11-2002

Data Access Specification and the Most Powerful Symbolic Attacker in MSR

Iliano Cervesato
ITT Industries

Data Access Specification and the Most Powerful Symbolic Attacker in *MSR* [*]

Iliano Cervesato

ITT Industries, Inc. — Advanced Engineering and Sciences Division
2560 Huntington Avenue, Alexandria, VA 22303-1410 — USA
Tel.: +1-202-404-4909, fax: +1-202-404-1167
iliano@itd.nrl.navy.mil

Abstract

Most systems designed for the symbolic verification of security protocols operate under the unproved assumption that an attack can only result from the combination of a fixed number of message transformations, which altogether constitute the capabilities of the so-called Dolev-Yao intruder. In this paper, we show that the Dolev-Yao intruder can indeed emulate the actions of an arbitrary symbolic adversary. In order to do so, we extend MSR, a flexible specification framework for security protocols based on typed multiset rewriting, with a static check called data access specification and aimed at catching specification errors such as a principal trying to use a key that she is not entitled to access.

1 Introduction

Cryptographic protocols are increasingly 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, and their deployment in distributed environments plagued by powerful and opportunistic attackers. Most systems designed for protocol analysis, *e.g.* [1, 5, 14, 16, 20] circumvent the first issue by relying on a symbolic idealization known as the Dolev-Yao model of security [15, 21]: 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 bit-strings. Systems that adopt the Dolev-Yao idealization achieve relative tractability by empowering their attacker model with a fixed set of basic capabilities, altogether known as the Dolev-Yao intruder. It is a commonly held, but unproved, belief that this model is sufficient to expose any attack that can be mounted by a symbolic adversary, *i.e.* one playing by the rules of the Dolev-Yao abstraction, but otherwise arbitrary.

MSR originated as a simple logic-oriented language aimed at investigating the decidability of protocol analysis within the Dolev-Yao model [11]. 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 [8, 9, 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 [8]. An earlier version of *MSR* fuels the *CAPSL* authentication protocol verification tool in the form of the underlying *CIL* intermediate language [14].

In this paper, we use *MSR* as a formal tool to study the very nature of the Dolev-Yao intruder, and ultimately to show that it subsumes any other attacker that follows the Dolev-Yao abstraction. In order to do so, we endow *MSR* with a novel static check, data access specification (DAS), aimed at enforcing such sensible requirements as, for example, that a principal may access the public key of any other principal, but in general not their private keys. The natural *MSR* specification of the Dolev-Yao intruder is shown not only to satisfy the data access policy, but in doing so to make use of every facet of it. This highlights subtle constraints on the Dolev-Yao intruder model that are typically not exposed by other approaches. Since, differently from most proposals, *MSR* allows specifying an arbitrary attacker using the same syntax as a regular protocol, we can provide an effective construction that shows that our formalization of the Dolev-Yao intruder can emulate the deeds of every adversary that satisfies the typing and data access policies of *MSR*.

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

The available tools designed to offer automated support to security protocol verification [4, 14, 18, 20, 22, 24, 25] provide a limited array of static checks, often just type-checking for simple types. In particular, we are not aware of any proposal that enforces DAS. Moreover, while nearly all approaches rely on some variant of the Dolev-Yao intruder, we could not find a formal proof in the literature that this model indeed implements the most powerful symbolic attacker. Finally, it should be observed that our notion of data access specification is orthogonal to the insightful guidelines of [2, 27], aimed at constructing protocols that are immune by design to certain classes of attack.

The main contributions of this paper are: 1) the definition of a decidable notion of DAS for *MSR* that is preserved by execution; 2) the formal definition of the concept of a generic symbolic attacker, based on DAS, and the formalization of the Dolev-Yao intruder; 3) a proof highlight that the Dolev-Yao intruder model can emulate an arbitrary attacker that satisfies the data access policy (the full proof can be found in [6]). This paper is about the specification of cryptographic protocols, not about protocol analysis.

This paper is structured as follows: in Section 2, we carefully recall the form of an *MSR* specification. Its DAS policy, a major novelty of this paper, and the relative decidability results are the subject of Section 3. In Section 4, we present the execution model of our framework and prove that it preserves access control. In Section 5, we formalize the Dolev-Yao intruder in *MSR* and in Section 6 we prove that it can emulate the actions of an arbitrary attacker. Section 7 hints at directions of future work.

2 MSR

In the past, cryptoprotocols 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. A role is given as a parameterized collection of multiset rewrite rules that encode the expected message receptions and the corresponding transmission. An illustrative example can be found in Appendix A. Rule firing emulates receiving (and accepting) a message and/or sending a message. The messages in transit, the actions and information available to the roles, and other data constitute the state of execution of a protocol. Rules implement partial transformations between states. Their applicability is constrained by the contents of the current state. Execution is preceded by static type-checking [8] and DAS validation (see Section 3) which limits the number of run-time checks and allows catching common specification errors early.

2.1 Messages

Messages, or more properly *terms*, 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):

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

Although we will limit our discussion to these kinds of atomic messages, it should be noted that others can be accommodated by extending the appropriate definitions [9]. The *message constructors* we will consider are concatenation, shared-key encryption, and public-key encryption:

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

Observe that we use a different syntax for shared-key and public-key encryption. We could have identified them, as done in many approaches. We choose instead to distinguish them to show the flexibility and precision of our technique. Again, other constructors, for example hash functions and digital signatures [9], can easily be accommodated by extending the appropriate definitions. Their inclusion would lengthen the discussion without introducing substantially new concepts.

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 variable. Constants and variables constitute the class of *elementary terms*, denoted with the letter e. Finally, we write $[t/x]t'$ for the substitution of a variable x with a term t in another term t' [7].

2.2 Message Predicates and States

States are a fundamental concept in *MSR*: they are the central constituent of the configurations 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. A state is a finite collection of atomic first-order formulas called *message predicates i.e.* predicate symbols applied to an ordered sequence of terms:

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

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, roles rely on a number of *role state predicates*, generally one for each of their rules, 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. Memory predicates are useful in modeling situations that need to maintain data private across role executions: for example, they allow a principal to remember his Kerberos ticket [9], or the trusted-third-party of a fair exchange protocol to avoid fraudulent recoveries from aborted transactions. They are also used to encapsulate such entities as local clocks [9]. Finally, they allow an intruder to accumulate knowledge to be used in mounting attacks, as described in Section 5.

A *state* is a finite collection of ground message predicates, as formalized in the following grammar:

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

We interpret the extension construct "," as a multiset union operator, abstracting in this way from the order of the component predicates of a state.

Protocol rules transform states by removing a number of component predicates, and adding other, usually related, state elements. The antecedent and consequent of a rewrite rule embed therefore parametric substates whose variables are instantiated at application time. However, role state predicates need to be created on the spot in order to avoid interferences between concurrently executing role instances. We achieve this by introducing variables, denoted L, that are instantiated to actual role state predicates during execution.

2.3 Types

Typing is available in *MSR*, as in many languages, as a mechanism for abstracting away aspects of a specification considered too low-level. In the case of a protocol, we are abstracting the method by which a principal distinguishes objects of a different nature and categorizes them: for example field lengths, redundancy, database accesses, trusted subprotocols, etc. *MSR* offers simple yet powerful means to express this abstraction. However, it is ultimately the specifier who is in charge of laying out a sensible set of types for a protocol, and in particular to avoid the danger of over-abstraction.

The typing machinery of *MSR* [8] is based on the type-theoretic notion of *dependent product types with subsorting* [12, 17, 3, 23]. In this paper, we use the following types to classify terms:

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

The types “principal” and “nonce” 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. A key “k” shared between principals “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”.

We use the type msg to classify generic messages. Clearly raw data have type msg. This is however not sufficient since nonces, keys, and principal identifiers are routinely part of messages. We address this issue by imposing a *subsorting* relation between types, formalized by the judgment “$\tau :: \tau'$” (read τ *is a subsort of* τ'). In this paper, the subsorting relation will amount to having each of the types discussed above be a subtype of msg. Its extension can be found in Appendix B.

Again, the above types 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” [9]. An untyped setting is obtained by ascribing type msg to every entity. On the other hand, other applications may find convenient 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 [8].

We use dependent Cartesian product types, here called *dependent type tuples*, to classify term tuples and consequently predicate symbols. These objects are defined as follows:

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

The notation $^{(x)}$ on the left of the Cartesian product symbol binds the variable x in the type tuple $\vec{\tau}$ to its right. Dependencies allow capturing fine associations between arguments, such as between a principal and his/her own public key: for example the type tuple “$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$” will only classify pairs (A, k) where k is the public key of A. 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 $\vec{\tau}$. Examples of dependent tuples can be found in Appendix A.

Ground objects are type-checked against a *signature*, defined as a list of type declarations for atomic constants and memory and role state predicate symbols (the network predicate is hardwired in *MSR*):

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

Objects containing variables rely instead on a *typing context*, defined as a signature extended with declarations for variables and role state predicate parameters:

$$\begin{array}{llcll}
\textit{Typing context} & \Gamma & ::= & \Sigma & \textit{(Plain signature)} \\
 & & \mid & \Gamma, x : \tau & \textit{(Variable declaration)} \\
 & & \mid & \Gamma, L : \vec{\tau} & \textit{(Role state predicate parameter declaration)}
\end{array}$$

We assume that each object in a signature (or context) is declared exactly once. We promote “,” to denote union. This operation is defined only if the resulting sequence does not contain multiple declaration for the same object.

The typing judgments and rules of *MSR* are thoroughly analyzed in [8] and summarized in Appendix B. In the sequel, we will make use of the typing judgments for terms “$\Sigma \vdash t : \tau$”, signatures “$\vdash \Sigma$” and states “$\Sigma \vdash S$”. Binary judgments “$\Sigma \vdash X$” will also be used for entities X still to be introduced, in particular rules, protocol theories, and active role sets. Type-checking *MSR* specifications has been proved decidable in [8].

2.4 Rules

Rules are the basic mechanism that enables the transformation of one state into another, and therefore the simulation of protocol execution: whenever the antecedent matches part of the current state, this portion may be substituted with the consequent (after some processing). Protocol rules are generally parametric so that the same rule can be used in a number of slightly different scenarios (*e.g.* without fixing interlocutors or nonces). Therefore a typical rule mentions variables that are instantiated to actual terms during execution. We introduce them by means of typed universal quantifiers:

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

Free variables can occur in the construction of a rule, but well-typed roles themselves have all their variables bound [8].

The *left-hand side*, or *antecedent*, of a rule is a finite collection of parametric message predicates:

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

Predicate sequences differ from states 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 (this is enforced by the typing rules in [8]). Network and memory predicates will in general contain parametric terms, although not necessarily raw variables as arguments. An example involving network and role state predicates is given in Appendix A. In Section 5, we rely on memory predicates to model the intruder's knowledge. Other usages can be found in [9].

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, in some applications, short-term keys. We rely on the existential quantification symbol to express data generation.

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

We write $[t/x]rhs$ for the capture-free substitution of a term t for a variable x in the consequent rhs [7]. We adopt a similar notation for rules and predicate sequences.

2.5 Roles and Protocol Theories

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.

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

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. 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. We require that the owner of a role ρ be the first argument of all the role state predicates and be the subscript of every memory predicate in ρ. These constraints are formally expressed through the typing and DAS policy of *MSR*.

A *protocol theory*, written $\mathcal{P}$, is a finite collection of roles (see Section 5 and Appendix A for concrete examples):

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

It should be observed that we do not make any special provision for the intruder. The adversary is expressed as a set of roles in the same way as proper protocols, as described in Section 5 for the standard Dolev-Yao intruder.

