Typed Multiset Rewriting Specifications of Security Protocols

Iliano Cervesato[*]

October 2011
CMU-CS-11-140
CMU-CS-QTR-108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Carnegie Mellon University, Qatar campus.
The author can be reached at iliano@cmu.edu.

Abstract

When it comes to security protocol analysis, the devil lies in the detail . . . or more precisely in how they are expressed. This report collects the formal definition of *MSR 2.0*, an unambiguous, flexible, powerful and relatively simple specification framework for cryptographic security protocols. *MSR 2.0* is a strongly typed multiset rewriting language over first-order atomic formulas. It uses existential quantifiers to model the generation of fresh data and memory predicates to encode systems consisting of a collection of coordinated subprotocols. Its typing infrastructure, based on the theory of dependent types with subsorting, supports type-checking and data access validation, a static check that is shown to be intimately connected to the Dolev-Yao model. We show that *MSR 2.0*'s dynamic semantics preserves typing and data access validy. We also provide a formal proof that the Dolev-Yao intruder can emulate any attacker that can be specified within the Dolev-Yao model of security, a fundamental assumption of many verification tools that however had never been proved before. We conclude with three specifications of increasing sophistication, starting with the classic Needham-Schroeder public-key authentication protocol and ending with the complex manipulations involved in the *OFT* group key management protocol.

Partially supported by QNRF under grant NPRP 09-667-1-100 and the Qatar Foundation under grant number 930107. This work was started while the author was working at the Naval Research Laboratory under contract N00173-00-C-2086.

Keywords: Specification of Security Protocol, Dolev-Yao Model, Parallel Multiset Rewriting, Dependent Types, Subsorting, Data Access Specification, Dolev-Yao Intruder, Strongest Attacker.

Contents

1 Introduction

While early work on comprehending cryptographic protocols goes back to the late 1970s and early 1980s [52, 44], the systematic study of these security artifacts emerged as a recognizable subfield within computer science in the mid-1990, with the widespread adoption of the Internet. Since then, there have been significant advancements both in the foundational understanding of cryptographic protocol security [46], and in the development of practical tools that can verify not just the toy protocols of the 1990s but commercial protocols [14], and not only convenient abstractions as embodied by the Dolev-Yao model [44], but the bitstring transformations performed by actual cryptographic primitives in what is now known as the computational model [2, 5, 9, 15, 40].

Up to the late 1990s, protocols were described in English by and large, which often led to ambiguities. Understandably, their verification was in its infancy. Towards the very end of that decade, several groups of researchers [1, 27, 47] started proposing formal languages in which to describe a protocol unambiguously, a necessary basis for robust verifications. This report focuses on one of these efforts, which adopts a form of first-order multiset rewriting to describe cryptographic protocols and developed it into an analysis methodology. This idea has been formalized in a family of languages generally referred to as *MSR*. This report is about *MSR 2.0*, which was used as the medium of a multi-year verification effort of the Kerberos 5 authentication protocol [14, 35, 4], as a formal language where to investigate foundational notions of security [21], and as an intermediate formalism to relate formalizations in other languages [30, 26].

In *MSR*, the observation that protocol transitions may be naturally expressed as a form of rewriting is sharpened to a rigorous, formal definition of the Dolev-Yao model by means of multiset rewriting with existential quantification [27, 46]. In this framework protocol execution may be carried out symbolically. Existential quantification, as commonly used in formal logic, provides a natural way of choosing new values, such as new keys or nonces. Multiset rewriting provides a very precise way of specifying security protocols and has been incorporated into various high-level specification languages for authentication protocols, for example CAPSL [43]. The multiset rewriting formalism allows us to formulate one standard intruder theory that describes any adversary for any protocol.

MSR 2.0 was built on top of a first version of *MSR*, which we will call *MSR 1.0* and that was largely designed at Stanford University in 1998–99. *MSR 1.0* consisted of an elementary fragment of first-order multiset rewriting with existential quantifiers. It was developed to study foundational questions about security protocols, whose answers were unknown back then. The first such result was that establishing whether a protocol maintained secrecy is undecidable in general [27, 45, 46]. It was then used to investigates specifications written in other protocol specification languages, namely strand spaces [28, 29, 30] and process algebra [8, 7, 26]. More abstractly, connections with linear logic were also made [31].

The main purpose of this report is to act as a centralized and largely complete references to *MSR 2.0* in the context of cryptographic protocol specification, as well as to collect results and proofs that were never published before. *MSR 2.0* was developed in 2000–02 at the Naval Research Laboratory as a usable language to write actual protocol specifications and verify them. It was extended with a flexible type system based on dependent types and aspects of its syntax were relaxed. This was mainly a response to the fragility of *MSR 1.0* as a language for specifying concrete protocols [30]. Bits and pieces of *MSR 2.0* have been presented in various venues, for example [16, 17, 19]. A formal specification of this language in a context that goes beyond security protocols can be found in [24]. This slightly more general language has been implemented in Maude [39, 38].

MSR 2.0 was meant to work with real-world protocols. Our first and longest case study involved the Kerberos 5 repeated authentication protocol. We formalized major aspects of this protocol at a level of detail that had not been attempted before and developed an accompanying verification methodology to analyze it. We found the the core protocol behaves according to its specifications, although some optional components displayed innocuous anomalies [12, 11, 20, 13, 14]. We also examined its support for cross-realm authentication, a rarely used feature, and found that, although it achieves stated security goals, these goals are so weak as to make them of little utility in practice [32]. We then turned our attention to PKINIT, an optional subprotocol of Kerberos based on public-key exchanges. Our analysis revealed a serious vulnerability in its specification [33, 34, 35] which led to an immediate revision of the Kerberos 5 standards. We also relied on this protocol in [25, 4] where we used *MSR 2.0* to support security analysis in the computational model [5]. Although it was designed for concrete specifications, *MSR 2.0* proved a useful formalism where to carry out foundational investigations on the nature of the Dolev-Yao intruder [21, 18] and also to explore ideas about quantitative notions of security analysis [23].

More recently, we have designed a third version of this language, *MSR 3.0* [22, 36, 37], which has much deeper roots in logic, type theory, and concurrency theory. Although mostly compatible with *MSR 2.0*, it simplifies and modularizes specifications, and also bridges the traditional divide between state-based and process-based specifications of concurrent systems, as respectively found in Petri nets and process algebra for example.

This report is organized as follows. We define *MSR*'s terms and messages in Section 2, its state constituents and their typing infrastructure in Section 3, and rules and protocol theories in Section 4. Section 5 defines the notion of data access specification and shows that it is decidable. In Section 6, we present the execution semantics of *MSR* and prove that it preserves both typing and data access specification validity, while some pragmatic implementation issues are discussed in Section 7. In Section 8, we encode the standard Dolev-Yao intruder in our language and prove that this specification captures any other attacker we may define in *MSR*. We give the *MSR* specification of three examples of increasing complexity in Section 9 and conclude in Section 10.

2 Typed Messages

In Section 2.1, we describe the messages, or more generally terms, that form the core of *MSR*. Then in Section 2.2, we introduce the typing infrastructure that will allow us to make sense of these terms, followed by the typing rules that defines their validity in Section 2.3. We proceed in Section 2.4 by slightly enlarging the definition of terms and types to admit variables. We will need them later when type-checking protocol rules. We conclude in Section 2.5 by pointing out the major differences with respect to the messages used in previous work on *MSR* [27, 30].

2.1 Messages

Messages are obtained by applying a number of message forming constructs, discussed below, to a variety of *atomic messages*. The atomic messages we will consider in this report are principal identifiers, keys, nonces, and raw data (i.e. pieces of data that have no other function in a protocol than to be transmitted). We formalize our notion of atomic message by means of the following grammatical productions:

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

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

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

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

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

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

2.2 Types

While types played a very modest role in the original definition of *MSR* [27, 30], they stand at the core of the extension presented in this report. Through typing, we will not only enforce basic well-formedness conditions (e.g. that only keys be used for encrypting a message), but types will provide a statically checkable way to ascertain complex desiderata such as, for example, that no principal may grab a key he/she is not entitled to access. The central role of types in our present approach is witnessed by the fact that they subsume and integrally replace the "persistent information" of the original *MSR* [30].

The typing machinery that best fits our goals is based on the type-theoretic notion of *dependent product types with subsorting*. Rather than delving into the depth of the definitions and properties of this formalism, we will introduce only the facets that we will use, and only to the extent we will need them. In particular, we will not conduct an in-depth discussion of this type theory; we will even stay away from the most exotic aspects of its syntax. Readers who wish to further their understanding of this formalism are invited to consult [41, 48, 3, 55].

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

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

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

We will use the type msg to classify generic messages. Clearly raw data will have type msg. This is however not sufficient since nonces, keys, and principal identifiers are routinely part of messages. We solve this problem by imposing a *subsorting* relation between types. We formalize this relation by means of the following *judgment*:

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

In this report, the subsorting relation will amount to having each of the types discussed above with the exception of private keys be a subtype of msg. Its extension is expressed by means of the following rules:

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

$$\frac{}{\mathsf{shK}\ A\ B :: \mathsf{msg}}\ \mathbf{ss\_shK} \qquad\qquad \frac{}{\mathsf{pubK}\ A :: \mathsf{msg}}\ \mathbf{ss\_pbK}$$

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

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

2.3 Typing

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

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

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

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

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

2.3.1 Typing Messages

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

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

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

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

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

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

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

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

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

$$\frac{}{(\Sigma, a : \tau, \Sigma') \vdash a : \tau}\ \mathbf{mtp\_a}$$

This concludes the presentation of the typing rules for messages. For the convenience of the reader, we collect all the rules presented in this report in Appendix A.

It is easy to see that the well-typedness of a message is a decidable property, assuming that the subsorting relation is acyclic and without infinitely descending chains. This fact is captured by the following property.

Property 2.1 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\Sigma \vdash t : \tau$ holds.*

Proof: Observe that the premises of all rules except **tp_ss** invoke the typing judgment on subterms of the message appearing in the rule conclusion, if at all. Since signatures are finite, the unbound meta-variables in the premises can be instantiated only in a finite number of ways. Rule **mtp_ss** does not change the message, but checks it on a subtype. Since the subsorting relation is a well-order, this rule can be applied only a finite number of times before breaking up the message using another rule. □

2.3.2 Validating Types

The dependence of types on terms can make a syntactically correct type meaningless. For example, "privK k" is not valid if k has type "shK A B". We check the validity of a type by means of the following judgment.

$$\Sigma \vdash \tau \qquad\qquad \textit{τ is a valid type in signature Σ}$$

Non-dependent types are valid in every signature:

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

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

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

Were we to have types other than the ones considered in this document, we would need to extend this rule set with additional rules to establish their validity.

Since we have shown that type-checking terms is decidable, this property extends straightforwardly to types.

