrary parallel execution can be simulated by sequential firing sequences. Unsurprisingly, the reverse of this property holds as well: any sequential execution can be lifted to a (degenerate) form of parallel firing.

Theorem 5.5 (*Admissibility of Parallel Execution*)

Let $\mathcal{P}$ *be a protocol theory, and* C *and* C' *two snapshots. If* $\mathcal{P} \triangleright C \Longrightarrow^{(*)} C'$, *then* $\mathcal{P} \triangleright C \longrightarrow^{*} C'$. *Viceversa, if* $\mathcal{P} \triangleright C \longrightarrow^{(*)} C'$, *then* $\mathcal{P} \triangleright C \Longrightarrow^{(*)} C'$.

Proof: In its forward direction, the proof of this result proceeds by induction on the structure of a given derivation $\mathcal{E}$ of the judgment $\mathcal{P} \triangleright C \Longrightarrow^{(*)} C'$. We will examine the one-step case in detail, but omit the simple proof of its

multi-step extension. We proceed by cases on the last rule of $\mathcal{E}$: for single-step parallel firing, only rules **exp_id** and **exp_par** need to be considered.

exp_id $\quad \mathcal{E} = \dfrac{}{\mathcal{P} \triangleright C \Longrightarrow C}\,\textbf{exp_id}$

with $C' = C$.

The result is obtained by instantiating rule **exs_it0** with the same parameters.

exp_par

$$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\mathcal{P} \triangleright [S_1]^{R_1}_{\Sigma} \longrightarrow [S'_1]^{R'_1}_{\Sigma,\Sigma'_1}} \qquad \overset{\mathcal{E}_2}{\mathcal{P} \triangleright [S_2]^{R_2}_{\Sigma} \Longrightarrow [S'_2]^{R'_2}_{\Sigma,\Sigma'_2}}}{\mathcal{P} \triangleright [S_1, S_2]^{R_1,R_2}_{\Sigma} \Longrightarrow [S'_1, S'_2]^{R'_1,R'_2}_{\Sigma,\Sigma'_1,\Sigma'_2}}\,\textbf{exp_par}$$

with $C = [S_1, S_2]^{R_1,R_2}_{\Sigma}$ and $C' = [S'_1, S'_2]^{R'_1,R'_2}_{\Sigma,\Sigma'_1,\Sigma'_2}$.

The proof of this case consists of the following transformations, where the first column gives a name to the derivation of the judgment in the second column:

$\mathcal{E}'_1$	$:: \mathcal{P} \triangleright [S_1, S_2]^{R_1,R_2}_{\Sigma} \longrightarrow [S'_1, S_2]^{R'_1,R_2}_{(\Sigma,\Sigma'_1)}$	by the Weakening Lemma 5.4(2) on $\mathcal{E}_1$,
$\mathcal{E}^*_2$	$:: \mathcal{P} \triangleright [S_2]^{R_2}_{\Sigma} \longrightarrow^* [S'_2]^{R'_2}_{(\Sigma,\Sigma'_2)}$	by induction hypothesis on $\mathcal{E}_2$,
$\mathcal{E}'_2$	$:: \mathcal{P} \triangleright [S'_1, S_2]^{R'_1,R_2}_{\Sigma,\Sigma'_1} \longrightarrow^* [S'_1, S'_2]^{R'_1,R'_2}_{(\Sigma,\Sigma'_1,\Sigma'_2)}$	by the Weakening Lemma 5.4(3) on $\mathcal{E}^*_2$,
$\mathcal{E}'$	$:: \mathcal{P} \triangleright [S_1, S_2]^{R_1,R_2}_{\Sigma} \longrightarrow^* [S'_1, S'_2]^{R'_1,R'_2}_{(\Sigma,\Sigma'_1,\Sigma'_2)}$	by rule **exs_itn** on $\mathcal{E}'_1$ and $\mathcal{E}'_2$.

Observe that a single-step parallel firing is mapped to a multi-step sequential execution. The multi-step parallel case builds on the result we just proved and is obtained by a simple induction on the derivation $\mathcal{E}$.

The reverse direction of this theorem is proved as follows: we wrap each one-step sequential firing with one application of rule **exp_par** by means of an instance of rule **exp_id** with empty state and active role set. The multi-step case mimicks the corresponding sequential construction with parallel iteration rules. □

It should be observed that this proof does not operate directly on the rules that implement the one-step sequential execution judgment. Therefore, this relation can be altered at will without invalidating this theorem, as long as Lemma 5.4 still holds.

This result allows us to focus our efforts on the rules for sequential execution: any property proved valid for the sequential case can immediately be ported to the parallel setting by using the appropriate direction of the above theorem. This does not mean, however, that we should do without parallel execution and eliminate its rules from our formalization: this form of execu-

tion is more faithful to the behavior of protocols in a distributed environment and therefore constitutes a more adequate model of their execution. The fact that parallel execution can be serialized does not imply that the resulting sequential behavior is better and should be adopted.

5.5 Changes

As done at the end of the previous major subdivisions of this paper, we conclude this section with a discussions of the changes in the execution semantics of *MSR* with respect to previous versions of this formalism, as presented in [8,9].

(i) In our previous work, the basic execution step of *MSR* consisted in the application of a protocol rule: instantiation of the variables appearing in its left-hand side happened by pattern matching with the current state, fresh constants were substituted for the existential parameters in its consequent and any additional variable was instantiated before installing the resulting right-hand side back into the state. Since rules were not threaded, there was no need to load an active role set before executing them, nor was there any point in "skipping" a rule. The execution model discussed in this paper has clearly a finer grain. This is due to two reasons: on the one hand, our more detailed specification calls for a precise description of how execution actually happens. In particular, we make the instantiation of each variable individually observable (rule **exs_all**) and separate this process from the actual application of a rule (rule **exs_rule**). On the other hand, special provisions are required to handle some novelties of our syntax, in particular the possible threading of protocol rules in a role (rule **exs_skp**), the presence of a distinguished role owner (rules **exs_arole** and **exs_grole**), and the notion of active role (rule **exs_dot**).

(ii) In [8,9], the universal variables (then implicitly quantified) were instantiated by pattern matching at the time a rule was applied. Our current model is apparently more non-deterministic in that it relies on some guess work to instantiated these objects before the current state is accessed. As already said, this model is purposefully abstract and not intended as the basis for an implementation. We should however observe that [8,9] did not discuss how variables that appear only in the consequent of a rule ought to be instantiated, while our current proposal treats them transparently. This is the subject of future work.

(iii) As mentioned in Section 4, our earlier work on *MSR* demanded that the graph obtained by chasing the role state predicate names in a specification be acyclic. This eliminated the possibility of roles executions that required an unbounded number of steps, a major simplification in the proofs of the decidability results of [8,15]. Our present syntax admits roles where the corresponding graphs contain cycles. It should however

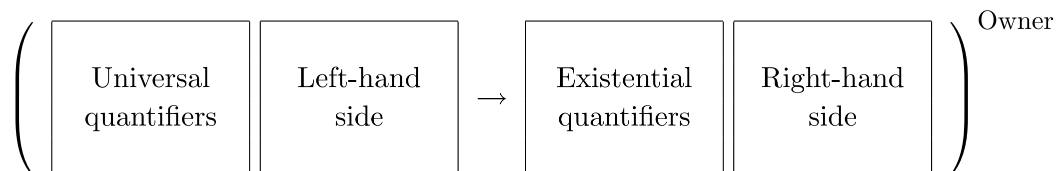
be noted that our execution model prevents taking advantage of them to implement a looping behavior: parametricity makes role state predicates unique to each individual invocation of a role. The rules in an active role, which realize this property, are however processed sequentially without any possibility of jumping back. Still, iteration is possible in *MSR* thanks to the introduction of memory predicates. This will be demonstrated in Section 6.