2.6 Active Roles

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 labeled with a principal name.

$$\begin{array}{llcll}
\textit{Active role sets:} & R & ::= & \cdot & \textit{(Empty active role set)} \\
 & & \mid & R,\ \rho^{\mathsf{A}} & \textit{(Extension with an instantiated role)}
\end{array}$$

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.

3 Data Access Specification

Well-typing does not prevent a rule from looking up and using information its owner should not have access to. For example, the fact that principal A is initiating a session with B shall not allow him/her to access a key that B shares with a server. Similarly, A should not be able to access the memory predicates of any other party. The following role incorporates several violations of this kind:

$$(\forall B\!:\!\mathsf{principal}.\ k_B\!:\!\mathsf{pubK}\ B.\ k'_B\!:\!\mathsf{privK}\ k_B.\ k_{B\mathsf{S}}\!:\!\mathsf{shK}\ B\ \mathsf{S}.\ n\!:\!\mathsf{nonce}.\ \mathsf{N}(\{\!\{n\}\!\}_{k_B})\ \longrightarrow\ \mathsf{N}(\{n\}_{k_{B\mathsf{S}}}),\ \mathsf{M}_B(n))^{\forall A}$$

In this section, we will use the typing declarations of *MSR* to formalize and implement these and other requirements by means of statically checkable *data access specification* (*DAS* for short) judgments. We shall assume that all the expressions we will be analyzing are well-typed. We will start with the presentation of DAS for macroscopic objects such as protocol theories and roles, and only later describe how it is enforced on their components. Therefore, the premises of inference rules will sometimes mention a judgment that has not yet been defined. We mark such occurrences by enclosing them in a $\colorbox{lightgray}{gray box}$ and always give abundant explanations. We ask the reader to ignore the $\boxed{\text{boxed}}$ text that follows most rules in this section. We will interpret it in Section 6.

3.1 Protocol Theories and Roles

The judgment "$\Sigma\ \Vdash\ \mathcal{P}$" expresses the fact that a protocol theory $\mathcal{P}$ realizes correct DAS with respect to a signature Σ. It is implemented by the following three inference rules (in structured operational semantics style), corresponding to the three productions in the syntax of a protocol theory. The right-hand premise of rules **has_grole** and **has_arole** invoke the DAS judgment "$\Gamma\ \Vdash_A\ \rho$" for rule collections, that will be introduced shortly.

$$\frac{}{\Sigma\ \Vdash\ \cdot}\,\textbf{has\_dot}\qquad\quad \frac{\Sigma\ \Vdash\ \mathcal{P}\quad \colorbox{lightgray}{$(\Sigma, A : \mathsf{principal})\ \Vdash_A\ \rho$}}{\Sigma\ \Vdash\ \mathcal{P}, \rho^{\forall A}}\,\textbf{has\_grole}\qquad\quad \frac{\Sigma\ \Vdash\ \mathcal{P}\quad \colorbox{lightgray}{$\Sigma\ \Vdash_{\mathsf{A}}\ \rho$}}{\Sigma\ \Vdash\ \mathcal{P}, \rho^{\mathsf{A}}}\,\textbf{has\_arole}$$

The central rule, which applies to generic roles, pushes the declaration for the role owner A in Σ, with the effect of invoking its right-hand premise with a typing context. Rule **has_arole** deals with anchored roles; since we are working under the assumption that all expressions are well-typed, we do not check that A is has type $\mathsf{principal}$.

Since DAS is about what information the owner of a role is entitled to access, it should come at no surprise that the judgments that operate on rule collections and their components have a principal as a distinguished parameter. We first see this in the rule collection DAS judgment "$\Gamma\ \Vdash_A\ \rho$", where A is the owner of ρ: It is implemented by the following syntax-directed rules:

$$\frac{}{\Gamma\ \Vdash_A\ \cdot}\,\textbf{oas\_dot}\ \boxed{\cdot}\qquad\quad \frac{(\Gamma, L : \vec{\tau})\ \Vdash_A\ \rho}{\Gamma\ \Vdash_A\ \exists L : \vec{\tau}.\,\rho}\,\textbf{oas\_rsp}\ \boxed{\cdot}\qquad\quad \frac{\colorbox{lightgray}{$\Gamma\ \Vdash_A\ r$}\quad \Gamma\ \Vdash_A\ \rho}{\Gamma\ \Vdash_A\ r, \rho}\,\textbf{oas\_rule}\ \boxed{\cdot}$$

Rule **oas_rsp** collects the declaration of an existentially quantified role in the context and verifies its body. We rely on implicit α-conversion to rename L in case a parameter with the same name is already declared in Γ. The rightmost inference rule, **oas_rule**, implements the situation where a collection starts with a rule r. Its left premise validates r itself by means of the DAS judgment for rules, "$\Gamma\ \Vdash_A\ r$".

The judgment "$\Gamma\ \Vdash_A\ r$" expresses DAS-validity for rules. It is implemented by the following two rules:

$$\frac{\colorbox{lightgray}{$\Gamma; \cdot\ \Vdash_A\ \mathit{lhs} > \cdot \gg \Delta$}\quad \colorbox{lightgray}{$\Gamma; \Delta\ \Vdash_A\ \mathit{rhs}$}}{\Gamma\ \Vdash_A\ \mathit{lhs} \rightarrow \mathit{rhs}}\,\textbf{uas\_core}\ \boxed{\cdot}\qquad\qquad \frac{(\Gamma, x : \tau)\ \Vdash_A\ r}{\Gamma\ \Vdash_A\ \forall x : \tau.\, r}\,\textbf{uas\_all}\ \boxed{\cdot}$$

Intuitively, the judgment in the left premise of rule **uas_core**, "$\Gamma; \cdot\ \Vdash_A\ \mathit{lhs} > \cdot \gg \Delta$", collects the data, $\vec{t}$ say, the rule owner A is given in the left-hand side lhs. This includes network messages and previously gathered information stored in memory or role state predicates. This judgment also produces the knowledge context Δ (defined shortly), which contains information that A can reasonably deduce from $\vec{t}$ and later use in the right-hand side. Since it contains information about which key belongs to whom, etc., the context Γ plays an important role in deciding what can legitimately enter Δ. This judgment and its realization are the topic of Section 3.2. Informally, the judgment on the right premise of this rule, "$\Gamma; \Delta \Vdash_A rhs$" uses the knowledge Δ produced by analyzing the antecedent to verify that A can construct all the messages mentioned in the right-hand side rhs. This judgment and the inference rules implementing it are the subject of Section 3.3.

We conclude this section by introducing the notion of *knowledge context*, often simply referred to as *knowledge*. These entities collect the information known to the owner of a rule during DAS-validation. Knowledge is deduced by means of simple inferences from data stored in role state or memory predicates, and messages received from the network. The knowledge context of a rule consists of atomic constants or variables only. Active role sets additionally allow ground terms (obtained by instantiating variables).

$$\begin{array}{lcll}
\textit{Knowledge contexts:} \quad \Delta & ::= & \cdot & \textit{(Empty knowledge context)} \\
& | & \Delta, a & \textit{(Extension with atomic knowledge)} \\
& | & \Delta, x & \textit{(Extension with parametric knowledge)} \\
& | & \Delta, t & \textit{(Extension with ground terms)}
\end{array}$$

We will ignore the last production until Section 3.4, where DAS for active roles is discussed. It is convenient to view knowledge contexts as multisets. A knowledge context Δ is said to be *compatible* with a signature Σ if for each term t in Δ, there is a type τ such that $\Sigma \vdash \tau$ and $\Sigma \vdash t : \tau$.

3.2 Accessing Information in the Left-Hand Side

The left-hand side of a rule gathers the information necessary for constructing the messages transmitted or stored in the consequent. The information in the left-hand side of a rule r consists of the arguments of the role state predicates and the data embedded in the messages received from the network or retrieved from memory predicates. We will now take an informal look at each of these sources:

- The arguments $\vec{e} = (e_1, \ldots, e_n)$ of a role state predicate L represent data passed to a rule from its logical predecessor. The owner of r, call him/her A, knows this information because he/she has put it there. These elementary symbols will generally stand for principal names, keys, or nonces, but variables may also represent complex terms whose inner values A cannot or does not need to access (*e.g.* a message encrypted with a key he/she does not know) [8]. For example, even if it is clear from the protocol at hand that the variable e_3 can only be substituted with a term of the form $\{y\}_k$, it cannot be used to access y, even if e_7 is precisely k. (This form of delayed message interpretation can easily be realized using memory predicates.)
- The term t in an incoming network message $\mathsf{N}(t)$ will generally consist of a number of operators applied to variables (in rare occasions to constants). Some of the associated values are expected to match previously known data (*e.g.* a nonce coming back in response to a challenge), and will be represented by variables listed in a role state predicate. Others will be unknown (*e.g.* a nonce generated by an interlocutor) and shall be bound to previously unused variables. The goal of DAS is to make sure that A has legitimate rights to access this information.
- Finally, A can retrieve previously stored information from a memory predicate $\mathsf{M}_A(\vec{t})$. As for network messages, each term in $\vec{t}$ may consist of a series of constructors applied to variables. Again, writing an argument in this way means accessing the subcomponents corresponding to each constant or variable, with the option of using them in the right-hand side. Observe that the fact that A generated $\mathsf{M}_A(\vec{t})$ does not automatically grant him/her access to the submessages of $\vec{t}$. For example, the third argument t_3 may have the form $\{\!\{t\}\!\}_k$: A is entitled to access t only if he/she is in possession of the private key corresponding to k.

In this section, we will ultimately devise a procedure that certifies that A is entitled to access all the elementary terms mentioned in the antecedent of a rule r. This proceeds in two phases: first we collect the arguments of all the predicates in the left-hand side of r, and then break the composite messages gathered in this way into their elementary components.

The judgment "$\Gamma; \Delta \Vdash_A lhs > \vec{t} \gg \Delta'$" collects the arguments of the predicates in the left-hand side of a rule. Its meta-variables are interpreted as follows: A is the owner of the rule r whose left-hand side we are analyzing. Γ is the typing context of r. The predicate sequence lhs is the portion of the antecedent of r that has still to be examined. The terms $\vec{t}$ are the arguments that have been gathered so far and that may need further processing. The *input knowledge* Δ lists collected arguments that are known to be elementary. Finally, the *output knowledge* Δ' stands for the elementary information that will ultimately be extracted from r's left-hand side. It is convenient to interpret this judgment operationally as a partial function that given A, Γ, lhs, $\vec{t}$ and Δ computes a value for Δ' if the DAS policy is obeyed. We shall interpret $\vec{t}$ as a multiset. As we start processing the antecedent of a rule in rule **uas_core**, Δ and $\vec{t}$ are empty (written "$\cdot$") as no argument has yet been collected.

