A Specification Language for Crypto-Protocols based on Multiset Rewriting, Dependent Types and Subsorting *

Iliano Cervesato

Advanced Engineering and Sciences Division
ITT Industries, Inc.
Alexandria, VA 22303-1410
iliano@itd.nrl.navy.mil

Abstract

MSR is an unambiguous, flexible, powerful and relatively simple specification framework for crypto-protocols. It uses multiset rewriting rules over first-order atomic formulas to express protocol actions and relies on a form of existential quantification to symbolically model the generation of nonces and other fresh data. It supports an array of useful static checks that include type-checking and data access verification. In this paper, we give a detailed presentation of the typing infrastructure of *MSR*, which is based on the theory of dependent types with subsorting. We prove that type-checking protocol specifications is decidable and show that execution preserves well-typing. We illustrate these features by formalizing a well-known protocol in *MSR*.

Keywords: Protocol specification, dependent types with subsorting, decidability of type checking.

* Partially supported by NRL under contract N00173-00-C-2086.

1 Introduction

Cryptographic protocols are widely used to secure transactions over the Internet and protect access to computer systems. Their design and analysis are notoriously complex and error-prone. Sources of difficulty include subtleties in the cryptographic primitives they rely on, their deployment in distributed environments plagued by powerful and opportunistic attackers, and the use of ambiguous, complex or inexpressive languages for their specification. Most systems designed for protocol analysis, *e.g.* [5, 16, 15, 1, 12] circumvent the first issue by relying on an idealization known as the Dolev-Yao model of security [17, 13]: the cryptography is assumed to be flawless, which permits viewing message-forming operations such as encryption as symbolic combinators ultimately applied to atomic abstractions of principal names, keys, nonces, etc, rather than to bitstrings. Within the Dolev-Yao model, the capabilities of the intruder are circumscribed. In ideal situations, they can be in many respects neutralized by relying on appropriate message formats [2]. Yet, in spite of these methodological advances, faulty designs and contradictory analyses abound. In many instances, we can link these failures to poor specification languages. Indeed, the still universal "usual notation", which describes a protocol as the sequence of the messages transmitted during an expected run, expresses fundamental assumptions and requirements about message components, the operating environment and the protocol's goals as potentially ambiguous side remarks in natural language. The increasingly popular strand formalization [15] also relies on a fair amount of natural language; like most modern languages, it however views protocols as a collection of independent roles that communicate by exchanging messages. Formalisms that ban natural language are typically geared toward studying properties of large classes of security protocols rather than writing specifications of individual protocols, for which they tend to be complex or inexpressive.

MSR originated as a simple logic-oriented language aimed at investigating the decidability of protocol analysis under a variety of assumptions [9, 14]. It evolved into a precise, powerful, flexible, and still relatively simple framework for the specification of complex cryptographic protocols, possibly structured as a collection of coordinated subprotocols [6, 7]. It uses strongly-typed multiset rewriting rules over first-order atomic formulas to express protocol actions and relies on a form of existential quantification to symbolically model the generation of nonces and other fresh data. It supports an array of useful static checks that include type-checking and data access verification.

In this paper, we give a detailed presentation of the flexible typing infrastructure of *MSR*, which significantly contributes to the expressiveness and simplicity of this language. Type annotations, drawn from the theory of dependent types with subsorting, enable precise object classification 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 the keys they use for digital signature. The type taxonomy is open-ended and can be tailored to best fit the protocol at hand. The typing policy of *MSR* allows not only verifying basic well-formedness conditions (*e.g.* that no message is encrypted with a nonce), but also provide a statically checkable way to enforce more complex requirements such as, for example, that long-term keys are never transmitted as part of messages. We prove that type-checking protocol specifications is decidable and show that execution preserves well-typing.

The typing infrastructure of *MSR* supports data access verification procedures for statically enforcing complex requirements such as, for example, that no principal may encrypt a message with a key he/she is not entitled to look up, or that he/she will not access information from the private state of another principal [6]. Moreover, typing yields a very precise representation of the Dolev-Yao intruder, and ultimately allows proving that it is the most powerful attacker that satisfies the access verification policy [6].

This paper aims at formally defining the typing infrastructure of *MSR*, and studying its main properties. In this, it complements [7] which concentrated on the syntax and examples of use, and prepares the ground for [6] which will be mostly concerned with the notion of data access verification and its relation to the Dolev-Yao intruder [13]. Our current endeavor focuses on the *specification* of cryptographic protocols. Although we expect that current verification techniques can be adapted to *MSR*, this paper is not about how to perform protocol *analysis* with *MSR*.

This paper is organized as follows: in Section 2 we introduce the term language of *MSR* together with the relative types and typing rules. In Section 3, we concentrate on the notion of state, while Section 4 examines the typing infrastructure of rules and protocol theories. Section 5 defines the execution model of *MSR*. In Section 6, we formalize the Otway-Rees authentication protocol [18] as an example. Section 7 outlines directions of future work.

2 Typed Messages

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.* information 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:

$$a \ ::= \ \mathsf{A} \ \mid \ \mathsf{k} \ \mid \ \mathsf{n} \ \mid \ \mathsf{m}$$

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 [7].

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

$$t \ ::= \ a \ \mid \ x \ \mid \ t_1\, t_2 \ \mid \ \{t\}_{\mathsf{k}} \ \mid \ \{\!\{t\}\!\}_{\mathsf{k}}$$

where a and x range over atomic terms and variables, respectively. 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 ($\{t\}_{\mathsf{k}}$ and $\{\!\{t\}\!\}_{\mathsf{k}}$ respectively).

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 [7]. We refrain from doing so since their inclusion would lengthen the discussion without introducing substantially new concepts.

Type tuples (discussed in Section 2.2) and protocol rules (see Section 4.1) rely on objects that may contain variables to be instantiated during type-checking and execution, respectively. A *parametric message* allows variables wherever terms could appear. 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, 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 variables. 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. Finally, we write $[t/x]t'$ for the substitution of a variable x with a term t in another term t'.

2.2 Types

While types played a very modest role in the original definition of *MSR* [9, 10], they stand at the core of the extension presented in this paper. Through typing, we enforce not only basic well-formedness conditions (*e.g.* that only keys be used for encrypting a message), but also more subtle requirements, such as that long term keys never be transmitted in messages. Furthermore, types 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 extensively analyzed in [6]. 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*, that cluttered protocol specifications and complicated reasoning about them [10].

The typing machinery that best fits our goals is based on the type-theoretic notion of *dependent product types with subsorting* [3, 19]. 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 will not conduct an in-depth discussion of this type theory; we will even stay away from the most exotic aspects of its syntax.