(iv) In the previous versions of *MSR*, the notion of execution was limited to sequential firing of rules. Parallel execution is therefore a novelty of this paper.

6 Intruder Models

In this section, we will demonstrate the expressive power of *MSR* by formalizing what has come to be accepted as the standard abstraction of the attacker: the Dolev-Yao intruder [20,14]. More precisely, we define the Dolev-Yao abstraction and the natural intruder model it induces in Section 6.1. We then provide a direct *MSR* encoding in Section 6.2. We conclude in Section 6.3 by presenting an optimized variant of the Dolev-Yao intruder model and hinting at its correctness.

Differently from what happens when specifying actual security protocols, all the roles we will need in this paper consist of a single rule. We will represent a role using the format in the following diagram, where we find it convenient to list the elements of its four blocks in columns.



We will mark types that can be reconstructed from the other information present in a rule by denoting them in a shaded font. We will provide informal reasons in support of these elision, but we reserve a formal treatment of type reconstruction in the context of *MSR* to a future publication.

6.1 The Dolev-Yao Intruder Model

The *Dolev-Yao abstraction* of a crypto-protocol appears to be drawn from positions taken in [20] and from a simplified model presented in [14]. It assumes that such elementary data as principal names, keys and nonces are atomic rather than strings of bits, as implemented in practice. Furthermore, it views the operations needed to assemble messages, i.e. concatenation, encryption and digital signature, as pure constructors in an initial algebra. Therefore, if n is a nonce and k a key, $\{n\}_k$ is a composite object whose structure is clearly recognizable. This means for example that a term of the form $\{t\}_k$ cannot be

mistaken for a concatenation $(t_1\,t_2)$, and that $\{t\}_k = \{t'\}_{k'}$ if and only if $t = t'$ and $k = k'$. This also means that the Dolev-Yao model abstracts away the details of the cryptographic algorithms in use, reducing in this way encryption and decryption to atomic operations. Indeed, it is often said to adopt a *black box* view on cryptography.

The atomicity and initiality of the Dolev-Yao abstraction limits considerably the attacks that can be mounted against a protocol. In particular, its idealized encryption model makes it immune to any form of crypto-analysis: keys cannot be exhaustively searched, piecewise inferred from observed traffic, or guessed in any other manner. An encrypted message can be deciphered only when in possession of the appropriate key. The symbolic nature of this abstraction allows then to very precisely circumscribe the operations an intruder has at his disposal to enact an attack against a protocol. All together, they define what has become to be known as the *Dolev-Yao intruder*. This attacker can do any combination of the following eight operations that we find convenient to organize in the five lines below:

- Intercept and learn messages
- Transmit known messages
- Decompose concatenated messages he has learnt
- Concatenate known messages
- Decipher encrypted messages if he knows the keys
- Encrypt known messages with known keys
- Access public information
- Generate fresh data

The first line implies that the Dolev-Yao intruder has complete control of the network. This is clearly a worst case scenario.

MSR is a clear instance of the the Dolev-Yao abstraction. Elementary data are indeed atomic, message are constructed by applying symbolic operators, and the criterion for identifying terms is plain syntactic equality. We will now give a specification of the Dolev-Yao intruder in *MSR*. More specifically, we describe the attacker discussed above in this section. In Section 6.3, we will give an encoding to an optimized version of this attacker proposed in [19] and formalized using a previous version of *MSR* in [8].

6.2 An MSR Specification of the Dolev-Yao Intruder

It has been proved that there is no advantage in considering more than one Dolev-Yao adversary in any given system [23]. Therefore, we select a principal, I say, and endow him with the powers of the Dolev-Yao intruder.

Since the intruder can learn and manipulate information, he must be able to store data out of sight from other principals. This is easily achieved in *MSR* by associating I with a memory predicate $\mathsf{M}_\mathsf{I}(_)$ whose single argument can hold a message. A signature for this model must therefore contain the

following two declarations:

$$\mathsf{I} : \mathsf{principal} \qquad \text{and} \qquad \mathsf{M}_ : \mathsf{principal} \times \mathsf{msg}$$

On the basis of this declarations, we will express each of the intruder's capabilities as one or more roles consisting of a single rule. Each of these roles will be anchored at I and altogether model the actions that the Dolev-Yao intruder can undertake to mount an attack. Thus, the knowledge of the intruder is represented in a distributed fashion as a collection of memory predicates of the form $\mathsf{M_I}(t)$ for all known terms t.

The first line of the description of the Dolev-Yao intruder is then expressed by the following two roles, anchored on I. With the rule on the left, the intruder can capture a network message $\mathsf{N}(t)$ and store its contents in a memory predicate. Observe that the execution semantics of *MSR* implies that $\mathsf{N}(t)$ is removed from the current state and therefore this message is not available any more to the principal it was supposed to reach. The rule on the right emits a memorized message out in the public network.

$$\big(\forall t : \mathsf{msg}.\ \mathsf{N}(t) \ \rightarrow\ \mathsf{M_I}(t)\big)^{\mathsf{I}} \qquad \big(\forall t : \mathsf{msg}.\ \mathsf{M_I}(t) \ \rightarrow\ \mathsf{N}(t)\big)^{\mathsf{I}}$$

From now on, we will only be concerned with the memory predicate $\mathsf{M_I}$, which acts as a workshop where the intruder can dismantle intercepted communications and counterfeit messages. Concatenated messages do not offer any barrier to the intruder: he can take them apart at will. Similarly he can construct the concatenation of any two messages he knows. This is realized by the following two rules:

$$\left(\forall t_1, t_2 : \mathsf{msg}.\ \mathsf{M_I}(t_1\ t_2) \ \rightarrow\ \begin{matrix}\mathsf{M_I}(t_1)\\ \mathsf{M_I}(t_2)\end{matrix}\right)^{\mathsf{I}}$$

$$\left(\forall t_1, t_2 : \mathsf{msg}.\ \begin{matrix}\mathsf{M_I}(t_1)\\ \mathsf{M_I}(t_2)\end{matrix} \ \rightarrow\ \mathsf{M_I}(t_1\ t_2)\right)^{\mathsf{I}}$$

The third line of the specification of the Dolev-Yao intruder, at the beginning of this section, states that I must know the appropriate decryption keys in order to access the contents of an encrypted message. Dually, he must be in possess of the correct key in order to perform an encryption. The following two rules formalize this requirement in *MSR* in the case of shared-key codings:

$$\left(\begin{matrix}\forall A, B : \mathsf{principal}.\\ \forall k : \mathsf{shK}\ A\ B.\\ \forall t : \mathsf{msg}.\end{matrix}\ \begin{matrix}\mathsf{M_I}(\{t\}_k)\\ \mathsf{M_I}(k)\end{matrix} \ \rightarrow\ \mathsf{M_I}(t)\right)^{\mathsf{I}}$$

$$\left(\begin{matrix}\forall A, B : \mathsf{principal}.\\ \forall k : \mathsf{shK}\ A\ B.\\ \forall t : \mathsf{msg}.\end{matrix}\ \begin{matrix}\mathsf{M_I}(t)\\ \mathsf{M_I}(k)\end{matrix} \ \rightarrow\ \mathsf{M_I}(\{t\}_k)\right)^{\mathsf{I}}$$

Both for taking apart and constructing a shared-key encrypted message, the intruder must know the key. Observe that most typing information can be inferred assuming that the specification at hand it well-typed [4]. Take for example the role on the left. In order for the message $\{t\}_k$ to be well-formed, k must be a shared key between two principals. Thus its type must be $\mathsf{shK}\,A\,B$ and the type of the dependent arguments A and B has to be $\mathsf{principal}$. An algorithmic description of how type reconstruction is performed is the subject of future work.