Property 2.2 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\Sigma \vdash \tau$ holds.*

Proof: The rules implementing this judgment are syntax-directed with respect to the structure of types. The premises, when present, invoke the typing judgment that we know is decidable given the assumptions of this property. □

2.3.3 Validating Signatures

Since signatures contain declarations that attribute a type to an atomic message (they will be extended in Section 3), we must concern ourselves with their validity. We express this notion by means of the judgment:

$$\vdash \Sigma \qquad\qquad \textit{Σ is a valid signatures}$$

The rules that validate this judgment are given below: an empty signature is trivially valid, a non-empty signature $\Sigma, a : \tau$ is valid if the type of its last declaration is valid in Σ, and the tail Σ is itself a valid signature.

$$\frac{}{\vdash \cdot}\,\textbf{itp\_dot} \qquad \frac{\Sigma \vdash \tau \quad \vdash \Sigma}{\vdash \Sigma, a : \tau}\,\textbf{itp\_a} \qquad (\ldots)$$

Since we will extend the notion of signature, we will need to provide rules validating each new form of declaration. This is indicated by the ellipsis on the right.

We postpone the simple decidability statement till Section 3 where we will complete the description of signatures.

2.4 Parametric Messages

Tuple types (see Section 3) and protocol rules (see Section 4) rely on objects that are similar to messages except that they may contain variables to be instantiated during type-checking and execution, respectively. We will now extend the definitions given above to allow for this possibility. This will result in the concepts of parametric terms, parametric types, and typing rules that can handle them.

2.4.1 Messages

A parametric message allows variables where terms could appear in the messages of Section 2.1. The definition is updated as follows:

$$\begin{array}{lrll} \textit{Parametric messages:} \quad t & ::= & a & \textit{(Atomic messages)} \\ & \mid & x & \textit{(Variables)} \qquad \Leftarrow \\ & \mid & t_1\, t_2 & \textit{(Concatenation)} \\ & \mid & \{t\}_k & \textit{(Symmetric-key encryption)} \\ & \mid & \{\!\{t\}\!\}_k & \textit{(Asymmetric-key encryption)} \end{array}$$

where the arrow on the right indicates the added production.

When writing judgments, it will be convenient to blur the distinction between atomic constants and variables since they are generally treated in the same manner. Indeed, variables can been seen as temporary constants whose lifespan is limited to a type fragment, a rule or a role. With this in mind, we will write A (or B), k, and n, variously decorated, for atomic constants or variables that are principals, keys, and nonces respectively. Whenever the object we want to refer to cannot be but a constant (mostly in the execution rules), we will use the corresponding serifed letters: A (or B), k, and n. Finally, the letters x, y and z will stand for terms that must be variable.

In some circumstances, we will need to refer to objects that can be either variables or atomic message constants, but not composite terms. We call these terms *elementary* and denote them with the letter e, variously decorated.

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

2.4.2 Types and Contexts

The definition of types does not change except that the embedded terms can now be (possibly contain) variables. Therefore, there is no need to update the grammar presented in Section 2.2.

We reserve the name signature for a sequence of ground declarations. Whenever variables are involved, we will instead talk about *typing contexts*. A typing context, denoted Γ is a signature with a (finite) number of variable declarations appended to it:

$$\begin{array}{lrll} \textit{Typing contexts:} \quad \Gamma & ::= & \Sigma & \textit{(Plain signature)} \\ & \mid & \Gamma, x : \tau & \textit{(Extension with a variable declaration)} \\ & \mid & (\ldots) & \end{array}$$

As for signatures, we will later extend this definition. It should be clear that the type τ of a declaration $x : \tau$ in a typing context can itself depend on variables. Similarly to signatures, we require that variables be declared exactly once in a context.

2.4.3 Typing

The typing rules for parametric messages change little with respect to their ground counterparts: the main difference is that, similarly to the way (ground) atomic terms were treated, variables need to be validated in the typing context. These rules are as follows:

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

$$\frac{\Gamma \vdash t : \mathsf{msg} \quad \Gamma \vdash k : \mathsf{shK}\ A\ B}{\Gamma \vdash \{t\}_k : \mathsf{msg}}\ \mathbf{mtp'\_ske} \qquad \frac{\Gamma \vdash t : \mathsf{msg} \quad \Gamma \vdash k : \mathsf{pubK}\ A}{\Gamma \vdash \{\!\{t\}\!\}_k : \mathsf{msg}}\ \mathbf{mtp'\_pke}$$

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

Once we interpret variables as temporary constants, the distinction between typing contexts and signatures vanishes, and so does the distinction between rules $\mathbf{mtp'\_a}$ and $\mathbf{mtp'\_x}$. Indeed, the type systems for ground and parametric terms become isomorphic. We will observe this phenomenon over and over. Rather than duplicating inference rules in this manner (and doubling the size of this report), we decided to sacrifice in precision and present the same rules for typing both ground and parametric messages (and related concepts). Readers bothered by this approach are invited to fill this omission by themselves: it is a matter copying each rule xxx_yyy to an identical rule xxx'_yyy and replace the context meta-variables Γ with signature meta-variables Σ, or vice versa.

We apply this principle right away to the rules for checking the validity of type. We simply inherit them from Section 2.3.2.

The following judgment is used to validate a typing context:

$$\models^{c} \Gamma \qquad\qquad \Gamma \textit{ is a valid typing context}$$

The rules that implement it are given below. We type-check degenerate contexts by means of the similar judgment for signatures. If a context has a typing declaration $x : \tau$ for a variable as its last declaration, we validate τ as well as the prefix Γ. The ellipsis indicates that we will consider additional rules to validate extensions to the definition of context.

$$\frac{\vdash \Sigma}{\models^{c} \Sigma}\ \mathbf{ctp\_sig} \qquad \frac{\Gamma \vdash \tau \quad \models^{c} \Gamma}{\models^{c} \Gamma, x : \tau}\ \mathbf{ctp\_x} \qquad (\ldots)$$

Given our interpretation of variables as temporary constants, it is clear that our decidability results extend to the parametric case. We abstain from repeating them here. Direct proofs proceed as in the ground cases.

2.5 Changes

We will now compare the above term infrastructure with the notion of message used in earlier versions of *MSR* [27, 30]. The main differences concern the types and the way they are used.

1. Structurally, the original presentation of *MSR* referred to so-called *simple types*, i.e. types that do not depend on terms. These types would permit distinguishing objects into keys, nonces, messages, etc. (the taxonomy was open-ended), but would not allow any finer classification [30]. Subsorting introduced some flexibility. Most of the information that is now captured through dependencies was then expressed as “persistent predicates” that cluttered protocol specifications and complicated reasoning about them. These entities have been dropped.

2. Typing played a rather inessential role in [27, 30]: constants were given types, but rules themselves did not carry such information. Furthermore, no type system was explicitly presented to validate terms or rules. Typing is central to the language proposed in this report. Every object is typed and most of this article is devoted to type-checking and its properties. It will be clear as the discussion develops that our approach allows not only verifying basic well-formedness conditions (e.g. that no message is encrypted with a nonce), but types will provide a statically checkable way to enforce complex requirements such as, for example, that no principal uses a key he/she is not entitled to access.

3. In this document, we make a syntactic distinction between shared-key and public-key encryption. Although not present in [27, 30], we could have easily distinguished the two forms of encryption in the earlier version of *MSR*.

3 Message Predicates and States

States are a fundamental concept in *MSR*. Indeed, they are the central constituent of the snapshots of a protocol execution. They are the objects transformed by rewrite rules to simulate message exchange and information update. Finally, together with execution traces, they are the hypothetical scenarios on which protocol analysis is based.

In this section, we will formalize the concept of state in *MSR*, together with the constructions needed to implement it. In Section 3.1 we introduce message tuples and typing rules for them. Then in Section 3.2, we discuss the message predicates that appear in a state, and in Section 3.3 we define states. We conclude in Section 3.4 with the introduction of parametric variants of these various concepts.

3.1 Message Tuples and Dependent Types

As we will see shortly, message predicates are atomic first-order formulas with zero or more terms as their arguments. In this section, we are concerned with formalizing the concept of tuple and presenting suitable typing rules for them. We introduced term tuples in Section 3.1.1 and present the corresponding notion of dependent tuple types in Section 3.1.2. We define the notion of term substitution in Section 3.1.3 and use it in the typing rules for tuples in Section 3.1.4.

3.1.1 Message Tuples

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

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

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

3.1.2 Dependent Tuple Types

It is tempting to define the type of a tuple as the sequence of the types of its components. Therefore, if A is a principal name and $\mathsf{k_A}$ is a public key for A, the tuple $(\mathsf{A}, \mathsf{k_A})$ would have type "$\mathsf{principal} \times \mathsf{pubK\ A}$" (the *Cartesian product* symbol "$\times$" is the standard constructor for tuple types). This construction allows us to associate a generic principal with A's public key: if B is another principal, then $(\mathsf{B}, \mathsf{k_A})$ will have this type as well. We will often need stricter associations, such as between a principal and its *own* public key. In order to achieve this, we will rely on the notion of *dependent tuple type*. In this example, the tuple $(\mathsf{A}, \mathsf{k_A})$ will be attributed type "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$", where the variable A in "$\mathsf{principal}^{(A)}$" records the name of the principal at hands and forces the type of the key to be "$\mathsf{pubK}\ A$" for this particular A: therefore $(\mathsf{A}, \mathsf{k_A})$ is valid, but $(\mathsf{B}, \mathsf{k_A})$ is now ill-typed since $\mathsf{k_A}$ has type "$\mathsf{pubK\ A}$" rather than the expected "$\mathsf{pubK\ B}$".[1]

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

$$\begin{array}{llll} \textit{Tuple types} & \vec{\tau} & ::= & \cdot & \textit{(Empty tuple)} \\ & & | & \tau^{(x)} \times \vec{\tau} & \textit{(Tuple type extension)} \end{array}$$

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

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

3.1.3 Substitutions

In the example above, verifying that the tuple $(\mathsf{A}, \mathsf{k_A})$ has type "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$" proceeds as follows: we first check that A is indeed a principal, but before continuing with the validation of the right component of this tuple, we must replace

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

every occurrence of the variable A with the principal name A. Therefore, type-checking a tuple type in general relies on the operation of ***substitution*** of a term t for a variable x in a tuple type $\vec{\tau}$, denoted $[t/x]\vec{\tau}$. Since tuple types are built on types, which in turn can contain terms, we shall define two more substitution operations: the replacement of t for x in a type τ and a term t', denoted $[t/x]\tau$ and $[t/x]t'$ respectively. For simplicity, we overload the bracket notation to denote the application of the substitution operation to objects belonging to different syntactic categories. These operations are defined in the following table:

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

In the second line of the leftmost column, we implicitly assume that the bound variable y is different from x and from any variable possibly contained in t (this would result in a *variable capture*: a free variable in t would become bound once the substitution has been completed). We take care of these situations by implicitly renaming the bound variable (y in this case) to a variable name (z say) that does not appear in t or $\vec{\tau}$. This is known as *α-conversion* in type-theoretic circles. Intuitively, if we think of variables as place holders, binders dictate which place holders must be instantiated with the same term. Variable names implement this association. Besides a possibly mnemonic function, the names of variables themselves have little importance and can be replaced with arbitrary strings as long as no conflict with other variables is introduced.