The types we will use in this paper are summarized in the following grammar:

$$\begin{array}{rcl} \tau & ::= & \mathsf{princ} \mid \mathsf{nonce} \mid \mathsf{shK}\,A\,B \mid \mathsf{pubK}\,A \mid \mathsf{privK}\,k \\ & \mid & \mathsf{stK}\,A\,B \mid \mathsf{ltK}\,A\,B \mid \mathsf{msg} \end{array}$$

In the sequel, τ, possibly variously decorated, will stand for a type. Needless to say, the types "princ" 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". We will use the next two productions above in the example in Section 6 to distinguish between short-term (type "stK A B") and long-term (type "ltK A B") shared keys.

We will use the type msg to classify generic messages. Clearly raw data will have type msg. This is however not sufficient since nonces, keys, and principal identifiers are routinely part of messages. We solve this problem by imposing a *subsorting* relation between types. We formalize this relation by means of the *judgment* "$\tau :: \tau'$", to be read "τ *is a subsort of* τ'". In this paper, the subsorting relation will amount to having each of the types discussed above, with the exception of "shK A B" and "ltK A B", be a subtype of msg. Moreover, long-term and short-term key types will be subsorts of the type for shared keys with the same arguments. Its extension is expressed by the following rules:

$$\frac{}{\mathsf{princ} :: \mathsf{msg}}\,\textbf{ss\_pr} \qquad \frac{}{\mathsf{nonce} :: \mathsf{msg}}\,\textbf{ss\_nnc} \qquad \frac{}{\mathsf{stK}\ A\ B :: \mathsf{msg}}\,\textbf{ss\_stK}$$

$$\frac{}{\mathsf{stK}\ A\ B :: \mathsf{shK}\ A\ B}\,\textbf{ss\_stshK} \qquad \frac{}{\mathsf{ltK}\ A\ B :: \mathsf{shK}\ A\ B}\,\textbf{ss\_ltshK}$$

$$\frac{}{\mathsf{pubK}\ A :: \mathsf{msg}}\,\textbf{ss\_pbK} \qquad \frac{}{\mathsf{privK}\ k :: \mathsf{msg}}\,\textbf{ss\_pvK}$$

The fact that types of the form "ltK $A B$" are not a subsort of msg prohibits the transmission of long-term secrets as parts of messages. On the other hand, making "stK A B" and "ltK $A B$" subsorts of "shK $A B$" provides a convenient way to factor out properties common to those two key types.

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 can be accommodated by introducing dedicated dependent types akin to "pubK A" and "privK k" [7].

2.3 Typing

We will now present the typing rules that allow us to establish whether an expression built according to the syntax of terms can be considered a message (more in general whether a given term has a certain type).

The rules below systematically reduce the typability of a composite term to the validity of its subterms. The type of elementary terms is instead checked against a *context* Γ, *i.e.* a finite sequence of declarations that map elementary messages to their type. More formally,

$$\Gamma \ ::= \ \cdot \ \mid \ \Gamma, a : \tau \ \mid \ \Gamma, x : \tau \ \mid \ \ldots$$

Here "$\cdot$" represents an empty context. The dots express the fact that this definition is incomplete: we will extend it in the next section.

We assume that each elementary message in a context is declared exactly once. We will also often elide the leading "$\cdot$" from a non-empty context, and promote the extension operator "," to denote context union. This operation is defined only if the resulting sequence is itself a context (in particular it should not contain multiple declaration for the same object). In several circumstances, we will rely on contexts that shall not declare any variable. We call these entities *signatures* and denote them as Σ, possibly subscripted.

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 message typing judgment "$\Gamma \vdash t : \tau$", to be read "*term t has type τ in context* Γ".

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:

$$\frac{\Gamma \vdash t_1 : \mathsf{msg} \quad \Gamma \vdash t_2 : \mathsf{msg}}{\Gamma \vdash t_1\, t_2 : \mathsf{msg}}\,\mathbf{mtp\_cnc}$$

$$\frac{\Gamma \vdash t : \mathsf{msg} \quad \Gamma \vdash k : \mathsf{shK}\, A\, B}{\Gamma \vdash \{t\}_k : \mathsf{msg}}\,\mathbf{mtp\_ske} \qquad \frac{\Gamma \vdash t : \mathsf{msg} \quad \Gamma \vdash k : \mathsf{pubK}\, A}{\Gamma \vdash \{\!\{t\}\!\}_k : \mathsf{msg}}\,\mathbf{mtp\_pke}$$

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 where an object of type msg is expected. The formal rule is as follows:

$$\frac{\Gamma \vdash t : \tau' \quad \tau' :: \tau}{\Gamma \vdash t : \tau}\,\mathbf{mtp\_ss}$$

The last term typing rules deal with elementary message components. An atomic message a has a type τ if the context at hand contains the declaration "$a : \tau$", and similarly for variable declarations. The validity of the type τ in Γ is independently checked by verifying the validity of a context, defined in Section 3.4.

$$\frac{}{(\Gamma, a : \tau, \Gamma') \vdash a : \tau}\,\mathbf{mtp\_a} \qquad \frac{}{(\Gamma, x : \tau, \Gamma') \vdash x : \tau}\,\mathbf{mtp\_x}$$

The well-typedness of a message is a decidable property, assuming that the subsorting relation is acyclic and without infinitely descending chains:

Property 2.1 *If the subsorting relation is a well-order, it is decidable whether the judgment* $\Gamma \vdash t : \tau$ *holds.*

Proof: Observe that the premises of all rules except $\mathbf{tp\_ss}$ invoke the typing judgment on subterms of the message appearing in the rule conclusion, if at all. Since contexts are finite, the unbound meta-variables in the premises can be instantiated only in a finite number of ways. Rule $\mathbf{mtp\_ss}$ does not change the message, but checks it on a subtype. Since the subsorting relation is a well-order, this rule can be applied only a finite number of times before another rule is used to break the message. $\Box$

The dependence of types on terms can make a syntactically correct type meaningless. For example, "$\mathsf{privK}\ k$" is not valid if k has type "$\mathsf{shK}\ A\ B$". We check the validity of a type by means of the judgment "$\Gamma \vdash \tau$", read "*τ is a valid type in context* Γ".