The treatment of public-key cryptography is similar. Notice that here the intruder must have access to a private key for decrypting, while the public key is sufficient for generating encrypted messages.

$$\begin{pmatrix} \forall A : \mathsf{principal}. & & & \\ \forall k : \mathsf{pubK}\ A. & \mathsf{M_I}(\{\!\{t\}\!\}_k) & \to & \mathsf{M_I}(t) \\ \forall k' : \mathsf{privK}\ k. & \mathsf{M_I}(k') & & \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}}$$

$$\begin{pmatrix} \forall A : \mathsf{principal}. & \mathsf{M_I}(t) & & \\ \forall k : \mathsf{pubK}\ A. & \mathsf{M_I}(k) & \to & \mathsf{M_I}(\{\!\{t\}\!\}_k) \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}}$$

Again, most typing information can be reconstructed on the base of the structure of the terms present in this rule. The reasoning proceeds as in the case of shared keys.

We now tackle the often overlooked fourth line of the Dolev-Yao intruder specification above: the ability to access public information. He should clearly be entitled to look up the name and public keys of principals, but any attempted access to more sensitive information such as private keys should be forbidden. The clear exception to this rule consists of the keys he is independently entitled to look up, namely his own private key and keys he shares with other principals. This situation is therefore implemented by the following five rules:

$$\big(\forall A : \mathsf{principal}.\ \cdot \ \to\ \mathsf{M_I}(A)\big)^{\mathsf{I}}$$

$$\begin{pmatrix} \forall A : \mathsf{principal}. & \cdot & \to & \mathsf{M_I}(k) \\ \forall k : \mathsf{shK}\ \mathsf{I}\ A. & & & \end{pmatrix}^{\mathsf{I}} \qquad \begin{pmatrix} \forall A : \mathsf{principal}. & \cdot & \to & \mathsf{M_I}(k) \\ \forall k : \mathsf{shK}\ A\ \mathsf{I}. & & & \end{pmatrix}^{\mathsf{I}}$$

$$\begin{pmatrix} \forall A : \mathsf{principal}. & \cdot & \to & \mathsf{M_I}(k) \\ \forall k : \mathsf{pubK}\ A. & & & \end{pmatrix}^{\mathsf{I}} \qquad \begin{pmatrix} \forall k : \mathsf{pubK}\ \mathsf{I}. & \cdot & \to & \mathsf{M_I}(k') \\ \forall k' : \mathsf{privK}\ k. & & & \end{pmatrix}^{\mathsf{I}}$$

It should be observed that there is a close connection between these rules and the access control policy induced by the term language at hand [3]. In particular, the capabilities of the intruder with respect to public information access always correspond to the strongest rules that satisfy the access control policy.

The last line of the specification of the Dolev-Yao intruder hints at the fact that he should be able to create fresh data. We must be very careful when implementing this requirement: in most scenarios, it is inappropriate for I to generate a shared key k^* between two principals A and B since this would result in unwanted trivial attacks where A and B use k^* instead of their legitimate shared key $\mathsf{k_{AB}}$ (this would be very close to permitting the intruder to guess keys). In general, we do not allow the intruder to generate keys. Similarly, the adversary should not be entitled to create new principals. Nonces and atomic messages are instead risk-frees. Therefore, we propose the following two rules:

$$\big(\cdot \;\rightarrow\; \exists n : \mathsf{nonce}.\ \mathsf{M_I}(n)\big)^{\mathsf{I}} \qquad\qquad \big(\cdot \;\rightarrow\; \exists m : \mathsf{atm}.\ \mathsf{M_I}(m)\big)^{\mathsf{I}}$$

Observe that the rationale behind these two rules, although reasonable, may conflict with idiosyncrasies of individual protocols. For example, the full version of the Needham-Schroeder protocol specified in [4] may be more accurately validated in the presence of an intruder who can create public keys (but not the corresponding private keys). Therefore, depending on the protocol at hand, any of the following rules may be included in the specification of the Dolev-Yao intruder:

$$\big(\forall A, B : \mathsf{principal}.\ \cdot \;\rightarrow\; \exists k : \mathsf{shK}\ A\ B.\ \mathsf{M_I}(k)\big)^{\mathsf{I}}$$
$$\big(\forall A : \mathsf{principal}.\ \cdot \;\rightarrow\; \exists k : \mathsf{pubK}\ A.\ \mathsf{M_I}(k)\big)^{\mathsf{I}}$$
$$\begin{pmatrix}\forall A : \mathsf{principal}. & & & \\ \forall k : \mathsf{pubK}\ A. & \cdot & \rightarrow & \exists k' : \mathsf{privK}\ k.\ \mathsf{M_I}(k')\end{pmatrix}^{\mathsf{I}}$$

A similar rule for principals seems harder to justify since protocols never generate principals, at least not in the simple setting contemplated in this paper.

We provide our formalization of the Dolev-Yao intruder with two administrative rules to allow him to take full advantage of the above stated capabilities. The rule below on the left allows him to forget information. The more interesting rule on the right permits duplicating and reusing fabricated data.

$$\big(\forall t : \mathsf{msg}.\ \mathsf{M_I}(t) \;\rightarrow\; \cdot\big)^{\mathsf{I}} \qquad\qquad \begin{pmatrix}\forall t : \mathsf{msg}.\ \mathsf{M_I}(t) & \rightarrow & \begin{matrix}\mathsf{M_I}(t)\\ \mathsf{M_I}(t)\end{matrix}\end{pmatrix}^{\mathsf{I}}$$

Last, we have two seldom needed rule schemata that allow the intruder to shuffle data between the knowledge predicate $\mathsf{M_I}(_)$ used above and other memory predicates he may have. It is in general unnecessary for the standard Dolev-Yao intruder to rely on additional memory predicates. Here, we write M'_{I} for a generic intruder memory predicate with $n+1$ arguments. We have the following role patterns, where i is an index between 2 and $n+1$:

$$\begin{pmatrix}\forall t_1 : \tau_1. & & & \\ \cdots & M'_{\mathsf{I}}(t_1, \ldots, t_n) & \rightarrow & \begin{matrix}M'_{\mathsf{I}}(t_1, \ldots, t_n)\\ \mathsf{M_I}(t_i)\end{matrix} \\ \forall t_n : \tau_n. & & & \end{pmatrix}^{\mathsf{I}}$$

$$\begin{pmatrix} \forall t_1 : \tau_1. & & & \\ \quad \cdots \quad & M'_{\mathsf{I}}(t_1, \ldots, t_n) & \rightarrow & M'_{\mathsf{I}}(t_1, \ldots, t'_i, \ldots, t_n) \\ \forall t_n : \tau_n. & \mathsf{M}_{\mathsf{I}}(t'_i) & & \\ \forall t'_i : \tau_i. & & & \end{pmatrix}^{\mathsf{I}}$$

The rule on the right replaces the argument t_i of the memory predicate on its antecedent with t'_i in its consequent.