Our first rule describes how a role state predicate $L(A, \vec{e})$ is processed. Remember that, by definition, L is a parameter, its first argument A is a principal name, and the terms $(A, \vec{e})$ must be either constants or variables. Therefore, each object among $(A, \vec{e})$ is an elementary piece of information. We can therefore merge $(A, \vec{e})$ into the current input knowledge context Δ and use the resulting knowledge context Δ' to analyze the remaining predicates lhs. We have the following rule, which makes use of the merge judgment "$\Delta \;>\; \vec{e} \;>\; \Delta'$", whose simple realization is given in Appendix C.

$$\dfrac{\Delta \;>\; (A, \vec{e}) \;>\; \Delta' \qquad \Gamma; \Delta' \Vdash_A \; lhs > \vec{t}\,' \gg \Delta''}{\Gamma; \Delta \Vdash_A \; (L(A, \vec{e}), lhs) > \vec{t}\,' \gg \Delta''}\;\textbf{las\_rsp}^{\sharp}\;\boxed{\cdot}$$

We next turn to network and memory predicates in the antecedent of a rule. Since the messages in their arguments may not be elementary, we shall include them in the list of unprocessed arguments $\vec{t}\,'$ before examining the remaining predicates lhs. Only memory predicates belonging to A are accepted.

$$\dfrac{\Gamma; \Delta \Vdash_A \; lhs > (t, \vec{t}\,') \gg \Delta''}{\Gamma; \Delta \Vdash_A \; (\mathsf{N}(t), lhs) > \vec{t}\,' \gg \Delta''}\;\textbf{las\_net}\;\boxed{\textbf{INT}} \qquad\qquad \dfrac{\Gamma; \Delta \Vdash_A \; lhs > (\vec{t}, \vec{t}\,') \gg \Delta'}{\Gamma; \Delta \Vdash_A \; (\mathsf{M}_A(\vec{t}), lhs) > \vec{t}\,' \gg \Delta'}\;\textbf{las\_mem}\;\boxed{\cdot}$$

Once the arguments of all the predicates on the left-hand side of the rule have been collected, we move to the second phase which ascertains that the uninterpreted terms $\vec{t}$ satisfy the DAS policy. This is done in the following rule by invoking the judgment "$\Gamma; \Delta \Vdash_A \; \vec{t} \gg \Delta'$", discussed below.

$$\dfrac{\Gamma; \Delta \Vdash_A \; \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A \; \cdot > \vec{t} \gg \Delta'}\;\textbf{las\_dot}\;\boxed{\cdot}$$

The judgment "$\Gamma; \Delta \;\; \Vdash_A \;\; \vec{t} \;\gg\; \Delta'$", used in rule **las_dot**, examines possibly composite terms. The interpretation of its meta-variable is inherited from the argument collection judgment above. Again, this judgment can be seen as a partial function that computes a value for Δ' when given A, Γ, Δ and $\vec{t}$. It should be observed that A does have legitimate access to each term in $\vec{t}$: we want to verify that this property extends to their subterms.

Our first two rules deal with unchecked elementary messages e. There are two possibilities: either e is known and therefore appears in the current input knowledge, or it must be looked up in the typing context Γ.

$$\dfrac{\Gamma; (\Delta, e) \Vdash_A \; \vec{t} \gg \Delta'}{\Gamma; (\Delta, e) \Vdash_A \; e, \vec{t} \gg \Delta'}\;\textbf{tas\_kn}^{\sharp}\;\boxed{\textbf{DEL}} \qquad\qquad \dfrac{(\Gamma, e : \tau, \Gamma'); (\Delta, e) \Vdash_A \; \vec{t} \gg \Delta'}{(\Gamma, e : \tau, \Gamma'); \Delta \Vdash_A \; e, \vec{t} \gg \Delta'}\;\textbf{tas\_ukn}\;\boxed{\cdot}$$

The rule owner A can access the cleartext t of an encrypted message $\{t\}_k$ (or $\{\!\{t\}\!\}_k$) only if he/she is entitled to access the decryption key corresponding to k. This is ascertained by the left premises of the following rules. The judgment $\Gamma; \Delta \;\; \Vdash^{s}_{A}$ $k \gg \Delta'$ (resp. $\Gamma; \Delta \;\; \Vdash^{a}_{A} \;\; k \gg \Delta'$) verifies that A can access k (resp. its inverse) and if necessary updates the knowledge context Δ to Δ'. Once the key has been resolved, the cleartext t is put back in the pool of pending messages, which is recursively analyzed in the rightmost premise.

$$\dfrac{\Gamma; \Delta \Vdash^{s}_{A} \; k \gg \Delta' \qquad \Gamma; \Delta' \Vdash_A \; t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \; \{t\}_k, \vec{t} \gg \Delta''}\;\begin{array}{c}\textbf{tas\_ske}\\ \boxed{\textbf{SDC}}\end{array} \qquad\qquad \dfrac{\Gamma; \Delta \Vdash^{a}_{A} \; k \gg \Delta' \qquad \Gamma; \Delta' \Vdash_A \; t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \; \{\!\{t\}\!\}_k, \vec{t} \gg \Delta''}\;\begin{array}{c}\textbf{tas\_pke}\\ \boxed{\textbf{PDC}}\end{array}$$

Concatenated messages can be split unconditionally before recursively analyzing their submessages. Once all possibly composite terms have been reduced to their elementary constituents (and have been shown to respect the DAS policy), we simply return the accumulated input knowledge context as the output knowledge.

$$\dfrac{\Gamma; \Delta \Vdash_A \; t_1, t_2, \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A \; (t_1\; t_2), \vec{t} \gg \Delta'}\;\textbf{tas\_cnc}\;\boxed{\textbf{DCM}} \qquad\qquad \dfrac{}{\Gamma; \Delta \Vdash_A \; \cdot \gg \Delta}\;\textbf{tas\_dot}\;\boxed{\cdot}$$

We conclude the treatment of the left-hand side of a rule by devising a method to establish when the owner of a rule can decipher (and therefore access) a message encrypted with a key k. Since we assumed in Section 2 to have two kinds of encryption operations (shared-key and public-key), we will present two judgments and the relative rules. It should be noted that richer schemes, *e.g.* including digital signatures or a more refined key taxonomy, would need to define additional judgments and to provide the corresponding DAS rules.

We express the fact that the owner A of a rule can access the cleartext t of a message $\{t\}_k$ encrypted with a shared key k by means of the judgment "$\Gamma; \Delta \ \Vdash^s_A \ k \gg \Delta'$", where Γ, Δ and Δ' are the typing context, and the input and output knowledge respectively. Again, Δ' is computed from the other entities in this relation. In order for A to decrypt $\{t\}_k$, he/she must have access to k itself since we are in a symmetric-key setting. There are two scenarios to analyze in order to decide this judgment. First, A may know k, for example if it was previously transmitted in the clear. Then, k can be found in the input knowledge context.

$$\frac{}{\Gamma; (\Delta, k) \ \Vdash^s_A \ k \gg (\Delta, k)}\ \textbf{kas\_ss}\ \ \boxed{\textbf{DUP}}$$

The second scenario involves a key A does not know (yet) about, but to which he/she has legitimate access. A principal has the right to access a shared key only if this key was intended to communicate with him/her:

$$\frac{}{(\Gamma, k : \mathsf{shK}\ A\ B, \Gamma'); \Delta \ \Vdash^s_A \ k \gg (\Delta, k)}\ \textbf{kas\_su1}\ \ \boxed{\textbf{IS1}} \qquad \frac{}{(\Gamma, k : \mathsf{shK}\ B\ A, \Gamma'); \Delta \ \Vdash^s_A \ k \gg (\Delta, k)}\ \textbf{kas\_su2}\ \ \boxed{\textbf{IS2}}$$

Observe that the relationship between the key owner and the rule owner is encoded in the dependent type that qualifies the key itself. Since k was unknown to A but is being accessed, we include it among the output knowledge of these rules.

The judgment "$\Gamma; \Delta \ \Vdash^a_A \ k \gg \Delta'$" expresses the similar relation concerning public-key encryption, where the meaning of the meta-variables is as for symmetric keys. In order to decipher a message encrypted with a public key k, we must have access to the corresponding private key, call it k'. As for shared keys, the first place where to look is the current knowledge context. If the private key k' of some principal B has previously been encountered, then we can decipher transmissions encoded with the corresponding public key k.

$$\frac{}{(\Gamma, k : \mathsf{pubK}\ B, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k') \ \Vdash^a_A \ k \gg (\Delta, k')}\ \textbf{kas\_pus}\ \ \boxed{\textbf{DUP}}$$

If A does not know k', then he/she is entitled to access the cleartext of the encrypted message $\{\!\{t\}\!\}_k$ only if he/she owns k:

$$\frac{}{(\Gamma, k : \mathsf{pubK}\ A, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); \Delta \ \Vdash^a_A \ k \gg (\Delta, k')}\ \textbf{kas\_puu}\ \ \boxed{\textbf{IPV}, \textbf{DUP}}$$

3.3 Processing Information in the Right-Hand Side

The right-hand side of a rule is where messages are constructed, either to be emitted over the public network, or stored for future use. However, the first rule of an initiator role will generally have an empty left-hand side, and yet it can send complex messages in its consequent. Therefore, the right-hand side of a rule can also access data on its own, information that is not mentioned in its antecedent. This can happen in two ways: first by generating fresh data (*e.g.* nonces), and second by using information that is "out there" (*e.g.* the name of an interlocutor, or a key shared with him/her). Both alternatives have the potential of violating the DAS policy (*e.g.* when trying to access the private key of a third party).

DAS on the right-hand side rhs of a rule r is expressed by the judgment: "$\Gamma; \Delta \ \Vdash_A \ rhs$", where A is the owner of r, Γ is its typing context, and Δ is the knowledge gained by examining its antecedent, it is implemented by rules whose number depends on the intended application of the protocol at hand. Given a consequent of the form "$\exists x : \tau.\, rhs$". It is tempting to indiscriminately add x to the current knowledge context and proceed with the validation of rhs. This is in general inappropriate since it would allow any principal to construct information that can potentially affect the rest of the system. In most protocols, nobody should be allowed to create new principals. Similarly, only key-distribution protocols should enable a principal to create keys, and typically only short-term keys. On the other hand, principals will generally be allowed to generate nonces and atomic messages (*e.g.* an intruder may want to fake a credit card number). These considerations produce a family of rewrite rules that differ only by the type of the existential declaration they consider. In all cases, we recursively check the body rhs after inserting "$x : \tau$" in the context (for appropriate τ's) and adding x to the current knowledge:

$$\frac{(\Gamma, x : \mathsf{nonce}); (\Delta, x) \ \Vdash_A \ rhs}{\Gamma; \Delta \ \Vdash_A \ \exists x : \mathsf{nonce}.\, rhs}\ \textbf{ras\_nnc}\ \boxed{\textbf{GNC}} \qquad \frac{(\Gamma, x : \mathsf{msg}); (\Delta, x) \ \Vdash_A \ rhs}{\Gamma; \Delta \ \Vdash_A \ \exists x : \mathsf{msg}.\, rhs}\ \textbf{ras\_msg}\ \boxed{\textbf{GMS}} \qquad \frac{\Gamma; \Delta \looparrowright_A \ lhs}{\Gamma; \Delta \ \Vdash_A \ lhs}\ \textbf{ras\_ps}\ \boxdot$$

We must emphasize again that the exact set of rules for data generation depends on the intended functionalities of the protocol. Rule **ras_ps** invokes the predicate sequence validation judgment "$\Gamma; \Delta \looparrowright_A \ lhs$", discussed shortly, to verify that the inner core lhs of the rule's consequent satisfies DAS.