Non-dependent types are valid in every context:

$$\frac{}{\Gamma \vdash \mathsf{princ}}\ \mathbf{ttp\_pr} \qquad \frac{}{\Gamma \vdash \mathsf{nonce}}\ \mathbf{ttp\_nnc} \qquad \frac{}{\Gamma \vdash \mathsf{msg}}\ \mathbf{ttp\_msg}$$

However, whenever a type depends on a term, we must check that the latter is well-formed in the current context. We have the following rules:

$$\frac{\Gamma \vdash A : \mathsf{princ} \quad \Gamma \vdash B : \mathsf{princ}}{\Gamma \vdash \mathsf{stK}\ A\ B}\ \mathbf{ttp\_stK} \qquad \frac{\Gamma \vdash A : \mathsf{princ} \quad \Gamma \vdash B : \mathsf{princ}}{\Gamma \vdash \mathsf{ltK}\ A\ B}\ \mathbf{ttp\_ltK}$$

$$\frac{\Gamma \vdash A : \mathsf{princ}}{\Gamma \vdash \mathsf{pubK}\ A}\ \mathbf{ttp\_pbK} \qquad \frac{\Gamma \vdash k : \mathsf{pubK}\ A}{\Gamma \vdash \mathsf{privK}\ k}\ \mathbf{ttp\_pvK}$$

The absence of a rule validating types of the form "$\mathsf{shK}\ A\ B$" implies that no constant of this type can ever appear in the context or in a declaration: this forces specification writers to use either short-term or long-term keys, rather than generic shared keys. Were we to have types other than the ones considered in this paper, we would need to extend this rule set with additional rules to establish their validity [7].

The decidability of type-checking for terms extends straightforwardly to types. Here and in the sequel, we will assume that subsorting is a well-order.

Property 2.2 *The judgment* $\Gamma \vdash \tau$ *is decidable.*

Proof: The rules for this judgment are syntax-directed *w.r.t.* the structure of types. The premises, when present, invoke the term typing judgment that we know is decidable. □

3 Message Predicates and States

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 formalize the concept of state in *MSR*, together with the constructions needed to implement it.

3.1 Message Tuples and Dependent Types

A *message tuple* $\vec{t}$ is an ordered sequence of terms:

$$\vec{t} \ ::= \ \cdot \ \mid \ t, \vec{t}$$

We will 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 "$\mathsf{princ} \times \mathsf{pubK}\ \mathsf{A}$". 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 "$\mathsf{princ}^{(A)} \times \mathsf{pubK}\ A$", where the variable A in "$\mathsf{princ}^{(A)}$" records the name of the principal at hands and forces the type of the key to be "$\mathsf{pubK}\ A$" for this particular A: here $(\mathsf{A}, \mathsf{k_A})$ is valid, but $(\mathsf{B}, \mathsf{k_A})$ is now ill-typed since $\mathsf{k_A}$ has type "pubK A" rather than the expected "pubK B".[1]

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:

$$\vec{\tau} \ ::= \ \cdot \ | \ \tau^{(x)} \times \vec{\tau}$$

In the second production, the notation $^{(x)}$ on the left of the Cartesian product symbol binds the variable x in the type tuple $\vec{\tau}$ to its right. We write $[t/x]\vec{\tau}$ for the capture-avoiding substitution of t for x in $\vec{\tau}$.

We will drop the label $^{(x)}$ of a type tuple $\tau^{(x)} \times \vec{\tau}$ 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. As for term tuples, we will omit the leading "$\cdot$" whenever convenient.

Type-checking a message tuple with respect to a dependent type tuple reduces to verifying that each component message has the type in the corresponding position. This is formalized by the judgment "$\Gamma \ \vdash \ \vec{t} : \vec{\tau}$", to be read "*term tuple $\vec{t}$ has type $\vec{\tau}$ in* Γ". The rules implementing it are given below: the type of empty message tuple is the empty type tuple, otherwise, components must match after performing the appropriate substitutions to satisfy possible dependencies.

$$\frac{}{\Gamma \ \vdash \ \cdot : \cdot}\ \textbf{mtp\_dot} \qquad\qquad \frac{\Gamma \ \vdash \ t : \tau \quad \Gamma \ \vdash \ \vec{t} : [t/x]\vec{\tau}}{\Gamma \ \vdash \ (t, \vec{t}) : \tau^{(x)} \times \vec{\tau}}\ \textbf{mtp\_ext}$$

These rules implement a decision procedure for the tuple typing judgment, as expressed by the following property:

Property 3.1 *The judgment* $\Sigma \ \vdash \ \vec{t} : \vec{\tau}$ *is decidable.*

Proof: The substitution operation is clearly computable since its definition visits each subexpression in the type tuple being instantiated exactly once (and these objects are finite). The validity of a (finite) term tuple is then decidable based on the analogous property for types. □

We now extend the notion of validity of a type to tuples of types. We rely on the judgment "$\Gamma \ \vdash \ \vec{\tau}$", read "*$\vec{\tau}$ is a valid type tuple in* Γ", to express this relation. The corresponding typing rules simply check that every component of a type tuple is a valid type. Possible dependencies of later types on earlier components are resolved by assuming that the binding variable has the postulated type.

[1] Our dependent type tuples are usually called strong dependent sums and the standard notation for the dependent type tuple we have written as "$\mathsf{princ}^{(A)} \times \mathsf{pubK}\ A$" is "$\Sigma A : \mathsf{princ}.\ \mathsf{pubK}\ A$".

$$\frac{}{\Gamma \vdash \cdot}\ \mathbf{ttp\_dot} \qquad\qquad \frac{\Gamma \vdash \tau \quad \Gamma, x : \tau \vdash \vec{\tau}}{\Gamma \vdash \tau^{(x)} \times \vec{\tau}}\ \mathbf{ttp\_ext}$$

In rule $\mathbf{ttp\_ext}$, we rely on implicit α-conversion to rename the bound variable in case another declaration for it already occurs in Γ.

3.2 Message Predicates

Three kinds of predicates can enter a state or a rewrite rule:

- 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, running protocols rely on a number of *role state predicates* of the form $\mathsf{L}_l(\_, \ldots, \_)$, where l is a unique identifying label. They allow transfering control and data to the next rule to be executed within a role, *i.e.* a collection of rules representing the actions of principal in a protocol. Their arguments record the value of known parameters of the execution of the role up to the current point. As we will see in Section 4.2, rules will mention *role state predicate parameters*, written L, rather than predicate constants L_l.
- 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.

The reader familiar with our previous work on *MSR* will have noticed a number of differences with respect to the definitions given in [9, 10]. 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 [7], 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 [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 will therefore declare them in the context. We can now complete the definition of a context as follows:

$$\Gamma \ ::= \ \cdot \ \mid \ \Gamma, \, a : \tau \ \mid \ \Gamma, \, x : \tau \ \mid \ \Gamma, \, \mathsf{M}\_ : \vec{\tau} \ \mid \ \Gamma, \, \mathsf{L}_l : \vec{\tau} \ \mid \ \Gamma, \, L : \vec{\tau}$$

We write $[t/x]\Gamma$ for the capture-free substitution of term t for variable x in the type of every declaration in context Γ.