3.1.4 Typing

Type-checking a message tuple with respect to a dependent tuple type reduces to verifying that each component message has the type in the corresponding position. This is formalized by the following judgment:

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

The rules implementing the typing judgment for tuples are given below: the type of empty message tuple is the empty tuple type, otherwise, components must match after performing the appropriate substitutions to satisfy possible dependencies.

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

Observe that all the judgments in these rules mention a signature rather than a typing context. In particular, the variable x exposed in the rightmost premise of rule **mtp_ext** is immediately substituted with the term t, which is proved ground in the left premise of this rule.

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

Property 3.1 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\Sigma \vdash \vec{t} : \vec{\tau}$ holds.*

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

We now extend the notion of validity of a type to tuples of types. We rely on the following judgment to expresses this relation:

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

The corresponding typing rules simply check that every component of a tuple type is a valid type. Possible dependencies of later types on earlier components are resolved by assuming that the binding variable has the given type. Observe that, consequently, this typing judgment is assessed relative to a typing context rather than a signature.

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

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

3.2 Message Predicates

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

- First, the predicate $\mathsf{N}(\_)$ implements the contents of the *public network* in a distributed fashion: for each (ground) message t currently in transit, the state will contain a component of the form $\mathsf{N}(t)$.
- Second, active roles rely on a number of *role state predicates*, one for each rule in them, of the form $\mathsf{L}_l(\_, \ldots, \_)$, where l is a unique identifying label. The arguments of this predicate record the value of the 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.

The reader familiar with our previous work on *MSR* will have noticed a number of differences with respect to the definitions given in [27, 30]. Memory predicates are indeed new. They are intended to model situations that need to maintain data private across role executions: for example, this allows a principal to remember its Kerberos ticket, or the trusted-third-party of a fair exchange protocol to avoid fraudulent recoveries from aborted transactions. Another difference with respect to earlier work is the absence of a dedicated predicate retaining the intruder's knowledge. This can however be easily implemented using memory predicates.

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

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

Given this extension to signatures, we can define typing rules to validate message predicates. This relation is captured by the following judgment:

$$\Sigma \vdash P \qquad\qquad \textit{P is a valid message predicate in signature } \Sigma$$

The rules implementing this judgment are given below. The argument of a network predicate must be a valid message. The type of the arguments of role state and memory predicates must correspond to the declared type for this predicate. We regard the subscript A in a memory predicate $\mathsf{M}_A(\vec{t})$ as if it were the first argument for typing purposes.

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

$$\frac{(\Sigma, \mathsf{M}\_ : \vec{\tau}, \Sigma') \vdash (A, \vec{t}) : \vec{\tau}}{(\Sigma, \mathsf{M}\_ : \vec{\tau}, \Sigma') \vdash \mathsf{M}_A(\vec{t})}\,\textbf{ptp\_mem}$$

We now extend the rules that validate signatures to take role state and memory predicate declarations into consideration. We require that the first argument of a role state and memory predicate be a principal, this will be the owner of the role.

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

The dots on the left of these rules represent the rules for signature validation given in Section 2.3.3. A complete set of rules is given in Appendix A.

We postpone discussing the decidability of type-checking message predicates till states are introduced. However, we are now in a position to verify the decidability of the signature validation judgment. We have the following property:

Property 3.2 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\vdash \Sigma$ holds.*

Proof: Since signatures are finite, this result reduces to the decidability of validating types and tuple types, which we have proved in Sections 2.3.2 and 3.1.4. □

3.3 States

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

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

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

The validity of a state reduces to checking that each component predicate is well-formed in the current signature. We formalize this by means of the judgment

$$\Sigma \vdash S \qquad\qquad S \textit{ is a valid state in signature } \Sigma$$

which is implemented by the following two rules:

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

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

Property 3.3 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\Sigma \vdash S$ holds.*

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

3.4 Parametric Message Predicates

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

For the most part, this reduces to admitting variables in embedded terms. However, role state predicates need to be created on the spot in order to avoid interferences. We achieve this by introducing variables, denoted L, that are instantiated to actual role state predicates during application. This makes our language weakly second-order, although we could easily reduce it to the first order by interpreting a role state predicate L_l as the symbol L indexed by a label l that is kept as a variable in rules. We however opt for the more direct solution. With this insight, we can now present the complete definition of a typing context.

$$\begin{array}{llcll} \textit{Typing context} & \Gamma & ::= & \Sigma & \textit{(Plain signature)} \\ & & | & \Gamma, x : \tau & \textit{(Extension with a variable declaration)} \\ & & | & \Gamma, L : \vec{\tau} & \textit{(Extension with a role state predicate declaration)} \end{array}$$

The typing rule for role state predicate variables is constructed as for their non-parametric counterpart in signatures. Consequently, the typing rules for contexts are completed as follows:

$$(\ldots) \qquad \frac{\Gamma \vdash \mathsf{principal}^{(A)} \times \vec{\tau} \qquad \models^{\!c} \Gamma}{\models^{\!c} \Gamma, L : \mathsf{principal}^{(A)} \times \vec{\tau}}\,\mathbf{ctp\_rsp}$$

3.5 Changes

In this final part of the current section, we will analyze the main differences between the notion of state and related concepts defined above and the analogous entities from the original definition of *MSR* [27, 30].

Typing constitutes a minor distinction at this level since both versions of *MSR* rely on typed predicate names, although little use of this aspect was made in [27, 30]. Our present use of dependent tuple types is mostly a consequence of our adoption of dependent types to classify terms. The major differences regard instead the introduction of memory predicates and the elimination of persistent information and distinguished intruder knowledge predicates in states.

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

2. There is no trace in the above definitions of the persistent information that formed the immutable portion of the state of *MSR* in [27, 30]. The functionality of those predicates is now performed by the strong typing policy of our updated framework, and in particular by our reliance on dependent types.

3. Finally, we should point out the absence of a dedicated predicate intended to hold the intruder's knowledge. This aspect of our earlier work can now be realized transparently through memory predicates.

4 Multiset Rewriting Theories

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. A role can therefore be seen as a reactive system.

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

This section defines rules, roles and protocol theories, and provides typing rules for them. More specifically, in Section 4.1, we introduce rules and their constituents. Roles are defined in Section 4.2. Then, in Section 4.3, we turn our attention to protocol theories. Finally, in Section 4.4 we anticipate the dynamic notion of an active role and present typing rules for these entities. Protocol theories and roles will be further discussed in Sections 5 and 6 which respectively focus on data access specification as an additional syntactic check and on execution. Section 4.5 examines how these definitions differ from our original presentation of *MSR*.

4.1 Rules

With a slight imprecision that will be corrected as the discussion proceeds, a rule has the form "$lhs \rightarrow rhs$". Rules are the basic mechanism that enables the transformation of a state into another, and therefore the simulation of protocol execution: whenever the antecedent "lhs" matches part of the current state, this portion may be substituted with the consequent "rhs" (after some processing).

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

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

The universal quantifiers used in rules *bind* the variables they are applied to. Variables that are not bound by any quantifier are said to be *free*. Free variables can occur in the construction of a rule, but roles themselves should have all their variables bound (this is enforced by the typing rules): they are said to be *closed*. The *scope* of a binder is the expression over which its binding action spans. The scope of all the binders in the above productions spans over a whole rule.

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

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

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

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

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

The notion of fresh and bound variable discussed earlier applies also here. Again, right-hand sides are considered equal up to α-conversion of their bound variables. Notice that the scope of these quantifiers is limited to the right-hand side of the current rule. Later rules can refer to the values created by these variables by introducing universal quantifiers of the proper type: synchronization is ensured by their occurrence in the role state predicates.

We can now present the typing rules for rules and their components. We shall first turn our attention to rule consequents, to which we attribute the following typing judgment (the superscript "*r*" is intended to distinguish this judgment from the analogous relation for states):

$$\Gamma \Vdash^{r} rhs \qquad\qquad \textit{rhs is a valid rule consequent in typing context } \Gamma$$

Since right-hand sides are usually deeply embedded in rules, their validity must in general be checked relative to a typing context rather than a simple signature. This context will include the declaration of all the previously introduced variables. We have the following typing rules:

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

Rule **rtp_nnc** validates consequents that start with a fresh datum declaration. The left premise verifies that the type τ of the variable x is valid. This is necessary since otherwise invalid types may not be caught. The right premise extends the current typing context with the declaration for x and attempts to validate the rest of the consequent. The addition of $x : \tau$ to the context will allow using rule **mtp_a** (see Section 2.3.1) to check every occurrence of x in ρ. We required that no variable in a typing context or signature be declared more than once. Implicit α-conversion provides a simple way to implement this constraint: if a variable named x is already contained in Γ, we implicitly choose an unused symbol, use it to rename every occurrence of x in $\exists x : \tau.\, \mathit{rhs}$, and seamlessly apply rule **rtp_nnc** to this expression.

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

Given the above typing rules, the following judgment validates rules themselves:

$$\Gamma \vdash r \qquad\qquad r \textit{ is a valid rule in typing context } \Gamma$$

It is implemented by the following two rules:

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

Rule **utp_core** reduces rule validation to type-checking its antecedent as a parametric state and verifying its right-hand side as described above. The quantifier in rule **utp_all** is treated as in the case of **rtp_nnc** and should not require further discussion.

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

Property 4.1 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\Gamma \vdash r$ holds.*

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

4.2 Roles

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

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

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

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

The following judgment has the function to validate a rule collection:

$$\Gamma \vdash \rho \qquad\qquad \textit{ρ is a valid rule collection in typing context } \Gamma$$

The following three rules that realize it:

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

The quantifier in rule **otp_rsp** is treated as in the case of **rtp_nnc**: the left premise validates the tuple type $\vec{\tau}$ in the current context Γ, while the other premise analyzes ρ after introducing the role state predicate parameter L in Γ. The techniques for handling bound variables we just discussed apply also here. Rule **otp_rule** applies when a rule collection starts with a rule r.

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

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

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

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

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

We will present the type-checking rules for roles after the notion of protocol theory has been formalized in Section 4.3

4.3 Protocol Theories

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

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

It should be observed that we do not make any special provision for the intruder. The adversary is expressed as one or more roles in the same way as the more legitimate message exchange in a protocol. We will illustrate in Section 9 how this is achieved for the standard Dolev-Yao intruder.

The validity of a protocol theory is checked against the current signature. This is an indication that roles containing free variables will not type-check, and therefore are not to be considered valid. The following judgment expresses this relation:

$$\Sigma \vdash \mathcal{P} \qquad\qquad \textit{$\mathcal{P}$ is a valid protocol theory in signature } \Sigma$$

The rules implementing this judgment are given below. They essentially reduce the validity of a protocol theory to the validation of the rule collections within roles, expressed by the judgment $\Gamma \vdash \rho$.

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

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

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

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

Property 4.2 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgments $\Gamma \vdash \rho$ and $\Sigma \vdash \mathcal{P}$ hold.*

Proof: A role is a finite collection of rules with a distinguished constant or variable (the role owner). Type-checking a role reduces then to validating its rules, which is decidable.