The premises of rule **ras_ps** above included the judgment “$\Gamma;\Delta \looparrowright_A \ \mathit{lhs}$” which verifies that all the messages in the predicate sequence lhs can be constructed in the current rule. It is implemented by the following rules. When lhs is not empty, we rely on the term constructions judgments “$\Delta \looparrowright t$” and “$\Delta \looparrowright \vec{t}$” that will be explained shortly.

$$\frac{}{\Gamma;\Delta \looparrowright_A \ \cdot}\ \textbf{ras\_dot}\ \boxed{\textbf{DEL}(\Delta)} \qquad \frac{\Gamma;\Delta \looparrowright_A \ t \quad \Gamma;\Delta \looparrowright_A \ \mathit{lhs}}{\Gamma;\Delta \looparrowright_A \ \mathsf{N}(t), \mathit{lhs}}\ \textbf{ras\_net}\ \boxed{\textbf{DUP}(\Delta), \textbf{TRN}}$$

$$\frac{\Gamma;\Delta \looparrowright_A \ \vec{t} \quad \Gamma;\Delta \looparrowright_A \ \mathit{lhs}}{\Gamma;\Delta \looparrowright_A \ \mathsf{M}_A(\vec{t}), \mathit{lhs}}\ \textbf{ras\_mem}\ \boxed{\textbf{DUP}(\Delta)} \qquad \frac{\Gamma;\Delta \looparrowright_A \ (A,\vec{e}) \quad \Gamma;\Delta \looparrowright_A \ \mathit{lhs}}{\Gamma;\Delta \looparrowright_A \ L(A,\vec{e}), \mathit{lhs}}\ \textbf{ras\_rsp}^{\sharp}\ \boxed{\textbf{DUP}(\Delta)}$$

Empty predicate sequences are always valid. Moreover, a principal A is allowed to publish any information he/she can construct on the public network, but he/she shall be able to update only his/her own role state and memory predicates.

The constructibility of a term t in the right-hand side of a rule is expressed by the judgment “$\Gamma;\Delta \looparrowright_A \ t$”. Elementary information appearing in a rule consequent can come from of two sources: rule **cas_kn** handles the case where it has been collected in the knowledge context while validating the antecedent and fresh data declarations of a rule. This can also be the first appearance of this information in the rule, in which case we must verify that the role owner is effectively entitled to access it. This achieved in rule **cas_ukn** through the right-hand side access judgment “$\Gamma \looparrowright_A \ e$”, discussed below.

$$\frac{}{\Gamma;(\Delta,e) \looparrowright_A \ e}\ \textbf{cas\_kn}^{\sharp}\ \boxed{\textbf{DEL}(\Delta)} \qquad \frac{\Gamma \looparrowright_A \ e}{\Gamma;\Delta \looparrowright_A \ e}\ \textbf{cas\_ukn}\ \boxed{\textbf{DEL}(\Delta)}$$

The simple rules implementing composite terms can be found in Appendix C, together with the implementation of the judgment “$\Gamma;\Delta \looparrowright_A \ \vec{t}$” that allows constructing message tuples $\vec{t}$ from the knowledge Δ at hand.

We conclude this section by describing the judgment “$\Gamma \looparrowright_A \ e$” that checks that a rule owner A has legitimate access to elementary data e that appear only in the rule consequent. Its implementation depends entirely on the atomic data that can be part of a message and therefore on the types that have been defined to classify them. We will now present inference rules relative to the types defined in Section 2, but it should be clear that different type layouts will require different rules.

Let us start with non-dependent types. We should clearly be able to access any principal name:

$$\frac{}{(\Gamma, e : \mathsf{principal}, \Gamma') \looparrowright_A \ e}\ \textbf{eas\_pr}\ \boxed{\textbf{IPR}}$$

The remaining simple types are nonce and msg. Were we to have a rule similar to **eas_pr** for nonces would allow A to access any nonce in the system, including nonces that he/she has not generated. This is clearly undesirable. The only nonces A is entitled to access are the ones he/she has created and the ones he/she has retrieved in received messages or as previously stored data. In all these cases, these nonces are included in some knowledge context. A similar argument applies to elementary objects of type msg: a rule akin to **eas_pr** would give A access to any message that can be constructed in the system, when invoked with a variable. This is particularly undesirable since msg is a supersort of all our types (see Section 2.3).

Next, we consider keys: A should have free access to all of his/her shared keys, but not to others; similarly, A has legitimate access to his/her private keys, and to the public keys of any principal.

$$\frac{}{(\Gamma, e : \mathsf{shK}\ A\ B, \Gamma') \looparrowright_A \ e}\ \textbf{eas\_s1}\ \boxed{\textbf{IS1}} \qquad \frac{}{(\Gamma, e : \mathsf{shK}\ B\ A, \Gamma') \looparrowright_A \ e}\ \textbf{eas\_s2}\ \boxed{\textbf{IS2}}$$

$$\frac{}{(\Gamma, e : \mathsf{privK}\ k, \Gamma', k : \mathsf{pubK}\ A, \Gamma'') \looparrowright_A \ e}\ \textbf{eas\_pp}\ \boxed{\textbf{IPV}} \qquad \frac{}{(\Gamma, e : \mathsf{pubK}\ B, \Gamma') \looparrowright_A \ e}\ \textbf{eas\_p}\ \boxed{\textbf{IPB}}$$

It should be observed that protocols that make use of a key distribution center should not rely on these rules. These kinds of protocols require a language and type layout that is more elaborate than the one in our running example.

3.4 Active Roles

In Section 2.6 we defined an active role as a role suffix whose free variables have been instantiated to ground terms. They correspond to roles in the midst of execution. Active roles should clearly be subject to the same access constraints as protocol theories. They are handled by allowing ground terms anywhere variables can appear in a role, and by treating them in the same way. Formally, for every of the above rules that look up or store (elementary) information in an active role set, we introduce a variant that performs the same operation on a ground term. The affected rules are marked with the symbol $\sharp$ in the above discussion and in Appendix C. Furthermore, since execution may instantiate a role state predicate parameter L with a constant L_l, we need additional variants of rules **las_rsp** and **ras_rsp**.

The judgment "$\Sigma \ \Vdash \ R$" expresses the fact that an active role set R satisfies our DAS policy in signature Σ. It is implemented by the following two simple rules:

$$\frac{}{\Sigma \ \Vdash \ \cdot}\ \textbf{aas\_dot}\ \boxed{\cdot} \qquad\qquad \frac{\Sigma \ \Vdash \ R \quad \Sigma \ \Vdash_{\mathsf{A}} \ \rho}{\Sigma \ \Vdash \ R, \rho^{\mathsf{A}}}\ \textbf{aas\_ext}\ \boxed{\rho^{\mathsf{A}} \quad \text{if } A \neq \mathsf{I}}$$

3.5 Decidability of DAS

All the judgments presented in this section have decidable implementations. Furthermore, the ones to which we have ascribed a functional behavior implement computable relations. For space reasons, we only give a sketch of the argument underlying this result and a condensed statement relative to protocol theories and active role sets only. A detailed proof of this statement for each of these judgments can be found in [6].

Property 3.1 *Given a signature Σ, a protocol theory $\mathcal{P}$ and an active role set R, it is decidable whether the judgments $\Sigma \ \Vdash \ \mathcal{P}$ and $\Sigma \ \Vdash \ R$ hold.*

Proof: All DAS rules are syntax-directed and, with the exception of **uas_core**, **las_rsp** (and variants), **tas_ske** and **tas_pke**, none contains meta-variables in its premises that are not also mentioned in its conclusion. The leftmost premise of each of these rules is a left-hand side judgment J that produces an output knowledge context Δ' (the one meta-variable that does not appear in their conclusion). Thus, we reduce our decidability result to proving that there are only finitely many such Δ' for which J is derivable, assuming all other parameters fixed. A close inspection of the DAS rules reveals a number of pairs of rules (for example **tas_ukn** and **tas_kn**) that may both apply in certain situations, and therefore have J succeed with two output knowledge contexts. In the worst case, the number of output knowledge contexts is exponential (but finite) in the number of symbols appearing in the other parts of J. □

When validating rules, alternative output knowledge contexts are identical up to the duplication of data. On the basis of this observation, we claim that DAS can be implemented with a complexity linear in the number of elementary terms in a rule. This argument extends to active role sets by prioritizing rules that look information up in the current knowledge context (*e.g.* **tas_kn**).

4 Execution Model

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'. Referring to the situation that the execution of a protocol has reached by means of a state is an oversimplification. Indeed, execution operates on *configurations* $[S]^{R}_{\Sigma}$ consisting of a state S, an active role set R and a signature Σ: R records the roles that can be used in order to continue the execution, at which point they were stopped, and how they were instantiated, while Σ is needed to ensure that variable instantiation is well-typed. No element in a configuration contains free variables. Configurations will be indicated with the letter C.

Given a protocol $\mathcal{P}$, we describe the fact that execution transforms a configuration C into C' in one step by means of the judgment "$\mathcal{P} \triangleright C \ \longrightarrow \ C'$". The next two rules specify how to extend the current active role set R with a role from $\mathcal{P}$.

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

Anchored roles are simply copied to the current active role sets since their syntax meets the requirements for active roles. We instead make a generic role available for execution in rule **ex_grole** by assigning it an owner. The typing judgment in its premise makes sure that A is defined as a principal name.

Once a role has been activated, 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. In rule **ex_rsp**, L_l shall be a new symbol that appears nowhere in the current configuration (in particular it should not occur in Σ).

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

Rule $\mathbf{ex\_all}$ instantiates the universal variables that may appear in a rule. 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 legitimate observation: the rule above provides an idealized model of the execution rather than the basis for the implementation of an actual simulator. An operational model suited for implementation is the subject of current research. It should also be observed that the premise of $\mathbf{ex\_all}$ describes A's acceptance of t as a term of type τ. How this happens is kept abstract, but it should correspond to some lower level mechanism to adequately express the protocol at hand.

We now consider execution steps resulting from the application of a fully instantiated rule $(lhs \to rhs)$ from the current active role set R. The antecedent lhs must be ground and therefore it has the structure of a legal state. This rules identifies lhs in the current state and replaces it with a substate lhs' derived from the consequent rhs by instantiating its existential variables with fresh constants of the appropriate type. This latter operation is performed in the premise of this rule by the right-hand side instantiation judgment "$(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}$", whose implementation is given in Appendix C.

$$\frac{(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}}{\mathcal{P} \triangleright [S, lhs]_{\Sigma}^{R,((lhs \to rhs),\rho)^{\mathsf{A}}} \;\longrightarrow\; [S, lhs']_{\Sigma'}^{R,(\rho)^{\mathsf{A}}}}\;\mathbf{ex\_core}$$

Security protocols often allow various forms of branching. In a protocol theory, the control structure is mostly realized by the role state predicates appearing in a role. Branching can indeed 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.

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

Rule $\mathbf{ex\_dot}$ does some housekeeping by throwing away active roles that have been completely executed.

The judgment "$\mathcal{P} \triangleright C \;\longrightarrow^{*}\; C'$" allow chaining atomic transitions into multi-step firings. It is defined in Appendix C as the reflexive and transitive closure of the above one-step relation. A parallel version of this judgment has been defined in [7]. Moreover, we have proved in [8] that well-typing is preserved by execution, *i.e.* that when starting from well-typed objects firing will always produce well-typed entities (a detailed proof can be found in [6]).

A similar result applies to DAS. Indeed, the DAS Preservation Theorem below states that, under reasonable typing assumptions, no execution sequence can take a configuration that satisfies the DAS policy to a situation that violates it. In particular, instantiating variables cannot invalidate DAS.

Theorem 4.1 (*DAS Preservation*)

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

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

then the judgments $\Sigma' \Vdash \mathcal{P}$ *and* $\Sigma' \Vdash R'$ *are derivable.*

Proof: The proof proceeds by induction on a derivation of the given execution judgment. Rules $\mathbf{ex\_rsp}$ and $\mathbf{ex\_core}$ rely on a *Weakening Lemma* that allows extending the signature of an DAS judgment without affecting its derivability. Rules $\mathbf{ex\_grole}$ and $\mathbf{ex\_all}$ make use of *Substitution Lemma* that states that DAS is preserved under substitution, assuming some simple preconditions are met. □

Because of its passive role, the state S is not required to be well typed for this result to hold, although applications will generally operate on well-typed states. This theorem and the fact that the execution rules do not depend on any DAS judgment makes DAS verification a purely static check.