Validation of message predicates is expressed by the judgment "$\Gamma \vdash P$", read "P *is a valid message predicate in* Γ". The rules implementing it are given below. The argument of a network predicate must be a valid message. The type of the arguments of role state and memory predicates must correspond to the declared type for this predicate. We regard the subscript A in a memory predicate $\mathsf{M}_A(\vec{t})$ as if it were the first argument for typing purposes. Constant and variable role state predicate are treated identically.

$$\frac{\Gamma \vdash t : \mathsf{msg}}{\Gamma \vdash \mathsf{N}(t)}\,\textbf{ptp\_net} \qquad \frac{(\Gamma, \mathsf{M}\_ : \vec{\tau}, \Gamma') \vdash (A, \vec{t}) : \vec{\tau}}{(\Gamma, \mathsf{M}\_ : \vec{\tau}, \Gamma') \vdash \mathsf{M}_A(\vec{t})}\,\textbf{ptp\_mem}$$

$$\frac{(\Gamma, \mathsf{L}_l : \vec{\tau}, \Gamma') \vdash \vec{t} : \vec{\tau}}{(\Gamma, \mathsf{L}_l : \vec{\tau}, \Gamma') \vdash \mathsf{L}_l(\vec{t})}\,\textbf{ptp\_rsp} \qquad \frac{(\Gamma, L : \vec{\tau}, \Gamma') \vdash \vec{t} : \vec{\tau}}{(\Gamma, L : \vec{\tau}, \Gamma') \vdash L(\vec{t})}\,\textbf{ptp\_L}$$

3.3 States

A *state* is a finite collection of ground message predicates. The syntax of states is formalized by means of the following grammar:

$$S \;::=\; \cdot \;\mid\; S,\ \mathsf{N}(t) \;\mid\; S,\ \mathsf{L}_l(\vec{t}) \;\mid\; S,\ \mathsf{M}_{\mathsf{A}}(\vec{t})$$

As for contexts, we will omit the leading "·" and interpret the extension construct "," as a union operator. It is convenient to abstract from the order of the component predicates of a state, treating them as if they formed a multiset.

The validity of a state reduces to checking that each component predicate is well-typed in the current context. We formalize this by means of the judgment "$\Gamma \vdash S$" read "*S is a valid state in* Γ", which is implemented by the following two rules:

$$\frac{}{\Gamma \vdash \cdot}\,\textbf{stp\_dot} \qquad \frac{\Gamma \vdash S \quad \Gamma \vdash P}{\Gamma \vdash (S, P)}\,\textbf{stp\_ext}$$

We will use these rules to validate both states and rule components. States will always be ground, and therefore checked against a signature rather than a context.

We conclude the presentation of states by showing that validating these objects is decidable.

Property 3.2 *The judgment* $\Gamma \vdash S$ *is decidable.*

Proof: The decidability result for term tuples allows an easy proof of the analogous property for each of our message predicates. Since states are finite sequences of such predicates, it is decidable whether a state is valid. □

3.4 Validating Contexts

Since contexts contain declarations that attribute a type to an elementary message and most predicate symbols, we must concern ourselves with their validity. We express this notion by means of the judgment "$\vdash \Gamma$", to be read "Γ *is a valid context*".

An empty context is trivially valid, a non-empty context $\Gamma, a : \tau$ or $\Gamma, x : \tau$ is valid if the type of its last declaration is valid in Γ, and the tail Γ is itself a valid context.

$$\frac{}{\vdash \cdot}\,\textbf{itp\_dot} \qquad \frac{\Gamma \vdash \tau \quad \vdash \Gamma}{\vdash \Gamma, a : \tau}\,\textbf{itp\_a} \qquad \frac{\Gamma \vdash \tau \quad \vdash \Gamma}{\vdash \Gamma, x : \tau}\,\textbf{itp\_x}$$

Role state and memory predicate declarations are validated similarly. We require that the first argument of a role state and memory predicate be a principal, the role owner.

$$\frac{\Gamma \vdash \mathsf{princ}^{(A)} \times \vec{\tau} \quad \vdash \Gamma}{\vdash \Gamma, \mathsf{L}_l : \mathsf{princ}^{(A)} \times \vec{\tau}}\,\textbf{itp\_rsp} \qquad \frac{\Gamma \vdash \mathsf{princ}^{(A)} \times \vec{\tau} \quad \vdash \Gamma}{\vdash \Gamma, L : \mathsf{princ}^{(A)} \times \vec{\tau}}\,\textbf{itp\_L}$$

$$\frac{\Gamma \vdash \mathsf{princ}^{(A)} \times \vec{\tau} \quad \vdash \Gamma}{\vdash \Gamma, \mathsf{M}_\_ : \mathsf{princ}^{(A)} \times \vec{\tau}}\ \mathbf{itp\_mem}$$

The typing rule for role state predicate variables is constructed as for their non-parametric counterpart.

We are now in a position to verify the decidability of the context validation judgment:

Property 3.3 *The judgment* $\vdash \Gamma$ *is decidable.*

Proof: Since contexts are finite, this result reduces to the decidability of validating types and type tuples, which we have proved in Sections 2.3 and 3.1. □

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.

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 specifications, and provides typing rules for them.

4.1 Rules

With a slight imprecision that will be corrected shortly, 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 that will be instantiated to actual terms during execution. We use typed universal quantifiers to account for this fact. Therefore, we have the following grammar for rules:

$$r \quad ::= \quad lhs \to rhs \quad | \quad \forall x : \tau.\, r$$

In earlier versions of *MSR* [9, 10], parameters were implicitly universally quantified and therefore untyped.

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*:

$$lhs \ ::= \ \cdot \ \mid \ lhs,\ \mathsf{N}(t) \ \mid \ lhs,\ L(\vec{e}) \ \mid \ lhs,\ \mathsf{M}_A(\vec{t})$$

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 (as enforced by the typing rules below). 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:

$$rhs \ ::= \ lhs \ \mid \ \exists x : \tau.\, rhs$$

Later rules can refer to the values created by existential quantifiers by introducing universal quantifications of the proper type: synchronization is ensured by the occurrence of these values in the role state predicates. We write $[t/x]rhs$ for the capture-free substitution of a term t for a variable x in the consequent rhs. We adopt a similar notation for rules and predicate sequences.