This concludes the *MSR* formalization of the Dolev-Yao intruder. A few aspects of this encoding deserve to be emphasized:

(i) This specification lies fully within *MSR* and can therefore be adapted, were the protocol at hand to require it. This differs from many specification languages which either hardwire the intruder, or express him in a language different from the protocol under examination. It should be observed that our previous version of *MSR* belonged to this latter class: although the specification of the intruder was subject to the same execution semantics as the other roles, it could not be specified in the same way as regular principals.

(ii) Typing allows a very precise characterization of what the intruder's capabilities actually are. This is clearly manifested in the rule clusters that formalize access to public information and fresh data generation.

(iii) All but the fresh data generation rules can be automatically generated from the term language, and their typing [4] and access control rules [3]. We conjecture that which fresh data generation rules are admissible is dependent on the protocol the intruder is running against. In turn, this may be realized by constraining the fresh data that a principal can generate by imposing a stricter access control policy.

6.3 An Optimized Dolev-Yao Intruder

The role set given in the previous section provides a declarative specification of the capabilities of the Dolev-Yao intruder. It is however too non-deterministic to stand as a basis for the form of model-checking simulation needed to analyze protocols, possibly uncovering attacks. We will now present a more operational variant of the Dolev-Yao intruder that eliminates a useless consequence of this non-determinism: looping behaviors caused by the intruder assembling a message and then taking it apart, and then recomposing it, etc. *ad infinitum.*

Whenever the adversary intercepts a message, we will have him decompose it in its most elementary bits, store them in a dedicated memory predicate, and construct any message intended for transmission from stored elementary terms, without the possibility of undoing his own work. We therefore partition the actions of the intruder in three distinct activities: message decomposition, storage of elementary information, and message construction. This idea was first proposed in [19] and analyzed in an earlier version of *MSR* in [8].

We replace the single memory predicate $\mathsf{M_I}(_)$ used in Section 6.2 with three predicates, $\mathsf{D_I}(_)$, $\mathsf{A_I}(_)$ and $\mathsf{C_I}(_)$. The first is intended to contain messages while they are Decomposed into their elementary constituents. The second holds the Atomic terms learnt in this way. The third is used in the message Construction phase. Our signature shall therefore contain the following four declarations:

$$\begin{array}{r@{\,:\,}l} \mathsf{I} & \mathsf{principal} \\ \mathsf{D}_ & \mathsf{principal} \times \mathsf{msg} \\ \mathsf{A}_ & \mathsf{principal} \times \mathsf{atm} \\ \mathsf{C}_ & \mathsf{principal} \times \mathsf{msg} \end{array}$$

The interception and transmission rules are updated as follows:

$$\big(\forall t : \mathsf{msg}.\ \mathsf{N}(t) \quad \rightarrow \quad \mathsf{D_I}(t)\big)^{\mathsf{I}} \qquad\qquad \big(\forall t : \mathsf{msg}.\ \mathsf{C_I}(t) \quad \rightarrow \quad \mathsf{N}(t)\big)^{\mathsf{I}}$$

Observe that an intercepted message is placed in the decomposition memory predicate since it must be disassembled before its elementary constituents can be used. Dually, only constructed messages can be transmitted over the public network. This is enforced by having a construction predicate in the antecedent of the rule on the right.

The rules that dealt with composite message in Section 6.2 are adapted by replacing the generic $\mathsf{M_I}(_)$ predicate with the appropriate refinement. When decomposing a message, its components may need further disassembling. Dually, messages intended for transmission are built by putting together constructible pieces. Keys constitute an exception to this rule: they are clearly atomic and therefore can be accessed from the $\mathsf{A}_$ predicate where such information is stored. We also need to copy them to the consequent of rules so that the intruder can use them again for other encryptions (as we will see shortly, this optimized scheme does without an explicit duplication rule).

$$\left(\forall t_1, t_2 : \mathsf{msg}.\ \mathsf{D_I}(t_1\ t_2) \quad \rightarrow \quad \begin{array}{l} \mathsf{D_I}(t_1) \\ \mathsf{D_I}(t_2) \end{array}\right)^{\mathsf{I}} \left(\forall t_1, t_2 : \mathsf{msg}.\ \begin{array}{l} \mathsf{C_I}(t_1) \\ \mathsf{C_I}(t_2) \end{array} \quad \rightarrow \quad \mathsf{C_I}(t_1\ t_2)\right)^{\mathsf{I}}$$

$$\left(\begin{array}{l} \forall A, B : \mathsf{principal}. \\ \forall k : \mathsf{shK}\ A\ B. \\ \forall t : \mathsf{msg}. \end{array} \begin{array}{l} \mathsf{D_I}(\{t\}_k) \\ \mathsf{A_I}(k) \end{array} \quad \rightarrow \quad \begin{array}{l} \mathsf{D_I}(t) \\ \mathsf{A_I}(k) \end{array}\right)^{\mathsf{I}}$$

$$\left(\begin{array}{l} \forall A, B : \mathsf{principal}. \\ \forall k : \mathsf{shK}\ A\ B. \\ \forall t : \mathsf{msg}. \end{array} \begin{array}{l} \mathsf{C_I}(t) \\ \mathsf{A_I}(k) \end{array} \quad \rightarrow \quad \begin{array}{l} \mathsf{C_I}(\{t\}_k) \\ \mathsf{A_I}(k) \end{array}\right)^{\mathsf{I}}$$

$$\left(\begin{array}{l} \forall A : \mathsf{principal}. \\ \forall k : \mathsf{pubK}\ A. \\ \forall k' : \mathsf{privK}\ k. \\ \forall t : \mathsf{msg}. \end{array} \begin{array}{l} \mathsf{D_I}(\{\!\{t\}\!\}_k) \\ \mathsf{A_I}(k') \end{array} \quad \rightarrow \quad \begin{array}{l} \mathsf{D_I}(t) \\ \mathsf{A_I}(k') \end{array}\right)^{\mathsf{I}}$$

$$\begin{pmatrix} \forall A : \mathsf{principal}. & \mathsf{C_I}(t) & \rightarrow & \mathsf{C_I}(\{\!\{t\}\!\}_k) \\ \forall k : \mathsf{pubK}\ A. & \mathsf{A_I}(k) & & \mathsf{A_I}(k) \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}}$$

It should be observed that the above rules do not apply to situations where the intruder does not know the decryption key of a ciphered message. We will treat this case shortly.