The second judgment reduces to the first since a protocol theory consists of finitely many roles. □

4.4 Active Roles

As we will see in Section 6, 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. The following grammar captures their macroscopic structure:

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

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

Typing-checking active role sets is achieved by means of the following judgment:

$$\Sigma \vdash R \qquad\qquad \textit{R is a valid active role set in signature } \Sigma$$

and implemented by the following two typing rules:

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

Active role sets are type-checked with respect to signatures, and therefore they must be ground in order to be valid. In rule **atp_ext**, we first verify that the role owner A is an actual principal declared in the signature. Then we check in the right-hand premise that ρ is a valid rule collection. It should be observed that this will proceed slightly differently than if we were checking an anchored role, although the calls look very alike: indeed ρ will in general be more instantiated than an anchored role and possibly have declared constants for role state predicate symbols. Since the typing rules that verify the antecedent and the consequent rely ultimately on the state validation judgment, this is processed transparently.

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

Property 4.3 *Whenever the subsorting relation $\tau :: \tau'$ is a well-order, it is decidable whether the judgment $\Sigma \vdash R$ holds.*

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

4.5 Changes

It should be noted that the above syntax of rules is much more liberal than the original definition of *MSR* [27, 30]. We dedicate the remainder of this section to commenting on these differences. We begin by noting structural changes (items 1–3), then move to persistent information (item 4), memory predicates (item 5), and network predicates (item 6). We then continue with a number of remarks concerning the role state predicates (items 7–10), and conclude with active roles (item 11).

1. A protocol theory was defined in [27, 30] as a collection of rules for which the graph generated by their role state predicates was acyclic (see item 10 below for further details). Each connected component corresponded to a role. Besides this constraint, the rules within a role were independent. This aspect is mostly maintained in our current formulation, but our definition of role allow threading rules by using role state predicate declarations as a sequencing device. This option is seldom used since this effect can be achieved more simply by appropriately using role state predicates.

2. In [27, 30], all variables were implicitly universally quantified at the head of a rule, unless marked in the consequent as fresh data. Here, these variables are explicitly introduced by typed universal quantifiers. Explicit quantification simplifies rule analysis as far as type-checking and data access specification are concerned. More importantly, it declares the type of variables, which is necessary for the correct execution of a protocol (more on this aspect in 6). In Section 7.1, we will see how many quantifiers and typing declarations can be omitted while writing protocol specifications, and be reconstructed during type-checking.

3. The only quantifiers present in [27, 30] appeared in rule consequents and had the purpose of introducing variables to be instantiated with fresh data. The adoption of dependent types makes this mechanism more powerful. Our current schema allows for example a server to generate a key to be shared among two specific principals.

4. As already observed, our current formulation does not make use of the persistent predicates of [27, 30]. As we said, the information once conveyed by these objects is not entirely captured within the type system of *MSR*.

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

6. Any number of network predicates can appear in both the antecedent and the consequent of a rule. The original definition allowed at most once such predicate per rule, either in the left-hand side (in "receive" rules), or in the right-hand side (for "send" rules). Denker et al. [42] showed that, whenever some simple conditions are met, a sequence of contiguous message reception rules followed by a sequence of contiguous transmission rules can be soundly collapsed into a single rule of the proposed form. This is clearly a conservative extension of our previous proposal. Its practical value is in reducing the length of the specification of a protocol.

7. In [27, 30], each rule contained exactly one role state predicate on each side, with the exception of the single "role generation rule" that activated the role. The productions above instead allow zero, one, or several such predicates on either side. This definition simplifies the grammatical specification of roles without complicating the typing and data access specification rules. However, we will make a very limited use of this added expressive power. Having more than one role state predicate in the left-hand or right-hand side of a rule does not appear particularly useful since, as we will see in Section 5, the arguments of these predicates collect all the data known to a principal in the current thread of execution of a role. The antecedent of the first rule in a role cannot sensibly mention any role state predicate since its left-hand side could not match any state. The antecedent of the other rules typically will have such a predicate for synchronization, although this is not enforced. For the same reason, the consequent of a rule will generally contain a role state predicate. The natural exception to this practice regards the final rules of a role, which do not need to make provisions for further synchronization.

8. The role state predicate symbols allowed in [27, 30] were constants rather than variables. We now prefer to make them unique to each instantiation of a rule in order to avoid the possibility of interferences (although we have no examples in which such a phenomenon is harmful).

9. With the exception of [30], our previous work did not refer to any argument of a role state predicate as a distinguished role owner. The objectives of these early versions of *MSR* did not actively rely on role owners, and therefore no particular emphasis was given to this notion. This information becomes crucial when trying to enforce data access specification as we will do in Section 5.

10. For any given role, it is interesting to study the graph whose nodes are its role state predicates and whose edges link the predicate occurring in the left-hand side to the predicate in the right-hand side of each rule. In [27, 30, 31] we required that this graph be an upper semi-lattice, and in particular that it had a single entry point and be acyclic. This was a crucial assumption for studying the complexity of protocols [27, 46]. Here, we drop this condition from the syntax of a role, but our firing rules will de facto implement a similar constraint at execution time.

11. Our previous work [27, 30] did not have a notion of active role: since role state predicate symbols were constants, rules could not be threaded within a role and each rule was necessarily an independent object that could be applied whenever its antecedent matched the current state. The more complex pattern proposed in this chapter require memorizing which roles are in current use, and how they are instantiated. This is the purpose of active role sets.

5 Data Access Specification

Type-checking allows statically verifying that language expressions are constructed in accordance to the intended functionality of the objects that constitute them. For example, it can be used to catch such errors as the encryption of a message with a tuple, or even with a nonce. It applies to all aspects of the encoding of a protocol, from state components, to rules, to types themselves. The underlying idea is that expressions that violate the typing policy cannot be part of the specification of a valid protocol: they are therefore meaningless.

We will now use the typing declarations of *MSR* to further restrict the spectrum of sensible expressions, in particular as far as protocol rules are concerned. Indeed, 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, the owner of a role should clearly be able to access his/her own memory predicates, but not those of any other party.

In this section, we will formalize and implement these various requirements (and others) by means of statically checkable *data access specification* judgments. Data access verification and type-checking are independent procedures, nonetheless it will be convenient to assume that all the expressions we will be analyzing are well-typed, since it simplifies the discussion. It should be observed that this way of organizing our presentation does not entail that validating a protocol specification requires two passes: both typing (actually type reconstruction, see Section 7.1) and data access specification can take place simultaneously. The resulting judgments and combined inference rules would become however more complex and we leave their write-up to the interested reader.

In Sections 2–4, we incrementally introduced the syntax and typing rules of *MSR* starting from the most basic entities (atomic messages) and building on our own work as we presented more complex constructions (ultimately protocol theories). Now that the syntax has been defined, we believe that we can attain better clarity by moving in the inverse direction: in the following, we will initially present the data access specification judgments and inference rules for macroscopic objects such as protocol theories and roles, but only later describe how data access specification is enforced on their components. Therefore, the premises of inference rules will sometimes mention a judgment that has not yet been formally defined. We will indicate such occurrences by enclosing them in a gray box. In all such cases, we will give ample warnings and enough explanations in the text to permit understanding these rules. Given this disclaimer, the remainder of our discussion on data access specification is organized as follows: in Section 5.1 we discuss data access specification for protocol theories, roles, and individual rules. In Section 5.2, we concentrate on the left-hand side of a rule, where most information is gathered. Section 5.3 focuses instead on rule consequents. In Section 5.4, we slightly extend the discussion to enforce data access specification on active roles. In Section 5.5, we show that checking data access specification is decidable. We conclude in Section 5.6 by listing the changes with respect to earlier versions of *MSR*.

5.1 Protocol Theories and Roles

In this section, we present the highest (and simplest) levels of data access specification: we describe how protocol theories, roles and rules are processed, leaving the treatment of the antecedent and consequent of a rule for Sections 5.2 and 5.3, respectively. We conclude by defining the notion of knowledge context, which underlies our data access specification verification procedure.

The following judgment expresses the fact that a protocol theory $\mathcal{P}$ realizes correct data access specification with respect to a signature Σ.

$\Sigma \Vdash \mathcal{P}$ *Protocol theory $\mathcal{P}$ implements valid data access specification in signature Σ*

This judgment is implemented by the following three inference rules, corresponding to the three productions in the syntax of a protocol theory. The right-hand premise of rules **hac_grole** and **hac_arole** invoke the data access specification judgment "$\Gamma \Vdash_A \rho$" for rule collections, that will be introduced shortly.

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

Rule **hac_dot** trivially validates the empty protocol theory. Observe that 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 **hac_arole** deals with anchored roles. It may appear surprising that we do not check that A is declared in its signature as a principal. Remember however that we are working under the assumption that all expressions are well-typed, therefore we know by rule **htp_arole** from Section 4.3 that Σ includes a declaration of the form "A : principal".

Since data access specification 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 following data access specification judgment for rule collections themselves, where A is the owner of ρ:

$\Gamma \Vdash_A \rho$ *Rule collection ρ implements valid data access specification for principal A in context Γ*

The inference rules implementing this judgment are given below. Again, their one-to-one correspondence with the grammatical productions that define a rule collection makes them syntax-oriented.

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

The leftmost rule deals with empty rule collections. Rule **oac_rsp** collects the declaration of an existentially quantified role in the context and verifies its body. Again, since we operate on well-typed objects, we do not need to check the validity of the tuple type $\vec{\tau}$. The rightmost inference rule, **oac_rule**, implements the situation where a collection starts with a rule r. Its left premise validates r itself by means of the data access specification judgment for rules, "$\Gamma \Vdash_A r$", that we will describe shortly. The rest ρ of the rule collection is recursively verified in the right premise

The following judgment expresses valid data access specification for rules:

$\Gamma \Vdash_A r$ *Rule r implements valid data access specification for principal A in context Γ*

It is implemented by the two rules given below. We will dedicate most of the sequel to describing how this is achieved.

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

Intuitively, the judgment in the left premise of rule **uac_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 5.2, where we will explain the meaning of the various "$\cdot$"s.

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 5.3.

Rule **uac_all** collects the universal declaration prefixing the core of a rule in the context.

We conclude this section by introducing the notion of *knowledge context*, often simply referred to as *knowledge*. These objects play an important role in data access specification verification for the antecedent and consequent of a rule. The knowledge context of a rule collects all the information that its owner knows. As we will see, this information comes from role state predicates, data stored in memory predicates, and messages received from the network. It can also contain data deduced from all of the above by means of simple inferences.