We can now present the typing rules for rules and their components. We shall first turn our attention to rule consequents, to which we attribute the typing judgment "$\Gamma \ \overset{r}{\Vdash} \ rhs$", to be read "*$rhs$ is a valid rule consequent in* Γ" (the superscript "r" is intended to distinguish this judgment from the analogous relation for states). We have the following typing rules:

$$\frac{\Gamma \vdash \tau \quad (\Gamma, x : \tau) \overset{r}{\Vdash} rhs}{\Gamma \overset{r}{\Vdash} \exists x : \tau.\, rhs}\ \mathbf{rtp\_nnc} \qquad\qquad \frac{\Gamma \vdash lhs}{\Gamma \overset{r}{\Vdash} lhs}\ \mathbf{rtp\_seq}$$

Rule $\mathbf{rtp\_nnc}$ validates consequents that start with a fresh datum declaration. The left premise verifies that the type τ of the variable x is valid. This is necessary since otherwise invalid types may not be caught. The right premise extends the current typing context with the declaration for x and attempts to validate the rest of the consequent. The addition of $x : \tau$ to the context will allow using rule $\mathbf{mtp\_x}$ (see Section 2.3) to check every occurrence of x in rhs. We required that no variable in a typing context or signature be declared more than once. Implicit α-conversion provides a simple way to implement this constraint: if a variable named x is already contained in Γ, we implicitly choose an unused symbol, use it to rename every occurrence of x in $\exists x : \tau.\, rhs$, and seamlessly apply rule $\mathbf{rtp\_nnc}$ to this expression.

Once all existential quantifiers have been stripped from a rule consequent, rule $\mathbf{rtp\_seq}$ invokes the typing judgment for (parametric) states to validate the exposed predicate sequence. This is sufficient since, as we observed, a predicate sequence is a parametric state of a particular form.

Given the above typing rules, the judgment "$\Gamma \vdash r$", to be read "*r is a valid rule in* Γ", validates rules themselves. It is implemented by the following two rules:

$$\frac{\Gamma \vdash lhs \quad \Gamma \Vdash^{\Sigma} rhs}{\Gamma \vdash lhs \to rhs}\,\mathbf{utp\_core} \qquad \frac{\Gamma \vdash \tau \quad (\Gamma, x : \tau) \vdash \rho}{\Gamma \vdash \forall x : \tau.\, \rho}\,\mathbf{utp\_all}$$

Rule $\mathbf{utp\_core}$ reduces rule validation to type-checking its antecedent as a parametric state and verifying its right-hand side as described above. The quantifier in rule $\mathbf{utp\_all}$ is treated as in the case of $\mathbf{rtp\_nnc}$.

At this point, we can show that it can be decided whether a rule is well typed. We have the following proposition:

Property 4.1 *The judgment* $\Gamma \vdash r$ *is decidable.*

Proof: Type-checking a rule reduces to the validation of parametric states and types after a finite number of applications of the typing rules seen in this section. Since the analogous judgments for states and types are decidable and the choice of which rule to apply is syntax-directed, type-checking is decidable for rules as well. $\square$

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 $\vec{\tau}$. With this insight, the following grammar defines the notion of *rule collection*:

$$\rho \ ::= \ \cdot \ \mid \ \exists L : \vec{\tau}.\, \rho \ \mid \ r,\, \rho$$

We write $[\mathsf{L}_l/L]\rho$ for the capture-free substitution of the constant predicate symbol L_l for the role state predicate parameter L in ρ.

The judgment "$\Gamma \vdash \rho$", read *ρ is a valid rule collection in* Γ, has the function of validating a rule collection. The following three rules realize it:

$$\frac{}{\Gamma \vdash \cdot}\,\mathbf{otp\_dot} \qquad \frac{\Gamma \vdash \vec{\tau} \quad (\Gamma, L : \vec{\tau}) \vdash \rho}{\Gamma \vdash \exists L : \vec{\tau}.\, \rho}\,\mathbf{otp\_rsp} \qquad \frac{\Gamma \vdash r \quad \Gamma \vdash \rho}{\Gamma \vdash r, \rho}\,\mathbf{otp\_rule}$$

The quantifier in rule $\mathbf{otp\_rsp}$ is treated as in the case of $\mathbf{rtp\_nnc}$: the left premise validates the type tuple $\vec{\tau}$ in the current context Γ, while the other premise analyzes ρ after introducing the role state predicate parameter L in Γ. Rule $\mathbf{otp\_rule}$ applies when a rule collection starts with a rule r.

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 ρ^{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. These *generic roles* are denoted $\rho^{\forall A}$, 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.

4.3 Protocol Theories

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

$$\mathcal{P} \;::=\; \cdot \;\mid\; \mathcal{P},\ \rho^{\forall A} \;\mid\; \mathcal{P},\ \rho^{A}$$

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. How this is achieved for the standard Dolev-Yao intruder is illustrated in [6].

The validity of a protocol theory is always checked against a signature. Therefore, roles containing free variables will not type-check. The judgment "$\Sigma \vdash \mathcal{P}$", read "$\mathcal{P}$ *is a valid protocol theory in signature* Σ", expresses this relation. The rules implementing it are given below. They essentially reduce the validity of a protocol theory to the validation of the rule collections within roles.

$$\frac{}{\Sigma \vdash \cdot}\ \mathbf{htp\_dot} \qquad\qquad \frac{\Sigma \vdash \mathcal{P} \quad (\Sigma, A : \mathsf{princ}) \vdash \rho}{\Sigma \vdash \mathcal{P}, \rho^{\forall A}}\ \mathbf{htp\_grole}$$

$$\frac{(\Sigma, \mathsf{A} : \mathsf{princ}, \Sigma') \vdash \mathcal{P} \quad (\Sigma, \mathsf{A} : \mathsf{princ}, \Sigma') \vdash \rho}{(\Sigma, \mathsf{A} : \mathsf{princ}, \Sigma') \vdash \mathcal{P}, \rho^{\mathsf{A}}}\ \mathbf{htp\_arole}$$

Observe the difference in the way the signature is used in rule **htp_grole** and **htp_arole**. In the former, A is a variable, and therefore, the assumption "$A : \mathsf{princ}$" is added to the signature Σ, with the effect of invoking the role validation judgment with a typing context. The latter rule instead checks whether the signature at hands has a record that the constant A has type princ: the role validation judgment is called with a signature.

We conclude this section by stating and proving decidability results for type-checking roles and protocol theories.

Property 4.2 *The judgments* $\Gamma \vdash \rho$ *and* $\Sigma \vdash \mathcal{P}$ *are decidable.*

Proof: A role is a finite collection of rules with a distinguished constant or variable (the role owner). Type-checking a role reduces then to validating its rules, which is decidable. The second judgment reduces to the first since a protocol theory consists of finitely many roles. $\square$

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:

$$R \ ::= \ \cdot \ \mid \ R,\ \rho^{\mathsf{A}}$$

The notation ρ^{A} is reminescent 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.