Once a captured message has been reduced to its atomic constituents, they are memorized in individual $\mathsf{A}_$ predicates by the following rule:

$$\left(\forall a : \mathsf{atm}.\ \mathsf{D_I}(a) \quad \rightarrow \quad \mathsf{A_I}(a)\right)^{\mathsf{I}}$$

The atomicity of the decomposable message in this rule is enforced by assigning type atm to the variable a.

As observed earlier, not all terms can be decomposed into to their atomic constituents. In particular encrypted message cannot be exposed unless the intruder has access to the proper decryption key. The following rule is intended to deal with this situation. Here, a message t being disassembled is promoted as a constructible term. Notice that a copy of t is is kept in the decomposition queue in the eventuality that later captured information may allow breaking t into more elementary pieces.

$$\left(\forall t : \mathsf{msg}.\ \mathsf{D_I}(t) \quad \rightarrow \quad \begin{matrix} \mathsf{D_I}(t) \\ \mathsf{C_I}(t) \end{matrix}\right)^{\mathsf{I}}$$

It should be observed that this rule provides a loophole in the scheme discussed in this section since it allows any message to transit from the decomposition pool to the construction queue without accessing its atomic components. To avoid this, it would be tempting to specialize this rule to the situations where t is indeed an encrypted message. This would however violate the access control policy in [3]. A precise specification that avoids these problems requires modifying the language underlying *MSR* to allow for partial pattern matching and negative patterns, which are the subject of current research.

The next rule makes an atomic component available as a constructible message. Copying is required since this object could be needed again later.

$$\left(\forall a : \mathsf{atm}.\ \mathsf{A_I}(a) \quad \rightarrow \quad \begin{matrix} \mathsf{A_I}(a) \\ \mathsf{C_I}(a) \end{matrix}\right)^{\mathsf{I}}$$

We continue with the rules that allow accessing public information and create fresh data. The changes with respect to the rules presented in the previous section is limited to replacing the memory predicate $\mathsf{M}_$ with $\mathsf{A}_$, since the objects under consideration are atomic.

$$\left(\forall A : \mathsf{principal}.\ \cdot \quad \rightarrow \quad \mathsf{A_I}(A)\right)^{\mathsf{I}}$$

$$\begin{pmatrix} \forall A : \mathsf{principal}. & \cdot & \rightarrow & \mathsf{A_I}(k) \\ \forall k : \mathsf{shK}\ \mathsf{I}\ A. & & & \end{pmatrix}^{\mathsf{I}} \qquad \begin{pmatrix} \forall A : \mathsf{principal}. & \cdot & \rightarrow & \mathsf{M_I}(k) \\ \forall k : \mathsf{shK}\ A\ \mathsf{I}. & & & \end{pmatrix}^{\mathsf{I}}$$

$$\begin{pmatrix} \forall A : \mathsf{principal}. & \cdot & \rightarrow & \mathsf{A_I}(k) \\ \forall k : \mathsf{pubK}\ A. & & & \end{pmatrix}^{\mathsf{I}} \qquad \begin{pmatrix} \forall k : \mathsf{pubK}\ \mathsf{I}. & \cdot & \rightarrow & \mathsf{A_I}(k') \\ \forall k' : \mathsf{privK}\ k. & & & \end{pmatrix}^{\mathsf{I}}$$

$$\left(\cdot \quad \rightarrow \quad \exists n : \mathsf{nonce}.\ \mathsf{A_I}(n) \right)^{\mathsf{I}} \qquad \left(\cdot \quad \rightarrow \quad \exists m : \mathsf{atm}.\ \mathsf{A_I}(m) \right)^{\mathsf{I}}$$

We conclude with the rules handling generic $n+1$-ary memory predicates M'_{I} the intruder may have. We treat them as if the terms they record were coming from the network: we read them into the decomposition queue, and store them back from the construction stack. Clearly, these rules can be specialized whenever we have additional information on the data stored in each argument of M'_{I}. In particular, atomic terms can be read to and from the $\mathsf{A_I}(_)$ predicates. We have the following two schematic roles, where i is an index in $2, \ldots, n+1$:

$$\begin{pmatrix} \forall t_1 : \tau_1. & & & M'_{\mathsf{I}}(t_1, \ldots, t_n) \\ \quad \cdots & M'_{\mathsf{I}}(t_1, \ldots, t_n) & \rightarrow & \mathsf{D_I}(t_i) \\ \forall t_n : \tau_n. & & & \end{pmatrix}^{\mathsf{I}}$$

$$\begin{pmatrix} \forall t_1 : \tau_1. & & & \\ \quad \cdots & M'_{\mathsf{I}}(t_1, \ldots, t_n) & \rightarrow & M'_{\mathsf{I}}(t_1, \ldots, t'_i, \ldots, t_n) \\ \forall t_n : \tau_n. & \mathsf{C_I}(t'_i) & & \\ \forall t'_i : \tau_i. & & & \end{pmatrix}^{\mathsf{I}}$$

It should be noted that we have structured the above rules in such a way that no explicit copying rule is ever needed: whenever atomic or decomposable information is accessed for constructing an outgoing message, we always leave a copy for future use. On a similar note, we omit the deletion rule of Section 6.2: this version of the Dolev-Yao intruder only accumulate information, never eliminates it.

Proving the correctness of this optimized intruder model with respect to the role set presented in Section 6.2 is a rather simple task: indeed, mapping the specialized predicates $\mathsf{D_I}(_)$, $\mathsf{A_I}(_)$ and $\mathsf{C_I}(_)$ back to $\mathsf{M_I}(_)$ yields a set of rules that is almost identical to our original role set for the Dolev-Yao intruder. The minor discrepancies brought about by this translation are corrected by uses of the structural rule of deletion, and the elimination of one redundantly produced rule, whose antecedent and consequent are identical. Alternatively, this property can be seen as an instance of a more general result that states that the Doley-Yao intruder model subsumes every other intruder model that plays by the rules of the Dolev-Yao abstraction. This theorem is formally proved in [3].

The proof of the corresponding completeness result, which shows that our optimized model is powerful enough to simulate the original Dolev-Yao intruder, is instead quite complex on the basis of the definitions given in this paper. The intuitively simple translation of runs into the optimized model is complicated by the need to perform context-dependent permutations between rule applications. Keeping track of firing dependencies among rules, while abstracting from unrelated rules that happen to be interleaved with them, is not handled elegantly in our execution model. Proofs of these forms call for a more abstract view of execution, that chains together cooperating rules. Strand spaces [16] have those qualities and we are currently developing a similar model on top of our current execution rules. The core of this proof in the context of strand spaces can indeed be found in [17].

7 Conclusions and Future Work

In this paper, we have presented the execution model of *MSR*, a strongly-typed framework for the specification of security protocols. *MSR* extends previous attempts at using multiset rewriting for formalizing crypto-protocols [8,15,9] with the introduction of dependent types as a simple and effective mechanism for expressing relations among objects (e.g. between a key and its owner), and by permitting rules to mention memory predicates that allow a principal to store data that survives role termination. The typing infrastructure of *MSR* is discussed in detail in [4] while the related notion of access control is the subject of [3]. Memory predicates are a powerful mechanism that enables the specification of protocols consisting of a set of coordinated subprotocols that exchange data and pass control [5]. The intruder model that best fits the analysis of a given protocol is indeed expressed as a collection of such subprotocols that communicate through dedicated memory predicates. We exemplified this feature by presenting two detailed specifications of the Dolev-Yao intruder [20,14]. The execution model of *MSR* is based on parallel multiset rewriting. We proved the admissibility of this approach with respect to the interleaving model that we considered in our previous work [8,9].