Well-constructed rules remember the data they have accessed, i.e. their knowledge, in the arguments of their role state predicates. Since these predicates can only be applied to variables or constants (in a rule), a knowledge context shall be a collection of elementary terms. We will extend this notion in Section 5.4 when describing our data access specification mechanism for active roles.

$$\begin{array}{llll} \textit{Knowledge contexts:} \quad \Delta & ::= & \cdot & \textit{(Empty knowledge context)} \\ & \mid & \Delta, a & \textit{(Extension with atomic knowledge)} \\ & \mid & \Delta, x & \textit{(Extension with parametric knowledge)} \\ & \mid & \textit{(\ldots)} & \end{array}$$

It will be convenient to see knowledge contexts as multisets. Therefore, we may write $\Delta = (\Delta', e)$ to denote that the variable or constant e is in Δ, even if it is not the last element of Δ.

A knowledge context Δ is compatible with a signature Σ if for each term t in Δ, there is a type τ such that $\Sigma \vdash \tau$ and $\Sigma \vdash t : \tau$.

As anticipated, we will dedicate the next section to the treatment of the left-hand side of rules, and then focus on their right-hand side in Section 5.3.

5.2 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 presence of variables makes this process parametric. At execution time, pattern matching will instantiate the variables in the antecedent, and carry the resulting substitution to the right-hand side to produce meaningful output messages. We will describe execution in more detail in Section 6.

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). No matter what they are or are intended to be instantiated with, each of the e_i's is an elementary piece of information within the rule. For example, even if e_3 is a variable and it is clear from the protocol at hand that it can only be substituted with a term of the form $\{y\}_k$, the variable x_3 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 expectation of a message t from the network is expressed by a predicate $\mathsf{N}(t)$ in the left-hand side of r. The term 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 from an interlocutor), and will be represented by variables (or constants) 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. Each of these variables represents data that A is accessing, and that he/she will use in the rule's consequent. The goal of data access specification 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. Again, we must ascertain that A can legitimately access this information.

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; how this is done is explained in Section 5.2.1. The second step involves breaking the composite messages gathered in this way into their elementary components (variables and constants). This is illustrated in Section 5.2.2. The most sensitive aspect of this phase is making sure that A has access to the keys needed to decipher encrypted messages. We isolate this critical check in Section 5.2.3.

5.2.1 Collecting Arguments

In this section, we present judgments and inference rules aimed at collecting the arguments of the predicates in the left-hand side of a rule. This information will be further processed in Section 5.2.2. We will rely on the following fairly complex judgment:

$\Gamma; \Delta \Vdash_A \mathit{lhs} > \vec{t} \gg \Delta'$ *Given knowledge Δ, predicate sequence lhs and terms $\vec{t}$, principal A can knows Δ' in context Γ*

The various 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 context* Δ lists the collected arguments that are known to be elementary. Finally, the *output knowledge context* Δ' stands for the elementary information that will ultimately be extracted from r's left-hand side (i.e. all the variables and atomic constants that appear in it). It is convenient to interpret this judgment operationally as a partial function that given A, Γ, lhs and Δ computes a value for Δ' if the data access specification policy is obeyed. It will be convenient to interpret $\vec{t}$ as a multiset.

As we start processing the antecedent of a rule, Δ and $\vec{t}$ are empty (written "$\cdot$") as no argument has yet been collected. This explains the left premise of rule **uac_core** in Section 5.1.

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 (recall also that we assume to be working with well-typed objects). 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 "$\Delta > \vec{e} > \Delta'$" which we will explain shortly:

$$\frac{\Delta > (A, \vec{e}) > \Delta' \quad \Gamma; \Delta' \Vdash_A \mathit{lhs} > \vec{t}' \gg \Delta''}{\Gamma; \Delta \Vdash_A (L(A, \vec{e}), \mathit{lhs}) > \vec{t}' \gg \Delta''}\ \textbf{lac\_rsp}$$

This rule applies only if the first parameter of the role state predicate symbol L is the owner of the rule under examination.

We next turn to network predicates in the antecedent of a rule. Since the received message t may not be elementary, we shall include it in the list of unprocessed arguments $\vec{t}'$ before examining the remaining predicates lhs. We have the following rule:

$$\frac{\Gamma; \Delta \Vdash_A \mathit{lhs} > (t, \vec{t}') \gg \Delta''}{\Gamma; \Delta \Vdash_A (\mathsf{N}(t), \mathit{lhs}) > \vec{t}' \gg \Delta''}\ \textbf{lac\_net}$$

Memory predicates are handled similarly. We shall however make sure that the owner A of the currently examined rule does not try to access information gathered by another principal. Therefore, only memory predicates of the form $\mathsf{M}_A(\vec{t})$ will be processed.

$$\frac{\Gamma; \Delta \Vdash_A \mathit{lhs} > (\vec{t}, \vec{t}') \gg \Delta'}{\Gamma; \Delta \Vdash_A (\mathsf{M}_A(\vec{t}), \mathit{lhs}) > \vec{t}' \gg \Delta'}\ \textbf{lac\_mem}$$

In this rule, we are slightly abusing our notation by interpreting the tuple $\vec{t}$ as a multiset to be merged with $\vec{t}'$ (the comma “,” is indeed a different constructor for these two entities). This can easily be corrected by defining a judgment and two inference rules that take each term in $\vec{t}$ in turn and adds it to $\vec{t}'$. For the sake of simplicity, we leave formalizing this correction to the strict readers.

Since predicate sequences are multisets, several instances of the above rules will in general be applicable to a non-empty lhs. It should be observed that the order in which they are applied is irrelevant to the success of this judgment and to the calculation of the output knowledge Δ'.

Once the arguments of all the predicates on the left-hand side of the rule have been collected, we pass to the second phase which ascertains that the uninterpreted arguments $\vec{t}$ satisfy the data access specification policy. This is done in the following rule by invoking the judgment “$\Gamma; \Delta \Vdash_A \vec{t} \gg \Delta'$” which will be discussed in Section 5.2.2.

$$\frac{\Gamma; \Delta \Vdash_A \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A \cdot > \vec{t} \gg \Delta'}\ \textbf{lac\_dot}$$

It may appear surprising that we insist in collecting the arguments of all the message predicates in the left-hand side of a rule before attempting to break any composite term. The alternative, eagerly examining arguments, is however incomplete in general. Assume for example we receive the following two network messages: $\mathsf{N}(k_1\ \{m_1\}_{k_2})$ and $\mathsf{N}(k_2\ \{m_2\}_{k_1})$. By first collecting their arguments and then decomposing them, we can clearly access the inner terms m_1 and m_2. However, if we require that the argument of one predicate is fully processed before examining the other, then neither of these messages can be exposed.

We conclude this section by defining the following merge judgment used in rule **lac_rsp**:

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

This judgment is implemented by the following three simple rules:

$$\frac{}{\Delta > \cdot > \Delta}\ \textbf{mac\_dot} \qquad \frac{\Delta > \vec{e} > \Delta'}{\Delta > e, \vec{e} > (\Delta', e)}\ \textbf{mac\_ukn} \qquad \frac{\Delta > \vec{e} > \Delta'}{(\Delta, e) > e, \vec{e} > (\Delta', e)}\ \textbf{mac\_kn}$$

The rightmost rule recognizes that the elementary term e is already present in the current knowledge context and therefore does not add it anew. It should be observed that in this situation, rule **mac_ukn** is applicable as well and will result in the duplication of e in the output knowledge context. Well-written protocol specifications will seldom require such redundancy, however.

It should be observed that the knowledge context Δ this judgment is initially invoked with is empty (“$\cdot$”) unless the antecedent of a rule contains more than one role state predicate (as mentioned earlier, we do not anticipate any situation where this would prove useful). Therefore, we do not expect a significant usage of rule **mac_kn**.

5.2.2 Certifying Composite Arguments

The previous section has left open the task of verifying that the elementary information embedded in the composite messages collected from the left-hand side of a rule can actually be accessed by its owner. In this section, we will show how this is achieved up to the verification of keys, which is the topic of Section 5.2.3.

We will rely on the following judgment to examine possibly composite terms. We used it in rule **lac_dot** in the previous section.

$\Gamma; \Delta \Vdash_A \vec{t} \gg \Delta'$	*Given knowledge Δ and terms $\vec{t}$, principal A can knows Δ' in context Γ*

The interpretation of each meta-variable in this judgment is inherited from the argument collection judgment in the previous section: A is the rule owner, Γ is the typing context, Δ is the input knowledge, $\vec{t}$ represents the terms that still have to be verified for data access specification, and Δ' is the output knowledge context extracted from $\vec{t}$ and Δ. Again, this judgment can be seen as a partial function that computes a value for Δ' when given A, Γ, Δ and $\vec{t}$, assuming that $\vec{t}$ implements correct data access specification for A. It should be observed that A has legitimate access to each term in $\vec{t}$: we want to verify that this property extends to their subterms. Again, we interpret $\vec{t}$ as a multiset.

Our first two rules deal with unchecked elementary messages. Thus the collection of terms still to be validated have the form $(e, \vec{t})$, where e is either an atomic constant or a variable. There are two possibility: either e is known and therefore appears in the current input knowledge, or it must be looked up in the typing context Γ. In the first case, implemented by rule **tac_kn**, we simply continue with the evaluation of $\vec{t}$. In the second case (rule **tac_ukn**), we shall first add e to the input knowledge context used to validate $\vec{t}$.

$$\frac{\Gamma; (\Delta, e) \Vdash_A \vec{t} \gg \Delta'}{\Gamma; (\Delta, e) \Vdash_A e, \vec{t} \gg \Delta'}\ \textbf{tac\_kn} \qquad \frac{(\Gamma, e : \tau, \Gamma'); (\Delta, e) \Vdash_A \vec{t} \gg \Delta'}{(\Gamma, e : \tau, \Gamma'); \Delta \Vdash_A e, \vec{t} \gg \Delta'}\ \textbf{tac\_ukn}$$

Concatenated messages can be split unconditionally. We need however to recursively analyze the two submessages since they may not be elementary.

$$\frac{\Gamma; \Delta \Vdash_A t_1, t_2, \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A (t_1\ t_2), \vec{t} \gg \Delta'}\ \textbf{tac\_cnc}$$

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 inverse of the encryption key k. This is ascertained by the left premises of the following rules. The judgment $\Gamma; \Delta \Vdash^s_A k \gg \Delta'$ (resp. $\Gamma; \Delta \Vdash^p_A k \gg \Delta'$) 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. We will discuss the key validations judgments in the next section.

$$\frac{\Gamma; \Delta \Vdash^s_A k \gg \Delta' \quad \Gamma; \Delta' \Vdash_A t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \{t\}_k, \vec{t} \gg \Delta''}\ \textbf{tac\_ske} \qquad \frac{\Gamma; \Delta \Vdash^p_A k \gg \Delta' \quad \Gamma; \Delta' \Vdash_A t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \{\!\!\{t\}\!\!\}_k, \vec{t} \gg \Delta''}\ \textbf{tac\_pke}$$

Once all possibly composite terms have been reduced to their elementary constituents (and have been shown to respect the data access specification policy), we simply return the input knowledge context that we have been extending as output knowledge. This is realized by the following rule.

$$\frac{}{\Gamma; \Delta \Vdash_A \cdot \gg \Delta}\ \textbf{tac\_dot}$$

5.2.3 Encryption Keys

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 judgment 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 data access specification rules.