Type-checking active role sets is achieved by means of the judgment "$\Sigma \vdash R$", read "*R is a valid active role set in Σ*", and implemented by the following two typing rules:

$$\frac{}{\Sigma \ \vdash \ \cdot}\ \mathbf{atp\_dot} \qquad\qquad \frac{(\Sigma, \mathsf{A} : \mathsf{princ}, \Sigma') \ \vdash \ R \quad (\Sigma, \mathsf{A} : \mathsf{princ}, \Sigma') \ \vdash \ \rho}{(\Sigma, \mathsf{A} : \mathsf{princ}, \Sigma') \ \vdash \ R, \rho^{\mathsf{A}}}\ \mathbf{atp\_ext}$$

Active role sets are type-checked with respect to signatures, and therefore they shall not contain free variables. In rule $\mathbf{atp\_ext}$, we first verify that the role owner A is an actual principal declared in the signature. Then we check in the right-hand premise that ρ is a valid rule collection.

Verifying the validity of an active role set is decidable, as stated by the following property:

Property 4.3 *The judgment* $\Sigma \ \vdash \ R$ *is decidable.*

Proof: Validating an active role set ultimately reduces to a finite number of uses of the type-checking judgment for rules, which we have proved decidable. □

5 Protocol Execution

The typing policy defined in the previous sections provide ways of statically checking that the *MSR* specification of a protocol meets well-understood default notions of adequacy. More complex analyses require a description of how a protocol evolves at run time. In this section, we will define such an execution model for *MSR*, and study how it interacts with typing.

5.1 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 introduce judgments and rules describing the atomic execution steps that lead from S to S'. We 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 $[S]_{\Sigma}^{R}$ consisting of a state S, an active role set R and a signature Σ. Such triples are called *snapshots*, and denoted with the letter C. 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 before stripping their binders.

Given a protocol $\mathcal{P}$, we describe the fact that execution transforms a snapshot C into another snapshot C' by means of the *one-step sequential firing* judgment "$\mathcal{P} \triangleright C \longrightarrow C'$". It 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.

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:

$$\frac{}{(\mathcal{P}, \rho^{\mathsf{A}}) \triangleright [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,\rho^{\mathsf{A}}}}\;\textbf{ex\_arole} \qquad \frac{\Sigma \vdash \mathsf{A} : \mathsf{princ}}{(\mathcal{P}, \rho^{\forall A}) \triangleright [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,([\mathsf{A}/A]\rho)^{\mathsf{A}}}}\;\textbf{ex\_grole}$$

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 **ex_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.

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 $(\forall L : \vec{\tau}.\, \rho)^{\mathsf{A}}$, 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 (*i.e.* it should not occur in Σ).

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

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.

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

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:

$$\frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(r,\rho)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R,(\rho)^{\mathsf{A}}}}\ \mathbf{ex\_skp} \qquad \frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(\cdot)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R}}\ \mathbf{ex\_dot}$$

The inference on the left skips a protocol rule. Rule $\mathbf{ex\_dot}$ does some housekeeping 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' in the extended signature Σ'. This operation is defined by the *rule application* judgment "$r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}$". 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:

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

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 "$(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}$" discussed below.

$$\frac{(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}}{(lhs \rightarrow rhs) \triangleright [S, lhs]_{\Sigma} \gg [S, lhs']_{\Sigma'}}\ \mathbf{ex\_core}$$

The *right-hand side instantiation* judgment used in the premise of rule $\mathbf{ex\_core}$ 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 written "$(rhs)_{\Sigma} \gg (lhs)_{\Sigma'}$". 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:

$$\frac{}{(lhs)_{\Sigma} \gg (lhs)_{\Sigma}}\ \mathbf{ex\_seq}$$

The instantiation of an existentially quantified term variable x is handled in rule $\mathbf{ex\_nnc}$ 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.

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

Observe that, in this rule, the generated constant 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 that allows us to chain the atomic steps presented above into multi-step sequential firings. This permits simulating executions consisting of any number of basic steps between two snapshots of interest. The *multi-step sequential firing* judgment "$\mathcal{P} \triangleright C \longrightarrow^{*} C'$" is defined as the reflexive and transitive closure of the atomic step relation discussed above. We omit its simple rules for space reasons. We will write "$\mathcal{P} \triangleright C \longrightarrow^{(*)} C'$" to indicate that we are considering the one-step and the multi-step versions of the sequential firing judgment at once.

5.2 Type Preservation

This section is dedicated to exploring the relationship between the simulation semantics outlined in Section 5.1 and the constraints on sensible objects provided by the typing rules in Sections 2–4. Our main result is the Type Preservation Theorem 5.2, which certifies that execution preserves typability: when starting with well-typed objects, firing will always produce well-typed entities. This property is an important sanity check concerning the interaction of the typing and execution rules in our system. On the other hand, it has useful implications for actual implementations of *MSR*.

The proof of the type preservation theorem makes use of a number of lemmas. Space constraints force us to omit the statement of most of these results. We will however mention them in our proof sketches. The main auxiliary result in the proof of the type preservation theorem is the following *Term Substitution Lemma*, a fairly general result about the effect of substitutions on valid typing judgments. It states that whenever a judgment mentions a variable x of type τ, replacing every occurrence of x with a term of the same type maintains typability, assuming that some simple preconditions are met. The exact text of this lemma is as follows:

Lemma 5.1 *Let Σ and Σ' be signatures, t a term and τ a type such that "$\Sigma, \Sigma' \vdash t : \tau$" is derivable. Let moreover x be a variable, and Γ a context such that "$\vdash \Sigma, x : \tau, \Gamma$".*
If $\Sigma, x{:}\tau, \Gamma \vdash X{:}Y$, then $\Sigma, \Sigma', [t/x]\Gamma \vdash [t/x]X : [t/x]Y$
where "$\Gamma' \vdash X : Y$" is one of the typing judgment in this paper that accepts a context. (Y is defined only for judgments with two objects on the right of the turnstyle.)

Proof: The proof of this lemma proceeds by induction on a derivation of the judgment "$\Sigma, x : \tau, \Gamma \vdash X : Y$". It relies on a number of omitted auxiliary results, namely a *Weakening Lemma* that allows enlarging the context of a derivable judgement, a *Subtyping Lemma* that analyzes the interaction of substitutions and subtyping, a *Variable Scoping Lemma* that investigates how substitutions actually affects a context, and a technical *Compound Substitutions Lemma* that states relevant properties of composed substitutions. Aspects of this proof concern specific constructs in our term language. Therefore, whenever extending this language for a particular application, we shall adapt the proof to the new syntax and inference rules. □

For space reasons, we omit the analogous *Role State Predicate Constant Substitution Lemma* for predicate parameters.