In addition to the execution model presented here, *MSR* also embeds an extensive type system, fully discussed in [4], as well as a set of judgments and rules that formalize the notion of access control [3]. In the near future, we plan to develop *MSR* in three directions. First we want to extend our collection of case studies [5] to encompass not only the most popular authentication protocols [10], but also fair exchange protocols and schemes developed for achieving secure multicast (we are currently formalizing the *OFT* key management policy [2]). Second, we are developing an operational execution model as a basis for simulation and model checking. This involves in particular delaying the instantiation of universal variables to limit non-determinism. Still on the pragmatic terrain, we are devising support for type reconstruction aimed at speeding up the correct formalization of a protocol. Third, we are designing a

flexible strand-like notation to describe protocol runs [16,17]. This is intended to simplify formal reasoning on chains of dependent rule applications while abstracting from non-interfering rules that may happen to be interleaved with them.

References

[1] David Aspinall and Adriana Compagnoni. Subtyping dependent types. In E. Clarke, editor, *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 86–97, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[2] David Balenson, David McGrew, and Alan Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization. Internet Draft (work in progres), draft-irtf-smug-groupkeymgmt-oft-00.txt, Internet Engineering Task Force (August 25, 2000).

[3] Iliano Cervesato. MSR, access control, and the most powerful attacker. Unpublished manuscript. Accessible from the author's web page at `http://www.cs.stanford.edu/~iliano/`.

[4] Iliano Cervesato. A specification language for crypto-protocol based on multiset rewriting, dependent types and subsorting. Unpublished manuscript. Accessible from `http://www.cs.stanford.edu/~iliano/`.

[5] Iliano Cervesato. Typed MSR: Syntax and examples. In V. Gorodetski, V. Skormin, and L. Popyack, editors, *Proceedings of the First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security — MMM'01*, St. Petersburg, Russia, 21–23 May 2001. Springer-Verlag LNCS.

[6] Iliano Cervesato, Nancy A. Durgin, Max Kanovich, and Andre Scedrov. Interpreting strands in linear logic. In *2000 Workshop on Formal Methods and Computer Security — FMCS'00*, Chigaco, IL, July 2000.

[7] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. Long version of [9]; forthcoming.

[8] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In P. Syverson, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, June 1999. IEEE Computer Society Press.

[9] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 35–51, Cambrige, UK, July 2000. IEEE Computer Society Press.

[10] John Clark and Jeremy Jacob. A survey of authentication protocol literature. Technical report, Department of Computer Science, University of York, 1997. Web Draft Version 1.0 available from http://www.cs.york.ac.uk/~jac/.

[11] Ph. de Groote, editor. *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique, Département de Philosophie, Université Catholique de Louvain*. Academia, 1995.

[12] Grit Denker, Jonathan Millen, A. Grau, and J. Filipe. Optimizing protocol rewrite rules of CIL specifications. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 52–62, Cambrige, UK, July 2000. IEEE Computer Society Press.

[13] Grit Denker and Jonathan K. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP*, Trento, Italy, July 1999.

[14] Danny Dolev and Andrew C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[15] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP*, Trento, Italy, July 1999.

[16] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171, Oakland, CA, May 1998. IEEE Computer Society Press.

[17] Joshua Guttman and Javier Thayer Fábrega. Authentication tests and the normal, efficient penetrator. *Theoretical Computer Science*, 2001. To appear.

[18] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[19] Will Marrero, Edmund M. Clarke, and Somesh Jha. Model checking for security protocols. In *Proceedings of the 1997 DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. A Preliminary version appeared as Technical Report TR-CMU-CS-97-139, Carnegie Mellon University, May 1997.

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

[21] B. Clifford Neuman and Stuart G. Stubblebine. A note on the use of timestamps as nonces. *Operating Systems Review*, 27(2):10–14, April 1993.

[22] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[23] Paul Syverson, Catherine Meadows, and Iliano Cervesato. Dolev-Yao is no better than Machiavelli. In P. Degano, editor, *First Workshop on Issues in the Theory of Security — WITS'00*, pages 87–92, Geneva, Switzerland, 7-8 July 2000.

A Collected Rules

This appendix collects the grammatical productions, judgments and rules used throughout the paper for the convenience of the reader. More specifically, the syntax of *MSR* is summarized in Section A.1. The minimal typing infrastructure needed in this paper is displayed in Section A.2. The definition of substitution for *MSR* is given in Section A.3. Finally, we collect the executions judgments of our language and associated inference rules in Section A.4. For the convenience of the reader, we specify the page number where each notion is first introduced.

A.1 Syntax

Atomic messages: a	$::=$	A	*(Principal)*	[p. 4]
	$\mid$	k	*(Key)*	
	$\mid$	n	*(Nonce)*	
	$\mid$	m	*(Raw datum)*	
Parametric messages: t	$::=$	a	*(Atomic messages)*	[p. 4]
	$\mid$	x	*(Variables)*	
	$\mid$	$t_1\ t_2$	*(Concatenation)*	
	$\mid$	$\{t\}_k$	*(Symmetric-key encryption)*	
	$\mid$	$\{\!\{t\}\!\}_k$	*(Asymmetric-key encryption)*	
Message tuples: $\vec{t}$	$::=$	$\cdot$	*(Empty tuple)*	[p. 9]
	$\mid$	$t,\ \vec{t}$	*(Tuple extension)*	
States: S	$::=$	$\cdot$	*(Empty state)*	[p. 11]
	$\mid$	$S,\ \mathsf{N}(t)$	*(Extension with a network predicate)*	
	$\mid$	$S,\ \mathsf{L}_l(\vec{t})$	*(Extension with a role state predicate)*	
	$\mid$	$S,\ \mathsf{M}_{\mathsf{A}}(\vec{t})$	*(Extension with a memory predicate)*	
Types: τ	$::=$	$\mathsf{principal}$	*(Principals)*	[p. 5]
	$\mid$	nonce	*(Nonces)*	
	$\mid$	$\mathsf{shK}\ A\ B$	*(Shared keys)*	
	$\mid$	$\mathsf{pubK}\ A$	*(Public keys)*	
	$\mid$	$\mathsf{privK}\ k$	*(Private keys)*	
	$\mid$	atm	*(Atomic messages)*	
	$\mid$	msg	*(Messages)*	
Type tuples: $\vec{\tau}$	$::=$	$\cdot$	*(Empty tuple)*	[p. 10]
	$\mid$	$\tau^{(x)} \times \vec{\tau}$	*(Type tuple extension)*	
Predicate sequences: lhs	$::=$	$\cdot$	*(Empty predicate sequence)*	[p. 14]
	$\mid$	$lhs,\ \mathsf{N}(t)$	*(Extension with a network predicate)*	
	$\mid$	$lhs,\ L(\vec{e})$	*(Extension with a role state predicate)*	
	$\mid$	$lhs,\ \mathsf{M}_A(\vec{t})$	*(Extension with a memory predicate)*	