5 The Dolev-Yao Intruder

The *Dolev-Yao abstraction* [15, 21] assumes that elementary data such as principal names, keys and nonces are atomic symbols rather than the bit-strings implemented in practice. Furthermore, it views the operations needed to assemble messages, *i.e.* concatenation and encryption, as pure constructors in an initial algebra. Therefore, for example, a term of the form $\{t\}_k$ cannot be mistaken for a concatenation $(t_1\, t_2)$, and $\{t\}_k = \{t'\}_{k'}$ 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 attack 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:

1. Intercept and learn messages.
2. Transmit known messages.
3. Decompose concatenated messages he has learned.
4. Concatenate known messages.
5. Decipher encrypted messages if he knows the keys.
6. Encrypt known messages with known keys.
7. Access public information.
8. Generate fresh data.

MSR, like most current systems geared toward specifying security protocol, is an instance of the the Dolev-Yao abstraction. Elementary data are indeed atomic, messages 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*.

Let I be the elected intruder. We represent the knowledge I has at his disposal to mount an attack in a distributed fashion as a collection of memory predicates of the form $\mathit{I}(t)$ for all known terms t (for conciseness, the subscript "$_\mathsf{I}$" of the correct form $\mathit{I}_\mathsf{I}(t)$ is kept implicit). Thus, the declarations "$\mathsf{I} : \mathsf{principal}$" and "$\mathit{I} : \mathsf{principal} \times \mathsf{msg}$" constitute the *standard Dolev-Yao intruder signature*, that we denote Σ_{DY}. We express each of the Dolev-Yao intruder's capabilities as one or more one-rule roles anchored at I. We give them a name (written in bold to its left) that will be referred to in Section 6. We also organize rule constituents in columns for legibility. These roles constitute the *standard Dolev-Yao intruder theory* that we denote $\mathcal{P}_{DY}$.

Items (1) and (2) of the description of the Dolev-Yao intruder are specified by rules $\mathbf{INT}$ and $\mathbf{TRN}$ below, respectively. The former captures a network message $\mathsf{N}(t)$ and stores its contents in the intruder's 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. Rule $\mathbf{TRN}$ emits a memorized message out in the public network.

$$\mathbf{INT}\text{:}\ \big(\forall t : \mathsf{msg}. \quad \mathsf{N}(t) \quad \rightarrow \quad \mathit{I}(t)\big)^{\mathsf{I}} \qquad\qquad \mathbf{TRN}\text{:}\ \big(\forall t : \mathsf{msg}. \quad \mathit{I}(t) \quad \rightarrow \quad \mathsf{N}(t)\big)^{\mathsf{I}}$$

From now on, we will only deal with the memory predicate $\mathit{I}(\_)$, which acts as a workshop where I can dismantle intercepted communications and counterfeit messages. Concatenated messages can be taken apart and constructed at will:

$$\mathbf{DCM}\text{:}\ \left(\forall t_1, t_2 : \mathsf{msg}. \quad \mathit{I}(t_1\ t_2) \quad \rightarrow \quad \begin{array}{l}\mathit{I}(t_1)\\ \mathit{I}(t_2)\end{array}\right)^{\mathsf{I}} \qquad\qquad \mathbf{CMP}\text{:}\ \left(\forall t_1, t_2 : \mathsf{msg}. \quad \begin{array}{l}\mathit{I}(t_1)\\ \mathit{I}(t_2)\end{array} \quad \rightarrow \quad \mathit{I}(t_1\ t_2)\right)^{\mathsf{I}}$$

Items (5) and (6) of the above specification state 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.

$$\mathbf{SDC}\text{:}\ \left(\begin{array}{l}\forall A, B : \mathsf{principal}.\\ \forall k : \mathsf{shK}\ A\ B.\\ \forall t : \mathsf{msg}.\end{array} \quad \begin{array}{l}\mathit{I}(\{t\}_k)\\ \mathit{I}(k)\end{array} \quad \rightarrow \quad \mathit{I}(t)\right)^{\mathsf{I}} \qquad\qquad \mathbf{SEC}\text{:}\ \left(\begin{array}{l}\forall A, B : \mathsf{principal}.\\ \forall k : \mathsf{shK}\ A\ B.\\ \forall t : \mathsf{msg}.\end{array} \quad \begin{array}{l}\mathit{I}(t)\\ \mathit{I}(k)\end{array} \quad \rightarrow \quad \mathit{I}(\{t\}_k)\right)^{\mathsf{I}}$$

$$\mathbf{PDC}\text{:}\ \left(\begin{array}{l}\forall A : \mathsf{principal}.\\ \forall k : \mathsf{pubK}\ A.\\ \forall k' : \mathsf{privK}\ k.\\ \forall t : \mathsf{msg}.\end{array} \quad \begin{array}{l}\mathit{I}(\{\!\!\{t\}\!\!\}_k)\\ \mathit{I}(k')\end{array} \quad \rightarrow \quad \mathit{I}(t)\right)^{\mathsf{I}} \qquad\qquad \mathbf{PEC}\text{:}\ \left(\begin{array}{l}\forall A : \mathsf{principal}.\\ \forall k : \mathsf{pubK}\ A.\\ \forall t : \mathsf{msg}.\end{array} \quad \begin{array}{l}\mathit{I}(t)\\ \mathit{I}(k)\end{array} \quad \rightarrow \quad \mathit{I}(\{\!\!\{t\}\!\!\}_k)\right)^{\mathsf{I}}$$

We now tackle the often overlooked item (7) of the Dolev-Yao intruder specification: the ability to access public information. The intruder 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. Our DAS policy already enforces this kind of requirements. Therefore, we will express the capabilities of the intruder with respect to public information access by means of the strongest rules that satisfy DAS.

$$\mathbf{IPR}\text{:}\ \big(\forall A : \mathsf{principal}. \quad \cdot \quad \rightarrow \quad \mathit{I}(A)\big)^{\mathsf{I}}$$

$$\mathbf{IS1}\text{:}\ \left(\begin{array}{l}\forall A : \mathsf{principal}.\\ \forall k : \mathsf{shK}\ \mathsf{I}\ A.\end{array} \quad \cdot \quad \rightarrow \quad \mathit{I}(k)\right)^{\mathsf{I}} \qquad\qquad \mathbf{IS2}\text{:}\ \left(\begin{array}{l}\forall A : \mathsf{principal}.\\ \forall k : \mathsf{shK}\ A\ \mathsf{I}.\end{array} \quad \cdot \quad \rightarrow \quad \mathit{I}(k)\right)^{\mathsf{I}}$$

$$\mathbf{IPB}\text{:}\ \left(\begin{array}{l}\forall A : \mathsf{principal}.\\ \forall k : \mathsf{pubK}\ A.\end{array} \quad \cdot \quad \rightarrow \quad \mathit{I}(k)\right)^{\mathsf{I}} \qquad\qquad \mathbf{IPV}\text{:}\ \left(\begin{array}{l}\forall k : \mathsf{pubK}\ \mathsf{I}.\\ \forall k' : \mathsf{privK}\ k.\end{array} \quad \cdot \quad \rightarrow \quad \mathit{I}(k')\right)^{\mathsf{I}}$$

The last item of the specification of the Dolev-Yao intruder hints at the fact that he should be able to create fresh data. We must again be very careful when implementing this requirement: in most scenarios, it is inappropriate for I to generate keys or to create new principals. As for the DAS rules, nonces and atomic messages are however risk-frees.

$$\textbf{GNC}\!: \left(\;\cdot\; \rightarrow \;\exists n : \mathsf{nonce}. \;\; \mathit{I}(n) \right)^{\mathsf{I}} \qquad\qquad \textbf{GMS}\!: \left(\;\cdot\; \rightarrow \;\exists m : \mathsf{msg}. \;\; \mathit{I}(m) \right)^{\mathsf{I}}$$

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 public-key authentication protocol presented in [9] is accurately validated only in the presence of an intruder who can create public keys.

Last, $\mathcal{P}_{DY}$ contains the following two administrative rules that allow the Dolev-Yao intruder to forget information and to duplicate (and therefore reuse) fabricated data, respectively.

$$\textbf{DEL}\!: \left(\forall t : \mathsf{msg}. \;\; \mathit{I}(t) \;\; \rightarrow \;\; \cdot \right)^{\mathsf{I}} \qquad\qquad \textbf{DUP}\!: \left(\forall t : \mathsf{msg}. \;\; \mathit{I}(t) \;\; \rightarrow \;\; \begin{array}{l} \mathit{I}(t) \\ \mathit{I}(t) \end{array} \right)^{\mathsf{I}}$$

It is easy to verify that the above *MSR* formalization of the Dolev-Yao intruder is well-typed and satisfies DAS:

Property 5.1 *The judgments* $\vdash \Sigma_{DY}$, $\Sigma_{DY} \vdash \mathcal{P}_{DY}$ *and* $\Sigma_{DY} \Vdash \mathcal{P}_{DY}$ *are derivable.*

The validation of the judgment "$\Sigma_{DY} \Vdash \mathcal{P}_{DY}$" makes use of all the DAS rules in Section 3, except the ones dealing with role state predicates.

A few aspects of this encoding deserve to be emphasized: first, this specification lies completely within *MSR* and can therefore be adapted, were the protocol at hand to require it. This differentiates *MSR* from most other formalisms which either rely on a fixed intruder, or express it in a language distinct from regular protocols. Second, typing allows a very precise characterization of what the intruder's capabilities actually are, especially as far as access to public information and fresh data generation are concerned. Third, $\mathcal{P}_{DY}$ can be automatically generated from DAS rules of the given term language [10].

6 The Most Powerful Symbolic Attacker

The Dolev-Yao intruder is by no means the only way to specify a protocol adversary. Indeed, *MSR* allows writing attacker theories of much greater complexity by using multi-rule roles, branching, long predicate sequences, diversified memory predicates, and deep pattern-matching. It is however commonly believed that any attack mounted by such an attacker can be uncovered by using the Dolev-Yao intruder. The assumption that the Dolev-Yao intruder subsumes any other symbolic adversary (*i.e.* that plays by the rules of the Dolev-Yao abstraction) is built into the most successful security protocol verification systems [4, 14, 18, 20, 22, 24, 25]. To our knowledge and from discussions with several security experts, it appears that this strongly held belief has never been proved. This is worrisome considering the seldom-acknowledged subtleties that our formalization of the Dolev-Yao intruder has exposed in Section 5. Our precise definition of DAS and the fact that an attacker is specified within *MSR* as any other protocol fragment give us the means to phrase that question and to formally prove that it has a positive answer. We dedicate this section to this task.

Again, let I be the intruder (we will consider situations involving multiple intruders at the end of this section). Assume that we are given a derivation of a generic well-typed and DAS-valid execution judgment $\mathcal{P} \rhd [S]_{\Sigma}^{R} \longrightarrow^{*} [S']_{\Sigma'}^{R'}$. Clearly, $\mathcal{P}$, R and R' can mention arbitrary (active) roles anchored on the intruder. Similarly, S and S' can contain role state and memory predicates belonging to I. We will show that we can construct an encoding $\ulcorner\_\urcorner$ for the entities appearing in that judgment such that: 1) $\ulcorner\mathcal{P}\urcorner$, $\ulcorner R\urcorner$ and $\ulcorner R'\urcorner$ do not mention any intruder specification besides $\mathcal{P}_{DY}$; 2) $\ulcorner S\urcorner$ and $\ulcorner S'\urcorner$ do not contain any role state predicate for I nor any intruder memory predicate except at most $\mathit{I}(\_)$; and 3) the judgment $\ulcorner\mathcal{P}\urcorner, \mathcal{P}_{DY} \rhd [\ulcorner S\urcorner]_{\ulcorner\Sigma\urcorner}^{\ulcorner R\urcorner} \longrightarrow^{*} [\ulcorner S'\urcorner]_{\ulcorner\Sigma'\urcorner}^{\ulcorner R'\urcorner}$ is derivable.