The Type Preservation Theorem asserts that if we start from a snapshot in whose signature the protocol theory, the initial state and the initial active role set are typable, then the application of an execution sequence will yield a snapshot where all the corresponding objects are typable.

Theorem 5.2 *Let $\mathcal{P}$ be a protocol theory, Σ and Σ' signatures, R and R' active role sets, and S and S' states such that the judgments "$\vdash\Sigma$", "$\Sigma\vdash\mathcal{P}$", "$\Sigma\vdash R$" and "$\Sigma\vdash S$" hold. If*

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

then the following judgments are derivable

$$\vdash \Sigma' \qquad \Sigma' \vdash \mathcal{P} \qquad \Sigma' \vdash R' \qquad \Sigma' \vdash S'.$$

Proof: The proof proceeds by induction on the a derivation of the given execution judgment. It relies on the Term Substitution Lemma 5.1 to handle substitutions, and on a number of omitted auxiliary results. In particular, we have lemmas with a text similar to this theorem for the rule application and the right-hand side instantiation judgments. The former makes use of a *State Merge/Join Lemma* to process rule $\mathbf{ex\_core}$. □

In many languages, the validity of a type preservation theorem has one important implication: types are not needed at run time. This allows for a convenient and efficient separation of phases: in a first, static, phase, the specification at hand is checked for type consistency. In a second, dynamic, phase, execution is emulated without any reference to types. Therefore, the data structures used at run-time do not need to record and maintain typing information, with significant speed advantages.

These benefits of type preservation do not apply in full in the case of *MSR*. Indeed, rules $\mathbf{ex\_grole}$ and $\mathbf{ex\_all}$ directly invoke the typing judgment to instantiate variables with terms of the proper type. We are currently investigating operational improvements to these rules and expect that in many situations type correctness is guaranteed. However, there are circumstances where omitting a run-time type-check would result in violations of the Type Preservation Theorem.

6 Examples

We will now exemplify the expressive power of *MSR* by formalizing the Otway-Rees authentication protocol [18]. This protocol is interesting for several reasons: it relies on a server to generate a shared key between two principals, it features both short-term and long-term keys, and our formalization is immune from the "type flaw" attack generally associated with it [5, 11] (we are currently investigating ways to make such attacks expressible in *MSR*). In [7], we have given *MSR* formalizations of several other protocols including the full version of the Needham-Schroeder public-key authentication protocol [11] and the Neuman-Stubblebine repeated authentication protocol [11], which makes an essential use of memory predicates. We are currently using *MSR* to give a formal specification of the *OFT* multicast key management protocol suite [4].

The Otway-Rees authentication protocol [18, 5] relies on a server S to mutually authenticate two generic parties A and B, and establish a common short-term key k_{AB}. This protocol is written as follows in the "usual notation":

1. $A \to B$: $n\ A\ B\ \{n_A\ n\ A\ B\}_{k_{A\mathsf{S}}}$
2. $B \to \mathsf{S}$: $n\ A\ B\ \{n_A\ n\ A\ B\}_{k_{A\mathsf{S}}}\ \{n_B\ n\ A\ B\}_{k_{B\mathsf{S}}}$
3. $\mathsf{S} \to B$: $n\ \{n_A\ k_{AB}\}_{k_{A\mathsf{S}}}\ \{n_B\ k_{AB}\}_{k_{B\mathsf{S}}}$
4. $B \to A$: $n\ \{n_A\ k_{AB}\}_{k_{A\mathsf{S}}}$

Throughout this protocol, n is a nonce intended as a run identifier; it is generated by the initiator A. In line (1), A sends the displayed message to the responder B. The nonce n_A in the encrypted component is intended for authentication; here $k_{A\mathsf{S}}$ is the long-term key A shares with the server S. Upon receiving this message in line (2), B forwards it to S together with a similarly encrypted component: n_B is his own authentication nonce, and $k_{B\mathsf{S}}$ is the long-term key he shares with S. The server decrypts the encoded components and checks that they agree on A, B, and n. Then, in line (3), it generates a short-term key k_{AB} intended for communication between A and B, and sends the shown three-part message to B, which forwards the first two components to A in line (4). Authentication is achieved, and k_{AB} is accepted, if A and B receive the nonces n_A and n_B they had generated in the first part of the protocol.

This protocol has been shown to be flawed [11]. Of particular interest here is the type flaw attack obtained by having an intruder replay A's initial message back to her: if she mistakes the triple $(n\,A\,B)$ as the key k_{AB} on line (4), she will not only erroneously believe to have authenticated B, but also use a "key" known to the intruder to encrypt subsequent messages. Most specifications of this protocol, in particular those founded on the Dolev-Yao model, avoid this attack by assuming that there is enough redundancy in real messages to differentiate a key from a concatenation. Our strongly-typed *MSR* formalization clearly follows this assumption. It should however be noticed that we could have set up our type system so that arbitrary messages can be used as keys, which would have enabled expressing this attack.

We will now express each role in turn in the syntax of *MSR*. For space reasons, we will typeset homogeneous constituents, namely the universal variable declarations and the predicate sequences in the antecedent and consequent, in columns within each rule; we will also rely on some minor abbreviations.

The initiator's actions are represented by the following two-rule generic role.

$$
\left(
\begin{array}{lll}
\multicolumn{3}{l}{\quad \exists L : \mathsf{princ}^{(A)} \times \mathsf{princ} \times \mathsf{nonce} \times \mathsf{nonce}.} \\[1ex]
\begin{array}{l} \forall B : \mathsf{princ}. \\ \forall k_{A\mathsf{S}} : \mathsf{ltK}\ A\ \mathsf{S}. \end{array} & \cdot & \rightarrow \begin{array}{l} \exists n, n_A : \mathsf{nonce}. \\ \mathsf{N}(n\ A\ B\ \{n_A\ n\ A\ B\}_{k_{A\mathsf{S}}}) \\ L(A, B, n, n_A) \end{array} \\[1ex]
\begin{array}{l} \forall \ldots \\ \forall n, n_A : \mathsf{nonce}. \\ \forall k_{AB} : \mathsf{stK}\ A\ B. \end{array} & \begin{array}{l} \mathsf{N}(n\ \{n_A\ k_{AB}\}_{k_{A\mathsf{S}}}) \\ L(A, B, n, n_A) \end{array} & \rightarrow \ \cdot
\end{array}
\right)^{\forall A}
$$

The first rule corresponds to A's view of line (1) above. She generates the nonces n and n_A and transmits the first message (intended to B). She memorizes information needed in order to interpret subsequent messages addressed to her in the role state predicate L. This information is used in the second rule, which captures line (4) from A's point of view. Notice that the presence of L in both rules forces the type of its arguments. We take advantage of this fact to save some space by keeping some of the universal quantifiers implicit as "$\forall \ldots$" in the second rule. Observe how dependent types in the declaration for L enforce the association between its first and last arguments.