Right-Hand sides: rhs	$::= \; lhs$	*(Sequence of message predicates)*	[p. 14]
	$\mid \; \exists x : \tau.\, rhs$	*(Fresh data generation)*	
Rule: r	$::= \; lhs \rightarrow rhs$	*(Rule core)*	[p. 13]
	$\mid \; \forall x : \tau.\, r$	*(Parameter closure)*	
Rule collections: ρ	$::= \; \cdot$	*(Empty role)*	[p. 15]
	$\mid \; \exists L : \vec{\tau}.\, \rho$	*(Parameter declaration)*	
	$\mid \; r,\ \rho$	*(Extension with a rule)*	
Protocol theories: $\mathcal{P}$	$::= \; \cdot$	*(Empty protocol theory)*	[p. 16]
	$\mid \; \mathcal{P},\ \rho^{\forall A}$	*(Extension with a generic role)*	
	$\mid \; \mathcal{P},\ \rho^{A}$	*(Extension with an anchored role)*	
Active role sets: R	$::= \; \cdot$	*(No active role)*	[p. 16]
	$\mid \; R,\ \rho^{\mathsf{A}}$	*(Extension with an instantiated role)*	
Signatures: Σ	$::= \; \cdot$	*(Empty signature)*	[p. 7]
	$\mid \; \Sigma,\ a : \tau$	*(Atomic message declaration)*	
	$\mid \; \Sigma,\ \mathsf{L}_l : \vec{\tau}$	*(Local state predicate declaration)*	[p. 11]
	$\mid \; \Sigma,\ \mathsf{M}_ : \vec{\tau}$	*(Memory predicate declaration)*	[p. 11]
Snapshot: C	$::= \; [S]\,^{R}_{\Sigma}$		[p. 20]

A.2 Typing Rules

$\tau :: \tau'$	τ *is a subsort of* τ'	[p. 6]
$\Sigma \vdash t : \tau$	*Term* t *has type* τ *in signature* Σ	[p. 7]

$\tau :: \tau'$	τ *is a subsort of* τ'	[p. 6]

$$\frac{}{\mathsf{atm} :: \mathsf{msg}}\,\textbf{ss_atm} \qquad \frac{}{\mathsf{principal} :: \mathsf{atm}}\,\textbf{ss_pr} \qquad \frac{}{\mathsf{nonce} :: \mathsf{atm}}\,\textbf{ss_nnc}$$

$$\frac{}{\mathsf{shK}\ A\ B :: \mathsf{atm}}\,\textbf{ss_shK} \qquad \frac{}{\mathsf{pubK}\ A :: \mathsf{atm}}\,\textbf{ss_pbK} \qquad \frac{}{\mathsf{privK}\ k :: \mathsf{atm}}\,\textbf{ss_pvK}$$

$\Sigma \vdash t : \tau$	*Term* t *has type* τ *in signature* Σ	[p. 7]

$$\frac{\Sigma \vdash t_1 : \mathsf{msg} \quad \Sigma \vdash t_2 : \mathsf{msg}}{\Sigma \vdash t_1\, t_2 : \mathsf{msg}}\,\textbf{mtp_cnc}$$

$$\frac{\Sigma \vdash t : \mathsf{msg} \quad \Sigma \vdash k : \mathsf{shK}\ A\ B}{\Sigma \vdash \{t\}_k : \mathsf{msg}}\,\textbf{mtp_ske} \qquad \frac{\Sigma \vdash t : \mathsf{msg} \quad \Sigma \vdash k : \mathsf{pubK}\ A}{\Sigma \vdash \{\!\{t\}\!\}_k : \mathsf{msg}}\,\textbf{mtp_pke}$$

$$\frac{\Sigma \vdash t : \tau' \quad \tau' :: \tau}{\Sigma \vdash t : \tau}\,\textbf{mtp_ss} \qquad \frac{}{(\Sigma, a : \tau, \Sigma') \vdash a : \tau}\,\textbf{mtp_a}$$

A.3 Substitutions

$[t/x]t'$	$[t/x]\tau$
$[t/x]a = a$	$[t/x]\mathsf{principal} = \mathsf{principal}$
$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{otherwise} \end{cases}$	$[t/x]\mathsf{nonce} = \mathsf{nonce}$
$[t/x](t'_1\ t'_2) = ([t/x]t'_1)\ ([t/x]t'_2)$	$[t/x]\mathsf{shK}\ A\ B = \mathsf{shK}\ ([t/x]A)\ ([t/x]B)$
$[t/x]\{t'\}_k = \{[t/x]t'\}_{[t/x]k}$	$[t/x]\mathsf{pubK}\ A = \mathsf{pubK}\ ([t/x]A)$
$[t/x]\{\!\{t'\}\!\}_k = \{\!\{[t/x]t'\}\!\}_{[t/x]k}$	$[t/x]\mathsf{privK}\ k = \mathsf{privK}\ ([t/x]k)$
	$[t/x]\mathsf{atm} = \mathsf{atm}$
	$[t/x]\mathsf{msg} = \mathsf{msg}$

$[t/x]\vec{t}$	$[t/x]\vec{\tau}$
$[t/x]\cdot = \cdot$	$[t/x]\cdot = \cdot$
$[t/x](t',\ \vec{t}) = ([t/x]t'),\ ([t/x]\vec{t})$	$[t/x](\tau^{(y)} \times \vec{\tau}) = ([t/x]\tau)^{(y)} \times ([t/x]\vec{\tau})$

$[t/x]\mathit{lhs}$	$[t/x]\mathit{rhs}$
$[t/x]\cdot = \cdot$	$[t/x]\mathit{lhs} = [t/x]\mathit{lhs}$
$[t/x](\mathit{lhs},\ \mathsf{N}(t)) = ([t/x]\mathit{lhs}),\ \mathsf{N}([t/x]t)$	$[t/x](\exists y : \tau.\ \mathit{rhs})$
$[t/x](\mathit{lhs},\ L(\vec{e})) = ([t/x]\mathit{lhs}),\ L([t/x]\vec{e})$	$= \exists y : ([t/x]\tau).\ ([t/x]\mathit{rhs})$
$[t/x](\mathit{lhs},\ \mathsf{M}_A(\vec{t})) = ([t/x]\mathit{lhs}),\ \mathsf{M}_{[t/x]A}([t/x]\vec{t})$	

$[t/x]r$	$[t/x]\rho$
$[t/x](\mathit{lhs} \rightarrow \mathit{rhs}) = ([t/x]\mathit{lhs}) \rightarrow ([t/x]\mathit{rhs})$	$[t/x]\cdot = \cdot$
$[t/x](\forall y : \tau.\ r) = \forall y : ([t/x]\tau).\ ([t/x]r)$	$[t/x](\exists L : \vec{\tau}.\ \rho) = \exists L : ([t/x]\vec{\tau}).\ ([t/x]\rho)$
	$[t/x](r,\ \rho) = ([t/x]r),\ ([t/x]\rho)$