We shall begin with shared-key encryption. We want to decide when the owner A of a rule can access the cleartext t of a message $\{t\}_k$ encrypted with k. We express this question by means of the following judgment:

$\Gamma; \Delta \Vdash^s_A k \gg \Delta'$	*Given knowledge Δ, principal A can decipher a message encrypted with shared key k in context Γ*

As in previous judgments, 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, which is simply returned as output knowledge. We have the following rule:

$$\frac{}{\Gamma; (\Delta, k) \ \Vdash^{s}_{A} \ k \gg (\Delta, k)}\ \textbf{kac\_ss}$$

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. The following two rules account for this possibility.

$$\frac{}{(\Gamma, k : \mathsf{shK}\ A\ B, \Gamma'); \Delta \ \Vdash^{s}_{A} \ k \gg (\Delta, k)}\ \textbf{kac\_su1} \qquad \frac{}{(\Gamma, k : \mathsf{shK}\ B\ A, \Gamma'); \Delta \ \Vdash^{s}_{A} \ k \gg (\Delta, k)}\ \textbf{kac\_su2}$$

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.

No other rules have this judgment as their conclusion. Therefore, a rule which attempts to decipher a message whose encryption key is unknown and does not belong to the rule's owner will not pass this test. It violates our data access specification policy.

We now turn to the case where access to the cleartext t of a message $\{\!\{t\}\!\}_k$ encrypted with public key k has been made. We shall rely on the following judgment to decide this situation, where the meaning of the meta-variables is as for symmetric keys:

$\Gamma; \Delta \ \Vdash^{a}_{A} \ k \gg \Delta'$	*Given knowledge Δ, principal A can decipher a message encrypted with public key k in context Γ*

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 in the case of 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 public key k. We have the following two rules, which differ by whether k was known to A or not:

$$\frac{}{(\Gamma, k : \mathsf{pubK}\ B, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k, k') \ \Vdash^{a}_{A} \ k \gg (\Delta, k, k')}\ \textbf{kac\_pss}$$

$$\frac{}{(\Gamma, k : \mathsf{pubK}\ B, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k') \ \Vdash^{a}_{A} \ k \gg (\Delta, k')}\ \textbf{kac\_pus}$$

The output knowledge shall clearly include k' since A must have access to it to perform the decryption. It may appear that k should always be included in the output knowledge as well since it appears in the message $\{\!\{t\}\!\}_k$ being deciphered. This is incorrect: k is needed to construct $\{\!\{t\}\!\}_k$ but not to access t. Including k would lead to inadequate specifications of key distribution protocols.

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 (and k'). The following rules express this possibility. Again they differ on whether A knows k, and the above explanation of why k is not necessarily part of the output context applies also here.

$$\frac{}{(\Gamma, k : \mathsf{pubK}\ A, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k) \ \Vdash^{a}_{A} \ k \gg (\Delta, k, k')}\ \textbf{kac\_psu}$$

$$\frac{}{(\Gamma, k : \mathsf{pubK}\ A, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); \Delta \ \Vdash^{a}_{A} \ k \gg (\Delta, k, k')}\ \textbf{kac\_puu}$$

We shall again emphasize how data access specification is built upon the type declarations of *MSR*, and in particular on the notion of dependent types.

5.3 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). This both alternatives have the potential of violating the data access specification policy (e.g. when trying to access the private key of a third party). We dedicate this section to enforcing these various forms of data access specification in the right-hand side of a rule. More precisely, Section 5.3.1 analyzes the high-level structure of a rule consequent, while Section 5.3.2 deals with the construction of the messages to be transmitted or stored, a major functionality of a rule's right-hand side. Finally, Section 5.3.3 analyzes information access in the right-hand side of a rule.

5.3.1 Fresh Data

data access specification on the right-hand side rhs of a rule r is performed by the following judgment:

$\Gamma; \Delta \Vdash_A \ rhs$ *Right-hand side rhs implements valid data access specification for principal A in context Γ given knowledge Δ*

where A is the owner of r, Γ is its typing context, and Δ is the knowledge gained by examining its antecedent.

This judgment is implemented by rules whose number depends on the intended application of the protocol at hand. We will first consider consequents prefixed with a fresh data declaration, *i.e.* 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 in a setting like ours, 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:

$$\frac{(\Gamma, x : \mathsf{nonce}); (\Delta, x) \Vdash_A \ rhs}{\Gamma; \Delta \Vdash_A \ \exists x : \mathsf{nonce}.\ rhs}\,\mathbf{rac\_nnc} \qquad \frac{(\Gamma, x : \mathsf{msg}); (\Delta, x) \Vdash_A \ rhs}{\Gamma; \Delta \Vdash_A \ \exists x : \mathsf{msg}.\ rhs}\,\mathbf{rac\_msg}$$

We must emphasize again that the exact set of rules for data generation depends on the intended functionalities of the protocol. Here, we adopt a minimalistic approach and do without rules for keys and principals. Other settings may require rules for selected keys.

Once all existential quantifiers have been stripped by uses of the above rule, we are left with a raw predicate sequence lhs. We invoke 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 data access specification. This is realized by the following rule:

$$\frac{\Gamma; \Delta \looparrowright_A \ lhs}{\Gamma; \Delta \Vdash_A \ lhs}\,\mathbf{rac\_mn}$$

5.3.2 Constructing Information

The main function of a rule's consequent is to constructs messages, either to be emitted over the public network, or stored for future use in a memory or role state predicate. In this section, we turn our attention to predicate sequences that appear in the right-hand side of a rule and to their embedded terms.

The premise of rule **rac_mn** above included the following judgment:

$\Gamma; \Delta \looparrowright_A \ lhs$ *Predicate sequence lhs is constructible from knowledge Δ for principal A*

which verifies that all the messages in the predicate sequence lhs, consisting only of memory and network predicates, can be constructed in the current rule. Clearly, this is based on the accumulated knowledge Δ.

This judgment is implemented by the following three rules when lhs is not empty. They rely on the term constructions judgments "$\Delta \hookrightarrow t$" and "$\Delta \hookrightarrow \vec{t}$" that will be explained shortly.

$$\frac{\Gamma; \Delta \hookrightarrow_A t \quad \Gamma; \Delta \hookrightarrow_A lhs}{\Gamma; \Delta \hookrightarrow_A \mathsf{N}(t), lhs}\,\textbf{rac\_net} \qquad \frac{\Gamma; \Delta \hookrightarrow_A \vec{t} \quad \Gamma; \Delta \hookrightarrow_A lhs}{\Gamma; \Delta \hookrightarrow_A \mathsf{M}_A(\vec{t}), lhs}\,\textbf{rac\_mem} \qquad \frac{\Gamma; \Delta \hookrightarrow_A (A, \vec{e}) \quad \Gamma; \Delta \hookrightarrow_A lhs}{\Gamma; \Delta \hookrightarrow_A L(A, \vec{e}), lhs}\,\textbf{rac\_rsp}$$

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 memory predicates. Observe that the first argument of any role state predicate must be the owner A of the rule it appears in.

Empty predicate sequences are always valid. This results from the following simple rule:

$$\frac{}{\Gamma; \Delta \hookrightarrow_A \cdot}\,\textbf{rac\_dot}$$

The constructability of terms in the right-hand side of a rule is expressed by the following judgment:

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

Elementary information appearing in the right-hand side of a rule can come from of two sources. Rule **cac_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 **cac_ukn** through the right-hand side access judgment "$\Gamma \hookrightarrow_A e$", that will be discussed in Section 5.3.3.

$$\frac{}{\Gamma; (\Delta, e) \hookrightarrow_A e}\,\textbf{cac\_kn} \qquad \frac{\Gamma \hookrightarrow_A e}{\Gamma; (\Delta, e) \hookrightarrow_A e}\,\textbf{cac\_ukn}$$

Composite terms are recursively reduced to their atomic constituents by the following three rules:

$$\frac{\Gamma; \Delta \hookrightarrow_A t_1 \quad \Gamma; \Delta \hookrightarrow_A t_2}{\Gamma; \Delta \hookrightarrow_A t_1\, t_2}\,\textbf{cac\_cnc} \qquad \frac{\Gamma; \Delta \hookrightarrow_A t \quad \Gamma; \Delta \hookrightarrow_A k}{\Gamma; \Delta \hookrightarrow_A \{t\}_k}\,\textbf{cac\_ske} \qquad \frac{\Gamma; \Delta \hookrightarrow_A t \quad \Gamma; \Delta \hookrightarrow_A k}{\Gamma; \Delta \hookrightarrow_A \{\!\{t\}\!\}_k}\,\textbf{cac\_pke}$$

Since these rules depend on the term constructors of the language at hand, they shall be updated for syntaxes that include additional operators, such as digital signatures.

The following judgment allows constructing message tuples $\vec{t}$ from the knowledge Δ at hand:

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

It is implemented by the following two simple rules.

$$\frac{}{\Gamma; \Delta \hookrightarrow_A \cdot}\,\textbf{cac\_dot} \qquad \frac{\Gamma; \Delta \hookrightarrow_A t \quad \Gamma; \Delta \hookrightarrow_A \vec{t}}{\Gamma; \Delta \hookrightarrow_A (t, \vec{t})}\,\textbf{cac\_ext}$$

5.3.3 Accessing Information on the Right-Hand Side

We conclude this section by describing the judgment that checks that a rule owner A has legitimate access to elementary data e that appear only in the consequent of this rule. This is achieved by the following judgment,

$\Gamma \hookrightarrow_A e$ *Principal A can access atomic information e in context Γ*

where Γ is the typing context in which e is defined.

The implementation of this judgment 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, and indeed we have the following rule:

$$\frac{}{(\Gamma, e : \mathsf{principal}, \Gamma') \looparrowright_A e}\ \mathbf{eac\_pr}$$

The remaining simple types are nonce and msg. Were we to have a rule similar to **eac_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, and this judgment does not need to be invoked. A similar argument applies to elementary objects of type msg: a rule akin to **eac_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 nearly all our types (see Section 2.2): this would authorize A to use all data that can be constructed in the signature the rule is executed in.

Next, let us consider shared keys. Clearly, A should have free access to all of his/her shared keys, but to no others. This is expressed by the following two rules:

$$\frac{}{(\Gamma, e : \mathsf{shK}\ A\ B, \Gamma') \looparrowright_A e}\ \mathbf{eac\_s1} \qquad \frac{}{(\Gamma, e : \mathsf{shK}\ B\ A, \Gamma') \looparrowright_A e}\ \mathbf{eac\_s2}$$

A similar argument holds for asymmetric keys: A has legitimate access to all of his/her private keys, and to the public keys of any principal.

$$\frac{}{(\Gamma, e : \mathsf{privK}\ k, \Gamma', k : \mathsf{pubK}\ A, \Gamma'') \looparrowright_A e}\ \mathbf{eac\_pp} \qquad \frac{}{(\Gamma, e : \mathsf{pubK}\ B, \Gamma') \looparrowright_A e}\ \mathbf{eac\_p}$$

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

This concludes the presentation of the data access specification judgments and rules for protocol rules and the syntactic entities they are part of, i.e. roles and protocol theories. We will collect all this sparse information in Appendix A. We will prove decidability results for these judgments in Section 5.5. We shall first extend the discussion we just concluded to active roles.