The encoding $\ulcorner\mathcal{P}\urcorner$ of a protocol theory $\mathcal{P}$ implements the idea that every role anchored on the intruder can be emulated by means of $\mathcal{P}_{DY}$. Therefore, we simply filter out every component of the form $(\rho)^{\mathsf{I}}$:

$$\left[\begin{array}{lcl} \ulcorner . \urcorner & = & \cdot \\ \ulcorner \mathcal{P}, (\rho)^{\forall A} \urcorner & = & \ulcorner\mathcal{P}\urcorner, \; (\rho)^{\forall A} \\ \ulcorner \mathcal{P}, (\rho)^{\mathsf{A}} \urcorner & = & \begin{cases} \ulcorner\mathcal{P}\urcorner, \; (\rho)^{\mathsf{A}} & \text{if } \mathsf{A} \neq \mathsf{I} \\ \ulcorner\mathcal{P}\urcorner & \text{otherwise} \end{cases} \end{array}\right.$$

The Dolev-Yao intruder model does not refer to any role state or memory predicate beside $\mathit{I}(\_)$. Whenever one of these objects appears in a state S, the encoding $\ulcorner S\urcorner$ will account for it by including one instance of the Dolev-Yao intruder memory predicate $I(\_)$ for each of its arguments, as specified by the following definition:

$$\left[\begin{array}{lcl} \ulcorner \cdot \urcorner & = & \cdot \\ \ulcorner S, \mathsf{N}(t) \urcorner & = & \ulcorner S \urcorner, \mathsf{N}(t) \\ \ulcorner S, \mathsf{M}_{\mathsf{A}}(\vec{t}) \urcorner & = & \begin{cases} \ulcorner S \urcorner,\ \ulcorner \vec{t} \urcorner & \text{if } \mathsf{A} = \mathsf{I} \\ \ulcorner S \urcorner,\ \mathsf{M}_{\mathsf{A}}(\vec{t}) & \text{otherwise} \end{cases} \\ \ulcorner S, \mathsf{L}_l(\mathsf{A}, \vec{t}) \urcorner & = & \begin{cases} \ulcorner S \urcorner,\ \ulcorner \mathsf{A}, \vec{t} \urcorner & \text{if } \mathsf{A} = \mathsf{I} \\ \ulcorner S \urcorner,\ \mathsf{L}_l(\mathsf{A}, \vec{t}) & \text{otherwise} \end{cases} \end{array}\right. \qquad \text{where} \quad \left[\begin{array}{lcl} \ulcorner \cdot \urcorner & = & \cdot \\ \ulcorner t, \vec{t} \urcorner & = & I(t),\ \ulcorner \vec{t} \urcorner \end{array}\right.$$

The encoding of a signature Σ is obtained by including any part of the Dolev-Yao intruder signature Σ_{DY} that may be missing in Σ. More precisely, $\ulcorner \Sigma \urcorner$ is defined as $\Sigma_{DY} \cup (\Sigma \setminus (\Sigma \cap \Sigma_{DY}))$. The target signature Σ' of an execution judgment may contain role state predicate symbol declarations introduced by the execution of a (non Dolev-Yao) attacker role. We shall remove them from the translation, as indicated in the statement of Theorem 6.3.

While the above entities can be given an encoding based exclusively on their structure, this approach does not work smoothly for active role sets. Attacker rules are problematic: clearly, we want to map their operations to Dolev-Yao intruder roles, but the direct realization of this idea requires a wider context than what offered by a simply-minded recursive definition. For example, upon encountering a term $\{t\}_k$ in an incoming message, we may or may not need to use one of the shared-key roles **IS1** and **IS2** to look up k. Furthermore, it is not clear whether a copy of k is needed in other parts of the rule.

If we only consider entities that satisfy the typing and DAS restrictions, we can circumvent this difficulty by basing the encoding of an active role set R on a derivation $\mathcal{A}$ of the DAS judgment $\Sigma \Vdash R$, for a given signature Σ. Indeed, $\mathcal{A}$ would specify how the key k in the above example is accessed, and indirectly how many times it is needed in the rule it appears in. The translation of each DAS rule is given in Section 3 as a $\boxed{\text{boxed}}$ annotation next to the name of each rule. These annotations are either 1) a non-intruder active role ρ^{A}, 2) the name of a role in $\mathcal{P}_{DY}$, 3) "$\cdot$" if no role needs to be mapped to this rule, or finally 4) the abbreviations **DEL**(Δ) and **DUP**(Δ) which stand for as many copies of role **DEL** (resp. **DUP**) as there are elements in the knowledge context Δ appearing in this rule (see [6] for a formal definition).

Given a derivation $\mathcal{A}$ of $\Sigma \Vdash R$, we construct $\ulcorner \mathcal{A} \urcorner$ by collecting the active roles corresponding to the annotation of each rule that appears in $\mathcal{A}$. We define $\ulcorner R \urcorner$ as $\ulcorner \mathcal{A} \urcorner$. This definition entails that the encoding of any active role anchored on I consists exclusively of Dolev-Yao roles from $\mathcal{P}_{DY}$.

As an example, consider an active role consisting of the partially instantiated rule $\rho^{\mathsf{I}} = (\forall n_B : \mathsf{nonce}.\ \mathsf{L}(\mathsf{I}, \mathsf{B}, \mathsf{k_B}, \mathsf{n_I}), \mathsf{N}(\{\!\{\mathsf{n_I} n_B\}\!\}_{\mathsf{k_I}}) \rightarrow \mathsf{N}(\{\!\{n_B\}\!\}_{\mathsf{k_B}}))^{\mathsf{I}}$, taken from the specification of the Needham-Schroeder protocol in Appendix A. This rule is being executed by the intruder. Assuming an appropriate signature Σ, we have the following derivation for ρ^{I}, where we have reported the non-empty boxed annotations.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \Delta \Vdash^{a}_{A} \mathsf{k_I} \gg (\Delta, \mathsf{k_I'})}\,{\scriptstyle \mathbf{kas\_puu}}\,\boxed{\mathbf{IPV,DUP}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \Delta', n_B \Vdash_A \cdot \gg (\Delta', n_B)}\,{\scriptstyle \mathbf{tas\_dot}}}{\Gamma; \Delta' \Vdash_A n_B \gg \Delta''}\,{\scriptstyle \mathbf{tas\_ukn}}}{\Gamma; \Delta' \Vdash_A \mathsf{n_I}, n_B \gg \Delta''}\,{\scriptstyle \mathbf{tas\_kn}}\,\boxed{\mathbf{DEL}}}{\Gamma; \Delta' \Vdash_A (\mathsf{n_I}\, n_B) \gg \Delta''}\,{\scriptstyle \mathbf{tas\_cnc}}\,\boxed{\mathbf{DCM}}}{\Gamma; \Delta \Vdash_A \{\!\{\mathsf{n_I}\, n_B\}\!\}_{\mathsf{k_I}} \gg \Delta''}\,{\scriptstyle \mathbf{tas\_pke}}\,\boxed{\mathbf{PDC}}}{\Gamma; \Delta \Vdash_A \mathsf{N}(\{\!\{\mathsf{n_I}\, n_B\}\!\}_{\mathsf{k_I}}) > \cdot \gg \Delta''}\,{\scriptstyle \mathbf{las\_net,las\_dot}}\,\boxed{\mathbf{INT}}}{\Gamma; \cdot \Vdash_A \mathsf{L}(\mathsf{I}, \mathsf{B}, \mathsf{k_B}, \mathsf{n_I}), \mathsf{N}(\{\!\{\mathsf{n_I}\, n_B\}\!\}_{\mathsf{k_I}}) > \cdot \gg \Delta''}\,{\scriptstyle \mathbf{las\_rsp}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma; \Delta'' \looparrowright_A n_B}\,{\scriptstyle \mathbf{cas\_kn}}\,\boxed{\mathbf{DEL^6}} \quad \dfrac{}{\Gamma; \Delta'' \looparrowright_A \mathsf{k_B}}\,{\scriptstyle \mathbf{cas\_kn}}\,\boxed{\mathbf{DEL^6}}}{\Gamma; \Delta'' \looparrowright_A \{\!\{n_B\}\!\}_{\mathsf{k_B}}}\,{\scriptstyle \mathbf{cas\_pke}}\,\boxed{\mathbf{DUP^6, PEC}}}{\Gamma; \Delta'' \looparrowright_A \mathsf{N}(\{\!\{n_B\}\!\}_{\mathsf{k_B}})}\,{\scriptstyle \mathbf{ras\_net,ras\_dot}}\,\boxed{\mathbf{DUP^6, TRN, DEL^6}}}{\dfrac{(\Sigma, n_B : \mathsf{nonce}) \Vdash_A \mathsf{L}(\mathsf{I}, \mathsf{B}, \mathsf{k_B}, \mathsf{n_I}), \mathsf{N}(\{\!\{\mathsf{n_I}\, n_B\}\!\}_{\mathsf{k_I}}) \rightarrow \mathsf{N}(\{\!\{n_B\}\!\}_{\mathsf{k_B}})\,{\scriptstyle \mathbf{uas\_core}}}{\Sigma \Vdash_A \forall n_B : \mathsf{nonce}.\, \mathsf{L}(\mathsf{I}, \mathsf{B}, \mathsf{k_B}, \mathsf{n_I}), \mathsf{N}(\{\!\{\mathsf{n_I}\, n_B\}\!\}_{\mathsf{k_I}}) \rightarrow \mathsf{N}(\{\!\{n_B\}\!\}_{\mathsf{k_B}})\,{\scriptstyle \mathbf{aas\_ext,aas\_dot,uas\_all}}}}$$

where $\Gamma = (\Sigma, n_B : \mathsf{nonce})$, $\Delta = (\mathsf{I}, \mathsf{B}, \mathsf{k_B}, \mathsf{n_I})$, $\Delta' = (\Delta, \mathsf{k_I'})$ and $\Delta'' = (\Delta, n_B)$. The translation of ρ^{I} is the active role given by collecting the boxed Dolev-Yao intruder actions: $\ulcorner \rho^{\mathsf{I}} \urcorner = \mathbf{INT}, \mathbf{PDC}, \mathbf{IPV}, \mathbf{DCM}, \mathbf{TRN}, \mathbf{PEC}, \mathbf{DUP}^{13}, \mathbf{DEL}^{19}$. Observe that, the first six rules correspond to the operations needed to dismantle the message $\mathsf{N}(\{\!\{\mathsf{n_I}\, n_B\}\!\}_{\mathsf{k_I}})$ and construct $\mathsf{N}(\{\!\{n_B\}\!\}_{\mathsf{k_B}})$. Most of the duplication and deletion rules elide each other; the remaining six are used to get rid of the knowledge Δ'' since it is not memorized in any way in the consequent of ρ^{I}.