$[\mathsf{L}_l/L]\mathit{lhs}$	$[\mathsf{L}_l/L]\mathit{rhs}$
$[\mathsf{L}_l/L]\cdot = \cdot$	$[\mathsf{L}_l/L]\mathit{lhs} = [\mathsf{L}_l/L]\mathit{lhs}$
$[\mathsf{L}_l/L](\mathit{lhs},\ \mathsf{N}(t)) = ([\mathsf{L}_l/L]\mathit{lhs}),\ \mathsf{N}(t)$	$[\mathsf{L}_l/L](\exists x : \tau.\ \mathit{rhs})$
$[\mathsf{L}_l/L](\mathit{lhs},\ \mathsf{M}_A(\vec{t})) = ([\mathsf{L}_l/L]\mathit{lhs}),\ \mathsf{M}_A(\vec{t})$	$= \exists x : \tau.\ ([\mathsf{L}_l/L]\mathit{rhs})$
$[\mathsf{L}_l/L](\mathit{lhs},\ L'(\vec{e})) = \begin{cases} ([\mathsf{L}_l/L]\mathit{lhs}),\ \mathsf{L}_l(\vec{e}) & \text{if } L = L' \\ ([\mathsf{L}_l/L]\mathit{lhs}),\ L'(\vec{e}) & \text{otherwise} \end{cases}$	

$[\mathsf{L}_l/L]r$	$[\mathsf{L}_l/L]\rho$
$[\mathsf{L}_l/L](\mathit{lhs} \rightarrow \mathit{rhs}) = ([\mathsf{L}_l/L]\mathit{lhs}) \rightarrow ([\mathsf{L}_l/L]\mathit{rhs})$	$[\mathsf{L}_l/L]\cdot = \cdot$
$[\mathsf{L}_l/L](\forall x : \tau.\ r) = \forall x : \tau.\ ([\mathsf{L}_l/L]r)$	$[\mathsf{L}_l/L](\exists L' : \vec{\tau}.\ \rho) = \exists L' : \vec{\tau}.\ ([\mathsf{L}_l/L]\rho)$
	$[\mathsf{L}_l/L](r,\ \rho) = ([\mathsf{L}_l/L]r),\ ([\mathsf{L}_l/L]\rho)$

A.4 Execution Rules

$(\mathit{rhs})_\Sigma \gg (\mathit{lhs})_{\Sigma'}$	*Right-hand side instantiation*	[p. 22]
$r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}$	*Rule application*	[p. 21]
$\mathcal{P} \triangleright C \longrightarrow C'$	*One-step sequential firing*	[p. 20]
$\mathcal{P} \triangleright C \longrightarrow^* C'$	*Multi-step sequential firing*	[p. 23]
$\mathcal{P} \triangleright C \Longrightarrow C'$	*One-step parallel firing*	[p. 24]
$\mathcal{P} \triangleright C \Longrightarrow^* C'$	*Multi-step parallel firing*	[p. 25]

$(rhs)_\Sigma \gg (lhs)_{\Sigma'}$	*Right-hand side instantiation*	[p. 22]

$$\frac{}{(lhs)_\Sigma \gg (lhs)_\Sigma}\ \textbf{exs_seq} \qquad \frac{([\mathsf{a}/x]rhs)_{(\Sigma,\mathsf{a}:\tau)} \gg (lhs)_{\Sigma'}}{(\exists x:\tau.\, rhs)_\Sigma \gg (lhs)_{\Sigma'}}\ \textbf{exs_nnc}$$

$r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}$	*Rule application*	[p. 21]

$$\frac{\Sigma \vdash t:\tau \quad [t/x]r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}}{(\forall x:\tau.\, r) \triangleright [S]_\Sigma \gg [S']_{\Sigma'}}\ \textbf{exs_all} \qquad \frac{(rhs)_\Sigma \gg (lhs')_{\Sigma'}}{(lhs \to rhs) \triangleright [S, lhs]_\Sigma \gg [S, lhs']_{\Sigma'}}\ \textbf{exs_core}$$

$\mathcal{P} \triangleright C \longrightarrow C'$	*One-step sequential firing*	[p. 20]

$$\frac{}{(\mathcal{P}, \rho^{\mathsf{A}}) \triangleright [S]^{R}_{\Sigma} \longrightarrow [S]^{R,\rho^{\mathsf{A}}}_{\Sigma}}\ \textbf{exs_arole} \qquad \frac{\Sigma \vdash \mathsf{A} : \mathsf{principal}}{(\mathcal{P}, \rho^{\forall A}) \triangleright [S]^{R}_{\Sigma} \longrightarrow [S]^{R,([\mathsf{A}/A]\rho)^{\mathsf{A}}}_{\Sigma}}\ \textbf{exs_grole}$$

$$\frac{}{\mathcal{P} \triangleright [S]^{R,(\exists L:\vec{\tau}.\,\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S]^{R,([\mathsf{L}_l/L]\rho)^{\mathsf{A}}}_{(\Sigma,\mathsf{L}_l:\vec{\tau})}}\ \textbf{exs_rsp} \qquad \frac{r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}}{\mathcal{P} \triangleright [S]^{R,(r,\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S']^{R,(\rho)^{\mathsf{A}}}_{\Sigma'}}\ \textbf{exs_rule}$$

$$\frac{}{\mathcal{P} \triangleright [S]^{R,(r,\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S]^{R,(\rho)^{\mathsf{A}}}_{\Sigma}}\ \textbf{exs_skp} \qquad \frac{}{\mathcal{P} \triangleright [S]^{R,(\cdot)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S]^{R}_{\Sigma}}\ \textbf{exs_dot}$$

$\mathcal{P} \triangleright C \longrightarrow^* C'$	*Multi-step sequential firing*	[p. 23]

$$\frac{}{\mathcal{P} \triangleright C \longrightarrow^* C}\ \textbf{exs_it0} \qquad \frac{\mathcal{P} \triangleright C \longrightarrow C' \quad \mathcal{P} \triangleright C' \longrightarrow^* C''}{\mathcal{P} \triangleright C \longrightarrow^* C''}\ \textbf{exs_itn}$$

$\mathcal{P} \triangleright C \Longrightarrow C'$	*One-step parallel firing*	[p. 24]

$$\frac{}{\mathcal{P} \triangleright C \Longrightarrow C}\ \textbf{exp_id} \qquad \frac{\mathcal{P} \triangleright [S_1]^{R_1}_{\Sigma} \longrightarrow [S'_1]^{R'_1}_{(\Sigma,\Sigma'_1)} \quad \mathcal{P} \triangleright [S_2]^{R_2}_{\Sigma} \Longrightarrow [S'_2]^{R'_2}_{(\Sigma,\Sigma'_2)}}{\mathcal{P} \triangleright [S_1, S_2]^{(R_1,R_2)}_{\Sigma} \Longrightarrow [S'_1, S'_2]^{(R'_1,R'_2)}_{(\Sigma,\Sigma'_1,\Sigma'_2)}}\ \textbf{exp_par}$$

$\mathcal{P} \triangleright C \Longrightarrow^* C'$	*Multi-step parallel firing*	[p. 25]

$$\frac{}{\mathcal{P} \triangleright C \Longrightarrow^* C}\ \textbf{exp_it0} \qquad \frac{\mathcal{P} \triangleright C \Longrightarrow C' \quad \mathcal{P} \triangleright C' \Longrightarrow^* C''}{\mathcal{P} \triangleright C \Longrightarrow^* C''}\ \textbf{exp_itn}$$