5.4 Active Roles

In Section 4.4 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, and this section will be dedicated to showing how this is achieved.

Intuitively, active roles are handled by allowing ground terms anywhere variables can appear in a role, and by treating them in the same way. We start therefore by updating the notion of knowledge context from Section 5.1 as follows:

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

Here, t stands for a ground term that has been substituted for a variable. Observe that active roles may contain bound variables that have yet to be instantiated. For this reason, we still include variables in the above definition.

As suggested above, instantiating terms shall be treated exactly in the same way as elementary information in the previous section. Therefore, we will examine in turn all the data access specification rules that use elementary terms and produce a version that can deal with active roles.

We start with rule **lac_rsp** which added the arguments of a role state predicate in a rule's antecedent to the current knowledge context. By systematically replacing "e" with "t", we obtain the following variant:

$$\frac{\Delta \;>\; (\mathsf{A}, \vec{t}) \;>\; \Delta' \quad \Gamma; \Delta' \Vdash_{\mathsf{A}} \mathit{lhs} > \vec{t}' \gg \Delta''}{\Gamma; \Delta \Vdash_{\mathsf{A}} (L(\mathsf{A}, \vec{t}), \mathit{lhs}) > \vec{t}' \gg \Delta''}\,\textbf{lac\_rsp}*$$

Since role state predicated parameters may have been instantiated in an active roles, L here should be interpreted as a constant or a variable. Observe that the arguments $(A, \vec{t})$ of L are inserted in the current knowledge context while the arguments of the network and memory predicates are introduced in the central area of this judgment. This implies that composite elements of $\vec{t}$ cannot be broken into elementary pieces. This is sensible since they are ground instantiations of variables, and a rule has no access to the expected structure of its variables.

Next, we need to update the implementation of the merge judgment that appears in the left premise of rule **lac_rsp**$*$. We have the following two rules:

$$\frac{\Delta \;>\; \vec{t} \;>\; \Delta'}{\Delta \;>\; t, \vec{t} \;>\; (\Delta', t)}\,\textbf{mac\_ukn}* \qquad\qquad \frac{\Delta \;>\; \vec{t} \;>\; \Delta'}{(\Delta, t) \;>\; t, \vec{t} \;>\; (\Delta', t)}\,\textbf{mac\_kn}*$$

The next rule to upgrade is **tac_kn** that checked for known pieces of elementary information in memory or network predicates. In an active role, such a term could have been the result of instantiation. Therefore, we must allow it to be looked up in the current knowledge context. We obtain the following rule:

$$\frac{\Gamma; (\Delta, t) \Vdash_{\mathsf{A}} \vec{t} \gg \Delta'}{\Gamma; (\Delta, t) \Vdash_{\mathsf{A}} t, \vec{t} \gg \Delta'}\,\textbf{tac\_kn}*$$

We now move to the right-hand side and to rule **rac_rsp**, which updated the current knowledge with pieces of information appearing for the first time in a role state predicate in a rule's consequent. As usual, it suffices to upgrade the e's to t's, which yields the following data access specification rule (again, L may be either a parameter or a role state predicate constant):

$$\frac{\Gamma; \Delta \looparrowright_A (A, \vec{t}) \quad \Gamma; \Delta \looparrowright_A \mathit{lhs}}{\Gamma; \Delta \looparrowright_A L(A, \vec{t}), \mathit{lhs}}\,\textbf{rac\_rsp}*$$

Our last upgrade involves rule **cac_gen**, which enabled constructing an object consisting of a previously known elementary message. We simply need to extend it to arbitrary ground terms:

$$\frac{}{(\Delta, t) \looparrowright t}\,\textbf{cac\_gen}*$$

We may expect a similar update for rule **cac_ukn**, which checks whether information not previously seen can be accessed to construct a term in the right-hand side of a rule. We observed in Section 5.3.3 that the implementation of the judgment "$\Gamma \looparrowright_A e$" depends on the term language in use. In our case study, we argued that this judgment can be fulfilled only if e is a principal name, one of A's shared or private keys, or an arbitrary public key. When dealing with an active role ρ^{A}, it may be tempting to accept more complex terms t in this position: for example the concatenation of a principal name and his public key seems completely innocuous. Nonetheless, we will disallow these possibilities: if t is a composite term, it has type msg and can only result from the instantiation of some variable x. Therefore, the role ρ from which ρ^{A} descends contains a variable x of type msg that appears for the first time in the right-hand side of one of its rules. This rule would violate our data access specification policy since x could have been instantiated to a term A does not have access to. If ρ is part of the protocol theory $\mathcal{P}$ that models some protocol under analysis, the data access specification judgment $\Sigma \Vdash \mathcal{P}$ would not be satisfiable.

In summary, our case study does not require modifying the judgment "$\Gamma \looparrowright_A e$" that checks whether A can be granted access to an object e that appears for the first time in the right-hand side of a rule. However, more complex languages may need to make changes to the implementation of this judgment that apply only to active roles.

The infrastructure in which the above upgrades take place has been kept informal for the sake of clarity. Clearly these rule variants are applicable only when examining an active role, and should not be used for validating the data access specification policy in a rule. A precise way to enforce this distinction is to create an active role version of each of the involved judgments, and to accordingly duplicate all the original rules (with new names). We leave this as an exercise to the zealous reader.

We now introduce the following judgment, that expresses the fact that an active role set satisfies our data access specification policy in the current signature:

$\Sigma \Vdash R$	*Active role set R implements valid data access specification in signature Σ*

It is implemented by the following two simple rules, where the right premise of rule **aac_ext** calls the variant of the rule collection data access specification judgment for active roles:

$$\frac{}{\Sigma \Vdash \cdot}\,\textbf{aac\_dot} \qquad\qquad \frac{\Sigma \Vdash R \quad \Sigma \Vdash_{\mathsf{A}} \rho}{\Sigma \Vdash R, \rho^{\mathsf{A}}}\,\textbf{aac\_ext}$$

5.5 Decidability of Data Access Specification

We continue this introduction to data access specification in *MSR* by proving that all the judgments presented in this section have decidable implementations. Furthermore, we will show that the judgments to which we have ascribed a functional behavior implement computable relations.

For simplicity, we will structure this presentation as follows: we will prove the decidability results concerning the antecedent and consequent of a rule in Sections 5.5.1 and 5.5.2, respectively. We will then put these properties together in Section 5.5.3 to produce analogous results for roles and protocol theories. Finally, we extend these findings to active role sets in Section 5.5.4. Throughout this section, it will be convenient to assume that we are exclusively working with typable objects. Clearly, a knowledge context is typable if all the contained terms are well-typed in the current signature or typing context.

5.5.1 Deciding the Left-Hand Side Judgments

Before tackling the task of proving computability issues about the numerous data access specification judgments that apply to the antecedent of a rule, it may be useful to recall them. Section 5.2 mentioned the following relations, where we have listed them in inverse order of their introduction, so that whenever a judgment J_1 depends on judgment J_2, J_1 is always listed after J_2 (we do not have mutually recursive judgments):

	$\Gamma; \Delta \Vdash^{s}_{A} k \gg \Delta'$	*Given knowledge Δ, principal A can decipher a message encrypted with shared key k in context Γ*	[p. 23]
	$\Gamma; \Delta \Vdash^{p}_{A} k \gg \Delta'$	*Given knowledge Δ, principal A can decipher a message encrypted with public key k in context Γ*	[p. 24]
$\Rightarrow$	$\Gamma; \Delta \Vdash_{A} \vec{t} \gg \Delta'$	*Given knowledge Δ and terms $\vec{t}$, principal A can knows Δ' in context Γ*	[p. 22]
	$\Delta > \vec{e} > \Delta'$	*Merging context knowledge Δ and elementary term tuple $\vec{e}$ yields Δ'*	[p. 22]
$\Rightarrow$	$\Gamma; \Delta \Vdash_{A} lhs > \vec{t} \gg \Delta'$	*Given knowledge Δ, predicate sequence lhs and terms $\vec{t}$, principal A can knows Δ' in context Γ*	[p. 21]

Clearly, we need to prove the decidability of each and every one of them. Rather than subjugating the reader to this long and tedious process, we pick the few key judgments marked with an arrow ($\Rightarrow$) on their left and only hint to the required results for the judgments they depend on (listed above them).

Property 5.1

Let Γ be a typing context, Δ and Δ' knowledge contexts, A a principal name , and $\vec{t}$ a multiset of terms. It is decidable whether the judgment $\Gamma; \Delta \Vdash_{A} \vec{t} \gg \Delta'$ holds. Moreover, for any choice of Γ, Δ, A and $\vec{t}$, there are finitely many knowledge contexts Δ' for which this judgment holds.

Proof: Since the rules that implement this judgment make use of the decryption judgments for shared and public keys, we shall first discuss the analogous property concerning them. Let us consider the former, "$\Gamma; \Delta \;\Vdash^{s}_{A}\; k \gg \Delta'$". In our language, this judgment is realized by exactly three inference rules, none of which has premises (see Section 5.2.3). This makes it clearly decidable. Moreover, depending on which rule is used (assuming one is applicable), the output knowledge context Δ' will either be identical to Δ (in rule $\mathbf{kac\_ss}$), or result from adding k to it (in rules $\mathbf{kac\_su1}$ and $\mathbf{kac\_su2}$): there are exactly two options. It is indeed possible, although probably not useful, to add a key that is already in the knowledge context; therefore these two rule sets are not exclusive. A similar argument applies to the public key decryption judgment. Observe that this part of the proof is language dependent.

We now turn to the judgment "$\Gamma; \Delta \;\Vdash_{A}\; \vec{t} \gg \Delta'$", which is implemented by the six rules given in Section 5.2.2. Which one to use is determined by $\vec{t}$. Several may be applicable at any instant, but the possible instantiations is limited by the number of elements of $\vec{t}$, which is finite. Therefore, in order to prove our decidability result, we only have to show that no branch of the derivation tree obtained by chaining these rules can be infinite, no matter which instantiation is chosen. In order to do so, observe that the premises of the rules in Section 5.2.2 either call the decryption judgment for keys, that we have just proved decidable, or invoke our judgment recursively on a multiset "smaller" than $\vec{t}$. Here the size of $\vec{t}$ is computed by counting the number of constructors and elementary terms that appear in it. Since the size of $\vec{t}$ cannot decrease *ad infinitum*, the derivation tree must be finite. Therefore the above judgment is decidable.