The responder's actions are expressed in the following generic role. The first rule corresponds to lines (1) and (2) in the "usual notation" and the second to lines (3) and (4).

$$
\left(
\begin{array}{lll}
\multicolumn{3}{l}{\quad \exists L : \mathsf{princ}^{(B)} \times \mathsf{princ} \times \mathsf{nonce} \times \mathsf{nonce} \times \mathsf{ltK}\ B\ \mathsf{S}.} \\[1ex]
\begin{array}{l} \forall A : \mathsf{princ}. \\ \forall n : \mathsf{nonce}. \\ \forall k_{B\mathsf{S}} : \mathsf{ltK}\ B\ \mathsf{S}. \\ \forall X : \mathsf{msg}. \end{array} & \mathsf{N}(n\ A\ B\ X) & \rightarrow \begin{array}{l} \exists n_B : \mathsf{nonce}. \\ \mathsf{N}(n\ A\ B\ X\ \{n_B\ n\ A\ B\}_{k_{B\mathsf{S}}}) \\ L(B, A, n, n_B, k_{B\mathsf{S}}) \end{array} \\[1ex]
\begin{array}{l} \forall \ldots \\ \forall n_A : \mathsf{nonce}. \\ \forall k_{AB} : \mathsf{stK}\ A\ B. \\ \forall Y : \mathsf{msg}. \end{array} & \begin{array}{l} \mathsf{N}(n\ Y\ \{n_B\ k_{AB}\}_{k_{B\mathsf{S}}}) \\ L(B, A, n, n_B, k_{B\mathsf{S}}) \end{array} & \rightarrow\ \mathsf{N}(n\ Y)
\end{array}
\right)^{\forall B}
$$

The responder is not entitled to observe the inner structure of the submessages $\{n_A n A B\}_{k_{A\mathsf{S}}}$ and $\{n_A\ k_{AB}\}_{k_{A\mathsf{S}}}$ in lines (1–2) and (3–4). We express this fact by using the variables X and Y, both of the generic type msg in this role. This allows B to forward this information, but not to inspect it.

The last role in this protocol encompasses the actions of the server. Assuming that there is a single server, S, they can conveniently be expressed by the following anchored role, which consists of a single rule. These actions correspond to lines (2) and (3) of the "usual notation" specification.

$$
\left(
\begin{array}{lll}
\begin{array}{l} \forall A, B : \mathsf{princ}. \\ \forall k_{A\mathsf{S}} : \mathsf{ltK}\ A\ \mathsf{S}. \\ \forall k_{B\mathsf{S}} : \mathsf{ltK}\ B\ \mathsf{S}. \end{array} & \begin{array}{l} \mathsf{N}(n\ A\ B\ \{n_A\ n\ A\ B\}_{k_{A\mathsf{S}}} \\ \quad \{n_B\ n\ A\ B\}_{k_{B\mathsf{S}}}) \end{array} & \rightarrow \begin{array}{l} \exists k_{AB} : \mathsf{stK}\ A\ B. \\ \mathsf{N}(n\ \{n_A\ k_{AB}\}_{k_{A\mathsf{S}}}\ \{n_B\ k_{AB}\}_{k_{B\mathsf{S}}}) \end{array}
\end{array}
\right)^{\mathsf{S}}
$$

Upon receiving the message in line (2), the server constructs a short-term key k_{AB} for principals A and B. It should be observed how key generation is specified as existential quantification. The use of dependent types makes this process particularly elegant. Observe also that the distinction between short-term and long-term keys and our typing rules prevent mistakenly transmitting $k_{A\mathsf{S}}$ instead of k_{AB}, for example.

Our notation provides a specific type for each variable appearing in the above rules. The equivalent "usual notation" specification relies instead on natural language and conventions to convey this same information, with clear potential for ambiguity: its succinctness is only apparent. On the other hand, most type declarations in the *MSR* specification can however be automatically reconstructed, as discussed in [7]. This simplifies the task of the author of a specification by enabling him or her to concentrate on the message flow rather than on typing details, and of course it limits the size of the specification. Algorithmic rules for this form of type reconstruction are the subject of current research.

7 Conclusions and Future Work

In this paper, we have presented the typing infrastructure of *MSR* [7, 6, 9, 10, 8], a strongly typed specification language for security protocols. Typing contributes to writing elegant and precise formalizations. It enables static checks aimed at discovering simple but potentially harmful specification mistakes, *e.g.* the unduly transmission of a long-term key. It also underlies static data access verification [6], which catches, for example, such errors as a principal encrypting a message with a key he/she has no access to, or look up private information that a principal holds private.

Future work includes extending our current collection of case studies to encompass not only the most common authentication protocols [11], but also complex schemes such as key management protocols for group multicast [4] and fair exchange protocols. On the terrain of usability, we are working on the design of a reliable type reconstruction algorithm in order to shelter users from the often tedious process of providing complete type declarations, and to make formalization reasonably sized. We are also developing an operational execution model that delays instantiations until information is available to guide the choice of the instantiating terms. Moreover, we are extending *MSR* to cope with protocols that rely on objects belonging to complex interpretation domains such as timestamps [7]. Finally, we are investigating the possibility of reconstructing the rules that describe the Dolev-Yao intruder from a typed specification of a protocol. On a related topic, we are adapting the type system of *MSR* to permit precise descriptions of type flaw attacks.

References

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

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. Research Report 125, Digital Equipment Corp., System Research Center, 1994.

[3] D. Aspinall and A. 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.

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

[5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society, Series A*, 426(1871):233–271, 1989.

[6] I. Cervesato. Typed multiset rewriting specification of security protocols. Unpublished manuscript.

[7] I. 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*, pages 159–177, St. Petersburg, Russia, 21–23 May 2001. Springer-Verlag LNCS 2052.

[8] I. Cervesato, N. A. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in linear logic. In *2000 Workshop on Formal Methods and Computer Security — FMCS'00*, Chigaco, IL, July 2000.

[9] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. 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.

[10] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. 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.

[11] J. Clark and J. 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/`.

[12] G. Denker and J. 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.

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

[14] N. Durgin, P. Lincoln, J. Mitchell, and A. 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.

[15] F. J. T. Fábrega, J. C. Herzog, and J. 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.

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

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

[18] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, Jan. 1987.

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