The family of translations $\ulcorner \_ \urcorner$ for our various objects preserves any entity not pertaining directly to the intruder. In particular, network messages, memory and role state predicates of other principals, and the roles that transform them are unaffected. It is easy to prove that $\ulcorner \_ \urcorner$ preserves typing and DAS [6].

Lemma 6.1 *Let Σ be a signature, $\mathcal{P}$ a protocol theory, S a state, and R an active role set.*
If $\vdash \Sigma$, $\Sigma \vdash \mathcal{P}$, $\Sigma \vdash S$, $\Sigma \vdash R$, $\Sigma \Vdash \mathcal{P}$ *and* $\Sigma \Vdash R$,
then $\vdash \ulcorner\Sigma\urcorner$, $\ulcorner\Sigma\urcorner \vdash \ulcorner\mathcal{P}\urcorner$, $\ulcorner\Sigma\urcorner \vdash \ulcorner S\urcorner$, $\ulcorner\Sigma\urcorner \vdash \ulcorner R\urcorner$, $\ulcorner\Sigma\urcorner \Vdash \ulcorner\mathcal{P}\urcorner$ *and* $\ulcorner\Sigma\urcorner \Vdash \ulcorner R\urcorner$.

Theorem 6.3 below states that the Dolev-Yao intruder is the most powerful attacker, in the sense that it can emulate the deeds of any other attacker. A proof of this result relies on a number of lemmas that describe how the translation of derivations for each of the DAS judgments from Section 3 is mapped to an execution sequence. Due to space limitations, we shall refer the interested reader to [6] for a presentation of these auxiliary results and of their proofs. We give a flavor of the elegant proof technique underlying our Theorem 6.3 by displaying the statement of one of these lemmas, which shows how the Dolev-Yao intruder can emulate the access to the information appearing in terms in the left-hand side of a rule. The representation $\ulcorner\Delta\urcorner$ of a knowledge context is defined as for term tuples (see [6] for a formal definition). We write “$\mathcal{A} :: J$” to indicate that $\mathcal{A}$ is a derivation of the judgment J.

Lemma 6.2 *Let Σ be a signature, $\vec{t}$ a term tuple, and Δ and Δ' knowledge contexts compatible with Σ.*
If $\mathcal{A} :: \Sigma; \Delta \Vdash_{\mathsf{I}} \vec{t} \gg \Delta'$, *then* $\cdot \rhd [\ulcorner\Delta\urcorner, \ulcorner\vec{t}\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\ulcorner\Sigma\urcorner} \longrightarrow^* [\ulcorner\Delta'\urcorner]^{\cdot}_{\ulcorner\Sigma\urcorner}$ *is derivable.*

This result is proved by induction on the structure of the given DAS derivation $\mathcal{A}$ [6]. We have the following statement for the main result in this section.

Theorem 6.3 (*The Dolev-Yao Intruder is the Most Powerful Attacker*)
Let $\mathcal{P}$ be a protocol theory, S and S' two states, R and R' two active role sets, Σ and Σ' signatures such that

$$\vdash \Sigma \qquad \Sigma \vdash \mathcal{P} \qquad \Sigma \vdash S \qquad \Sigma \vdash R \qquad \Sigma \Vdash \mathcal{P} \qquad \mathcal{A} :: \Sigma \Vdash R \qquad \mathcal{A}' :: \Sigma, \Sigma' \Vdash R'$$

If $\quad \mathcal{E} :: \mathcal{P} \rhd [S]^{R}_{\Sigma} \longrightarrow^* [S']^{R'}_{\Sigma,\Sigma'}$, $\quad$ *then* $\quad (\ulcorner\mathcal{P}\urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\ulcorner\Sigma\urcorner} \longrightarrow^* [\ulcorner S'\urcorner]^{\ulcorner\mathcal{A}'\urcorner}_{\ulcorner\Sigma\urcorner,\Sigma^*}$ *is derivable.*

where Σ^* is a subsignature of Σ' such that $\Sigma' = \Sigma^*, \Sigma_{\mathsf{L}}$ and Σ_{L} consists only of role state predicate symbol declarations.

Proof: This proof is constructive and proceeds by induction on the structure of $\mathcal{E}$. Due to space constraints, we refer the reader to [6] for the technical development and instead give the intuition behind the emulation of the most interesting execution rules.

Our emulation does not interfere with actions that involve non-intruder roles. Installing a role ρ^{I} anchored on I into the current active role set (rule $\mathbf{ex\_arole}$) is emulated by copying as many instances of objects from $\mathcal{P}_{DY}$ as specified by the encoding of ρ^{I}. Intruder-instantiated generic roles (rule $\mathbf{ex\_grole}$) are treated in the same way, which means that our emulation does not allow I to directly execute a generic role. Uses of rule $\mathbf{ex\_all}$ to instantiate a universal variable in an active intruder rule do not correspond to any action: we have proved that DAS is preserved under substitution [8] and that this process does not affect the encoding of a DAS derivation [6]. Finally, the application of a fully instantiated intruder rule ($\mathbf{ex\_core}$) relies on results such as lemma 6.2 above that specify the behavior of its constituents. □

Since, in models that relies on black-box cryptography, an attack of any kind is ultimately an execution sequence between two configurations, this theorem states that a security protocol has an attack if and only if it has a Dolev-Yao attack. This justifies the design of tools that rely on the Dolev-Yao intruder [4, 14, 18, 20, 22, 24, 25], but it does not mean that considering other specifications of the attacker is pointless. Indeed, precisely because of its generality, a straight adoption of the Dolev-Yao intruder often results in inefficient verification procedures. Overhead can be greatly relieved by relying on general optimizations that cut the search space [7, 13, 19, 24] and on per-protocol specializations, for example allowing the intruder to construct only message patterns actually used in the protocol [20, 22]. Finally, the environment in which a particular protocol is deployed may be so constraining that a weaker attacker model is sufficient to ensure the desired security goals.

Our result extends to settings that involve multiple intruders $\mathsf{I}_1, \ldots \mathsf{I}_n$. We process each of these attackers independently as specified above, obtaining n copies of $\mathcal{P}_{DY}$, each anchored on a particular I_i. We then make use of the attack-preservation result in [26] to reduce them to a single attacker I.

7 Conclusions and Future Work

In this paper, we have presented a data access specification system for the security protocol specification framework *MSR* [8, 9, 11] and used it to show that the Dolev-Yao intruder model embedded in most crypto-protocol verification tools [4, 14, 18, 20, 22, 24, 25] is indeed the most powerful attacker. In the near future, we intend to further investigate the relations between DAS and the Dolev-Yao intruder. While it appears that a specification of this attacker can be automatically constructed from the DAS rules [10], it is not yet clear whether a DAS policy can always be derived from an attacker specification. In order to answer this question, we are constructing an extended library of case studies [8, 6, 9] that require different DAS assumptions. Another important question that we intend to tackle using *MSR* is whether it is possible to derive sensible DAS rules (and a most powerful intruder model) from the specification of a protocol (including its term language) rather than by imposing them from above [10]. On a more practical side, we want to address the issues of type-reconstruction [9] and deterministic variable instantiation in order to develop a usable security protocol verification system based on *MSR*.

Acknowledgments

We are grateful to Catherine Meadows, Paul Syverson and Grit Denker for the insightful discussions on the topics presented in this paper. We would also like to thank Frank Pfenning and Andre Scedrov for their interest and encouragement.

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] S. Brackin. Automatically detecting most vulnerabilities in cryptographic protocols. In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition — DISCEX'00*, volume 1, pages pp. 222–236, Hilton Head, SC, 2000. IEEE Computer Society Press.

[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 specifications of security protocols. Unpublished manuscript. Accessible as `http://www.cs.stanford.edu/~iliano/TMP/msr.ps.gz`.

[7] I. Cervesato. Typed multiset rewriting specifications of security protocols. In A. Seda, editor, *Proceedings of the First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology — MFCSIT'00*, Cork, Ireland, 19–21 July 2000. Elsevier ENTCS.

[8] I. Cervesato. A specification language for crypto-protocol based on multiset rewriting, dependent types and subsorting. In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Workshop on Specification, Analysis and Validation for Emerging Technologies — SAVE'01*, Paphos, Cyprus, 2001.

[9] 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, 2001. Springer-Verlag LNCS 2052.

[10] I. Cervesato. The wolf within. In J. Guttman, editor, *Second Workshop on Issues in the Theory of Security — WITS'02*, Portland, OR, 2002.

[11] 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.

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

[13] G. Denker, J. 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.

[14] 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.

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

[16] 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.

[17] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[18] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[19] W. Marrero, E. M. Clarke, and S. 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] C. Meadows. The NRL protocol analyzer: an overview. *J. Logic Programming*, 26(2):113–131, 1996.

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

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

[23] 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.

[24] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *Proceedings of the 11th Computer Security Foundations Workshop*, pages 106–115, Rockport, MA, 1998. IEEE Computer Society Press.

[25] D. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Proceedings of the Twelth IEEE Computer Security Foundations Workshop*, pages 192–202, Mordano, Italy, June 1999. IEEE Computer Society Press.

[26] P. Syverson, C. Meadows, and I. 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.

[27] P. F. Syverson. A different look at secure distributed computation. In *Tenth IEEE Computer Security Foundations Workshop — CSFW-10*, pages 109–115. IEEE Computer Society Press, June 1997.

A Example

In this appendix, we show an actual *MSR* specification by reprinting from [9] the simple protocol theory that describes the two-party nucleus of the Needham-Schroeder public-key authentication protocol [21]. We choose this example for its conciseness and the fact that most reader will be familiar with it. More complex (and interesting) specifications can be found in the same paper and in [8].

The server-less variant of the Needham-Schroeder public-key protocol [21] is a two-party crypto-protocol aimed at authenticating the initiator A to the responder B (but not necessarily vice versa). It is expressed as the expected run on the right in the "usual notation" (where we have used our syntax for messages). In the first line, the initiator A encrypts a message consisting of a nonce n_A and her own identity with the public key k_B of the responder B, and sends it (ideally to B). The second line describes the action that B undertakes upon receiving and interpreting this message: he creates a nonce n_B, combines it with A's nonce n_A, encrypts the outcome with A's public key k_A, and sends the resulting message out. Upon receiving this message in the third line, A accesses n_B and sends it back encrypted with k_B. The run is completed when B receives this message and successfully recognizes n_B.

1. $A \to B$: $\{\!\{n_A\ A\}\!\}_{k_B}$
2. $B \to A$: $\{\!\{n_A\ n_B\}\!\}_{k_A}$
3. $A \to B$: $\{\!\{n_B\}\!\}_{k_B}$

MSR, like most modern security protocol specification languages, represents roles, *i.e.* the sequence of actions executed by each individual principal. We 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 abbreviation.

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

$$\left(\begin{array}{llcll} \multicolumn{5}{l}{\quad \exists L : \mathsf{principal} \times \mathsf{principal}^{(B)} \times \mathsf{pubK}\ B \times \mathsf{nonce}.} \\ \begin{array}{l} \forall B : \mathsf{principal}. \\ \forall k_B : \mathsf{pubK}\ B. \end{array} & \cdot & \to & \exists n_A : \mathsf{nonce}. & \begin{array}{l} \mathsf{N}(\{\!\{n_A\ A\}\!\}_{k_B}) \\ L(A, B, k_B, n_A) \end{array} \\ \\ \begin{array}{l} \forall \ldots \\ \forall k_A : \mathsf{pubK}\ A. \\ \forall k'_A : \mathsf{privK}\ k_A. \\ \forall n_A, n_B : \mathsf{nonce}. \end{array} & \begin{array}{l} \mathsf{N}(\{\!\{n_A\ n_B\}\!\}_{k_A}) \\ L(A, B, k_B, n_A) \end{array} & \to & & \mathsf{N}(\{\!\{n_B\}\!\}_{k_B}) \end{array}\right)^{\forall A}$$

Clearly, any principal can engage in this protocol as an initiator (or a responder). Our encoding is therefore structured as a generic role. Let A be its postulated owner. The first rule formalizes line (1) of the "usual notation" description of this protocol from A's point of view. It has an empty antecedent since initiation is unconditional in this protocol fragment. Its right-hand side uses an existential quantifier to mark the nonce n_A as fresh. The consequent contains the transmitted message and the role state predicate $L(A, B, k_B, n_A)$, necessary to enable the second rule of this protocol. The arguments of this predicate record variables used in the second rule.

The second rule encodes lines (2–3) of the "usual notation" description. It is applicable only if the initiator has executed the first rule (enforced by the presence of the role state predicate) and she receives a message of the appropriate form. Its consequent sends the last message of the protocol.

MSR assigns a specific type to each variable appearing in these rules. The equivalent "usual notation" specification relies instead on natural language and conventions to convey this same information, with clear potential for ambiguity. We shall mention that most declarations can be automatically reconstructed [9]: this simplifies the task of the author of the 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.

The responder is encoded as the generic role below, whose owner we have mnemonically called B. The first rule of this role collapses the two topmost lines of the "usual notation" specification of this protocol fragment from the receiver's point of view. The second rule captures the reception and successful interpretation of the last message in the protocol by B: this step is often overlooked. This rule has an empty consequent.

$$\left(\begin{array}{llcll} \multicolumn{5}{l}{\quad\exists L : \mathsf{principal} \times \mathsf{nonce}.} \\ \begin{array}{l}\forall k_B : \mathsf{pubK}\ B. \\ \forall k'_B : \mathsf{privK}\ k_B. \\ \forall A : \mathsf{principal}. \\ \forall n_A : \mathsf{nonce}. \\ \forall k_A : \mathsf{pubK}\ A\end{array} & \mathsf{N}(\{\!\{n_A\ A\}\!\}_{k_B}) & \rightarrow & \exists n_B : \mathsf{nonce}. & \begin{array}{l}\mathsf{N}(\{\!\{n_A\ n_B\}\!\}_{k_A}) \\ L(B, n_B)\end{array} \\ \begin{array}{l}\forall \ldots \\ \forall n_B : \mathsf{nonce}.\end{array} & \begin{array}{l}\mathsf{N}(\{\!\{n_B\}\!\}_{k_B}) \\ L(B, n_B)\end{array} & \rightarrow & & . \end{array}\right)^{\forall B}$$

Again, most typing information can be reconstructed from the way variables are used.

B Typing Judgments and Rules

B.1 Typing Judgments

$\tau :: \tau'$	*τ is a subsort of τ'*	$\Sigma \vdash P$	*P is a valid message predicate in signature Σ*
$\Sigma \vdash t : \tau$	*Term t has type τ in signature Σ*	$\Sigma \vdash S$	*S is a valid state in signature Σ*
$\Sigma \vdash \tau$	*τ is a valid type in Σ*	$\Gamma \Vdash^{r} rhs$	*rhs is a valid rule consequent in typing context Γ*
$\vdash \Sigma$	*Σ is a valid signatures*	$\Gamma \vdash r$	*r is a valid rule in typing context Γ*
$\Vdash^{c} \Gamma$	*Γ is a valid typing context*	$\Gamma \vdash \rho$	*ρ is a valid rule collection in typing context Γ*
$\Sigma \vdash \vec{t} : \vec{\tau}$	*Term tuple $\vec{t}$ has type $\vec{\tau}$ in signature Σ*	$\Sigma \vdash \mathcal{P}$	*$\mathcal{P}$ is a valid protocol theory in signature Σ*
$\Gamma \vdash \vec{\tau}$	*$\vec{\tau}$ is a valid type tuple in typing context Γ*	$\Sigma \vdash R$	*R is a valid active role set in signature Σ*

B.2 Typing Rules

$\tau :: \tau'$	*τ is a subsort of τ'*

$$\frac{}{\mathsf{principal} :: \mathsf{msg}}\,\textbf{ss\_pr} \qquad \frac{}{\mathsf{nonce} :: \mathsf{msg}}\,\textbf{ss\_nnc}$$

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

$\Sigma \vdash t : \tau \qquad \Gamma \vdash t : \tau$	*Term t has type τ in signature Σ (viz. context Γ)*

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

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

$$\frac{\Sigma \vdash t : \tau' \quad \tau' :: \tau}{\Sigma \vdash t : \tau}\,\textbf{mtp\_ss} \qquad \frac{}{(\Sigma, a : \tau, \Sigma') \vdash a : \tau}\,\textbf{mtp\_a}$$

$\Sigma \vdash \tau \qquad \Gamma \vdash \tau$	*τ is a valid type in signature Σ (viz. context Γ)*

$$\frac{}{\Sigma \vdash \mathsf{principal}}\,\textbf{ttp\_pr} \qquad \frac{}{\Sigma \vdash \mathsf{nonce}}\,\textbf{ttp\_nnc} \qquad \frac{}{\Sigma \vdash \mathsf{msg}}\,\textbf{ttp\_msg}$$

$$\frac{\Sigma \vdash A : \mathsf{principal} \quad \Sigma \vdash B : \mathsf{principal}}{\Sigma \vdash \mathsf{shK}\ A\ B}\,\textbf{ttp\_shK} \qquad \frac{\Sigma \vdash A : \mathsf{principal}}{\Sigma \vdash \mathsf{pubK}\ A}\,\textbf{ttp\_pbK} \qquad \frac{\Sigma \vdash k : \mathsf{pubK}\ A}{\Sigma \vdash \mathsf{privK}\ k}\,\textbf{ttp\_pvK}$$

$\vdash \Sigma$	*Σ is a valid signatures*

$$\frac{}{\vdash \cdot}\,\textbf{itp\_dot} \qquad \frac{\Sigma \vdash \tau \quad \vdash \Sigma}{\vdash \Sigma, a : \tau}\,\textbf{itp\_a} \qquad \frac{\Sigma \vdash \mathsf{principal}^{(A)} \times \vec{\tau} \quad \vdash \Sigma}{\vdash \Sigma, \mathsf{L}_l : \mathsf{principal}^{(A)} \times \vec{\tau}}\,\textbf{itp\_rsp} \qquad \frac{\Sigma \vdash \mathsf{principal}^{(A)} \times \vec{\tau} \quad \vdash \Sigma}{\vdash \Sigma, \mathsf{M}\_ : \mathsf{principal}^{(A)} \times \vec{\tau}}\,\textbf{itp\_mem}$$

$\models^{c}\ \Gamma$

$$\frac{\vdash\ \Sigma}{\models^{c}\ \Sigma}\,\textbf{ctp\_sig} \qquad \frac{\Gamma\ \vdash\ \tau \quad \models^{c}\ \Gamma}{\models^{c}\ \Gamma, x : \tau}\,\textbf{ctp\_x} \qquad \frac{\Gamma\ \vdash\ \mathsf{principal}^{(A)} \times \vec{\tau} \quad \models^{c}\ \Gamma}{\models^{c}\ \Gamma, L : \mathsf{principal}^{(A)} \times \vec{\tau}}\,\textbf{ctp\_rsp}$$

$\Sigma\ \vdash\ \vec{t} : \vec{\tau} \qquad \Gamma\ \vdash\ \vec{t} : \vec{\tau}$ *Term tuple $\vec{t}$ has type $\vec{\tau}$ in signature Σ (viz. context Γ)*

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

$\Gamma\ \vdash\ \vec{\tau}$ *$\vec{\tau}$ is a valid type tuple in typing context Γ*

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

$\Sigma\ \vdash\ P \qquad \Gamma\ \vdash\ P$ *P is a valid message predicate in signature Σ (viz. context Γ)*

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

$\Sigma\ \vdash\ S \qquad \Gamma\ \vdash\ lhs$ *S (viz. lhs) is a valid state (viz. predicate sequence) in signature Σ (viz. context Γ)*

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

$\Gamma\ \models^{r}\ rhs$ *rhs is a valid rule consequent in typing context Γ*

$$\frac{\Gamma\ \vdash\ \tau \quad (\Gamma, x : \tau)\ \models^{r}\ rhs}{\Gamma\ \models^{r}\ \exists x : \tau.\, rhs}\,\textbf{rtp\_nnc} \qquad \frac{\Gamma\ \vdash\ lhs}{\Gamma\ \models^{r}\ lhs}\,\textbf{rtp\_seq}$$

$\Gamma\ \vdash\ r$ *r is a valid rule in typing context Γ*

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

$\Gamma\ \vdash\ \rho$ *ρ is a valid rule collection in typing context Γ*

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

$\Sigma\ \vdash\ \mathcal{P}$ *$\mathcal{P}$ is a valid protocol theory in signature Σ*

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

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

$\Sigma\ \vdash\ R$ *R is a valid active role set in signature Σ*

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

C Other Omitted Rules

C.1 Omitted Access Control Rules

$\Delta > \vec{e} > \Delta'$ *Merging context knowledge Δ and term tuple $\vec{e}$ yields Δ'*

$$\frac{}{\Delta > \cdot > \Delta}\,\textbf{mas\_dot}\ \boxed{\cdot} \qquad \frac{\Delta > \vec{e} > \Delta'}{\Delta > e, \vec{e} > (\Delta', e)}\,\textbf{mas\_ukn}^{\sharp}\ \boxed{\cdot} \qquad \frac{\Delta > \vec{e} > \Delta'}{(\Delta, e) > e, \vec{e} > (\Delta', e)}\,\textbf{mas\_kn}^{\sharp}\ \boxed{\textbf{DEL}}$$

$\Gamma; \Delta \looparrowright_A t$ *Given knowledge Δ, principal A can construct term t*

$$\frac{\Gamma; \Delta \looparrowright_A t_1 \quad \Gamma; \Delta \looparrowright_A t_2}{\Gamma; \Delta \looparrowright_A t_1\, t_2}\,\textbf{cas\_cnc}\ \boxed{\textbf{DUP}(\Delta), \textbf{CMP}}$$

$$\frac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A k}{\Gamma; \Delta \looparrowright_A \{t\}_k}\,\textbf{cas\_ske}\ \boxed{\textbf{DUP}(\Delta), \textbf{SEC}} \qquad \frac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A k}{\Gamma; \Delta \looparrowright_A \{\!\{t\}\!\}_k}\,\textbf{cas\_pke}\ \boxed{\textbf{DUP}(\Delta), \textbf{PEC}}$$

$\Gamma; \Delta \looparrowright_A \vec{t}$ *Given knowledge Δ, principal A can construct term tuple $\vec{t}$*

$$\frac{}{\Gamma; \Delta \looparrowright_A \cdot}\,\textbf{cas\_dot}\ \boxed{\textbf{DEL}(\Delta)} \qquad \frac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A \vec{t}}{\Gamma; \Delta \looparrowright_A (t, \vec{t})}\,\textbf{cas\_ext}\ \boxed{\textbf{DUP}(\Delta)}$$

C.2 Omitted Execution Rules

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

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

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

$$\frac{}{\mathcal{P} \triangleright C \longrightarrow^* C}\,\textbf{ex\_it0} \qquad \frac{\mathcal{P} \triangleright C \longrightarrow C' \quad \mathcal{P} \triangleright C' \longrightarrow^* C''}{\mathcal{P} \triangleright C \longrightarrow^* C''}\,\textbf{ex\_itn}$$