As for the number of possible output knowledge contexts, observe that Δ' is set to Δ when all terms $\vec{t}$ have been processed by rule $\mathbf{tac\_dot}$. Therefore, we simply need to track how the input knowledge is modified when moving from the conclusion to the premises of a rule. Rules $\mathbf{tac\_ske}$ and $\mathbf{tac\_pke}$ invoke the decryption key judgment; we saw that each encryption operation in $\vec{t}$ can double the number of possible output contexts. Since rules $\mathbf{tac\_kn}$ and $\mathbf{tac\_ukn}$ are not exclusive, each piece of elementary information in $\vec{t}$ can also double this number. In any case, since $\vec{t}$ is finite in size, only finitely many knowledge contexts can be produced. □

We now turn to the left-hand side as a whole and show that the associated data access specification judgment is decidable. We have the following property:

Property 5.2

Let Γ be a typing context, Δ and Δ' knowledge contexts, A a principal name , lhs the left-hand side of a rule, and $\vec{t}$ a multiset of terms. It is decidable whether the judgment $\Gamma; \Delta \;\Vdash_{A}\; lhs > \vec{t} \gg \Delta'$ holds. Moreover, for any choice of Γ, Δ, A, lhs and $\vec{t}$, there are finitely many knowledge contexts Δ' for which this judgment holds.

Proof: The four rules implementing this judgment are displayed in Section 5.2.1. They rely on the term multiset validation judgment, for which we have just proved an analogous property, and on the knowledge merge judgment "$\Delta > \vec{e} > \Delta'$". Arguments akin to those used for Property 5.1 show that this latter relation is decidable and yields finitely many output knowledge contexts. Armed with these results, we observe that the number of possible instantiations of "$\Gamma; \Delta \;\Vdash_{A}\; lhs > \vec{t} \gg \Delta'$" is determined by lhs. Moreover, this same value also limits the number of possible consecutive invocations of this judgment. Therefore, it is decidable. The output knowledge counting argument proceeds as in Property 5.1. □

This conclude our investigation of the decidability results for the left-hand side of a rule. Observe that, assuming all other parameters fixed, the number of output knowledge contexts is exponential in the number of constants and variables we are starting from. At most one of these contexts will satisfy the analogous relation for the right-hand side of a rule. They will however be identical up to duplication of data, a fact that can be used to obtain a deterministic data access specification verification procedure for the antecedent. It should however be observed that the winning knowledge context will generally be the one with minimum redundancy when analyzing well-written specifications.

5.5.2 Deciding the Right-Hand Side Judgments

We move now to the right-hand side of a rule. The judgments we will be concerned about, collected in inverse dependency order from Section 5.3, are:

	$\Gamma \looparrowright_A e$	*Principal A can access atomic information e in context Γ*	[p. 26]
	$\Gamma; \Delta \looparrowright_A t$	*Given knowledge Δ, principal A can construct term t*	[p. 26]
	$\Gamma; \Delta \looparrowright_A \vec{t}$	*Given knowledge Δ, principal A can construct term tuple $\vec{t}$*	[p. 26]
$\Rightarrow$	$\Gamma; \Delta \looparrowright_A lhs$	*Predicate sequence lhs is constructible from knowledge Δ for principal A*	[p. 25]
$\Rightarrow$	$\Gamma; \Delta \Vdash_A rhs$	*Right-hand side rhs implements valid data access specification for principal A in context Γ given knowledge Δ*	[p. 25]

Again we will limit our statements and proofs to the judgments marked with an arrow. We start with the construction judgment for network and memory predicate sequences.

Property 5.3

Let Γ be a typing context, Δ a knowledge set, A a principal name, and lhs a sequence of network and memory predicates. It is decidable whether the judgment $\Gamma; \Delta \looparrowright_A lhs$ holds.

Proof: The rules implementing this judgment in Section 5.3 depend on the construction judgments for terms and term tuples, which in turn depends on the accessibility judgment for elementary terms "$\Gamma \looparrowright_A e$". The latter is clearly decidable in our language since it is implemented in Section 5.3.3 by five rules without premises. The applicability of the construction judgments is completely determined by the structure of their rightmost parameter. Therefore they are decidable. As for the judgment at hand, the number of applicable instantiations of the rules that implement it is determined by lhs, and so is the number of consecutive invocations of this judgment. Since a right-hand side always contains finitely many predicates, this judgment is decidable. □

We now turn to the judgment that validates data access specification for a whole right-hand side. We have the following property:

Property 5.4

Let Γ be a typing context, Δ a knowledge set, A a principal name , and rhs the right-hand side of a rule. It is decidable whether the judgment $\Gamma; \Delta \Vdash_A rhs$ holds.

Proof: Given the above result, the decidability of "$\Gamma; \Delta \Vdash_A rhs$" can be ascertained by inspecting the three rules that define it in Section 5.3.1. It is easy to see that the number of recursive invocations of this judgment is given by the number of fresh data declarations "$\exists x : \tau$", plus one. Therefore, this judgment is decidable. □

This concludes our treatment of the right-hand side of a protocol rule.

5.5.3 Deciding Data Access Specification for Protocol Theories

We will now put together the results obtained in Sections 5.5.1 and 5.5.2 in order to obtain decidability results for rules, roles and protocol theories. The involved judgments, in inverse dependency order, are the following:

$\Rightarrow$	$\Gamma \Vdash_A r$	*Rule r implements valid data access specification for principal A in context Γ*	[p. 19]
	$\Gamma \Vdash_A \rho$	*Rule collection ρ implements valid data access specification for principal A in context Γ*	[p. 19]
$\Rightarrow$	$\Sigma \Vdash \mathcal{P}$	*Protocol theory $\mathcal{P}$ implements valid data access specification in signature Σ*	[p. 19]

We will show this property for the two judgments marked with an arrow, and only hint at this result for the central relation. Let us then start with showing that the data access specification judgment for protocol rules is decidable.

Property 5.5

Let Σ be a signature, A a principal name, and r a rule. It is decidable whether the judgment $\Gamma \Vdash_A r$ holds.

Proof: This judgment is implemented by two rules ($\mathbf{uac\_all}$ and $\mathbf{uac\_core}$) displayed at the end of Section 5.1. The former is unproblematic since a rule is prefixed with finitely many universal quantifiers. The latter depends on the data access specification judgments for the antecedent and the consequent of r. We have shown these judgments decidable in Properties 5.2 and 5.4. However, the knowledge context Δ appears only in the premises and could be a source of undecidability. This is not the case as Property 5.2 showed that the leftmost premise can produced only a finite number of such contexts. □

We can now show that data access specification for roles and protocol theories can be effectively verified. We have the following property in the latter case:

Property 5.6

Let Σ be a signature and $\mathcal{P}$ a protocol theory over Σ. It is decidable whether the judgment $\Sigma \Vdash \mathcal{P}$ holds.

Proof: The rules implementing this judgment rely on the data access specification relation for roles "$\Gamma \Vdash_A \rho$", described in Section 5.1. This judgment is clearly decidable since it deterministically recurses over the finitely many rules and universal quantifications that constitute ρ. We have proved that data access specification is decidable for rules in Property 5.5.

This result allows us to prove the decidability of data access specification for protocol theories using a similar argument. □

5.5.4 Deciding Data Access Specification for Active Role Sets

We now extend the above results to active roles. The involve judgments are:

	(...)		[sec. 5.4]
$\Rightarrow$	$\Sigma \Vdash R$	*Active role set R implements valid data access specification in signature Σ*	[p. 29]

where the first line corresponds to the variants of most of the judgments examined so far, with the additional rules seen in Section 5.4 to deal with variables instantiated with ground terms. The decidability proofs for these variants differs from the "originals" above by the fact that more cases need to be considered. Concretely, these rules raise the number of possible output knowledge contexts (when one is present), but never make it infinite.

We have the following decidability result for active roles:

Property 5.7

Let Σ be a signature and R an active role set over Σ. It is decidable whether the judgment $\Sigma \Vdash R$ holds.

Proof: Given the above discussion on the decidability of the judgment variants for active roles, the above judgment is trivially valid since the rules that implement it (given in Section 5.4) simply recurse on the (finite) number of active roles present in R. □

This concludes our analysis of the computability properties of the data access specification judgments discussed in this section.

5.6 Changes

No notion of data access specification was present in our earlier versions of *MSR* [27, 30]. It is however interesting to observe that the satisfaction of these judgments constrains the structure of admissible protocol theories beyond typing (see Section 4) and beyond the requirements of these initial versions.

1. The fact that the arguments of a role state predicate should be elementary terms was mostly implicit in our earlier work.

2. Moreover, the requirement that the arguments of occurrences of these predicates on the right-hand side of a rule should mention all the information used in this rule was only verbally specified.

3. Finally, the request that the first argument of a role state predicate be a principal (the role owner) was made in [30] and [31], but not in previous work.

6 Protocol Execution

The typing and data access specification policies defined in the previous sections provide ways of statically checking that the *MSR* specification of a protocol meets well-understood default notions of adequacy. More complex analyses require a description of how a protocol evolves at run time. In this section, we will define such an execution model for *MSR*, and study how it interacts with typing and data access specification. More precisely, we will first complete the definition of the pervasive notion of substitution in Section 6.1. Then, we will present the basic one-step rule application and its sequential iteration in Section 6.2. In Section 6.3, we extend this notion to allow concurrent rule firing. In Section 6.4, we show that execution preserves typing and in Section 6.5 we prove a similar result about its interaction with data access specification. We conclude in Section 6.6 with a discussion of the main changes with respect to previous versions of *MSR*.

6.1 Substitutions

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

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

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

Our definition of this operation is relatively standard, but some aspects deserve comments. For the sake of unity, we start by redisplaying the definition of substitution for terms, types and tuple types from Section 3.1.3:

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

Observe again that, while dependent tuple types are universal constructions, the defining equalities for terms and types are language-specific, and therefore need to be adapted for languages more elaborate than what considered in this document. It should also be noticed that instantiation is concerned with the syntactic form of an object: in particular, a substitution should be applied to the encryption key of an encrypted message, even though the key itself it not (typically) transmitted.

Substitution for term tuples, another universal construction, is defined by the following two simple equalities:

$[t/x]\vec{t}$
$[t/x]\cdot \;=\; \cdot$
$[t/x](t',\ \vec{t}) \;=\; ([t/x]t'),\ ([t/x]\vec{t})$

We next turn to the predicate sequences, intended to model the left-hand side of a protocol rule, and to rule consequents. In the former case, the operation reduces to applying the substitution to the arguments of all the predicates that constitute the sequence lhs. Notice in particular that the owner A of a memory predicate $\mathsf{M}_A(\vec{t})$ is itself an argument, although in a special position. The first line in the second column deals with consequents that are simple predicate sequences. Although it looks redundant, we should remember that our syntax is overloaded: substitution on a degenerate consequent reduces to substitution on the embedded antecedent (defined in the column on the left). In the second line on the rightmost column, we rely on implicit α-conversion to make sure that the bound variable will have a different name than the variable x being instantiated. Observe how the substitution is pushed inside the type declaration.

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

Last, we turn to rules and rule collections. Again, we take advantage of α-conversion to avoid making special provision in case the bound variable happens to be called x. These definitions systematically push the substitution inside all the syntactic elements of a rule (collection).

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

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

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

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

The implementation of this operation is similar to term substitution. We have the following definition in the case of rule antecedent and rule consequents, respectively. Clearly this substitution does not need to be pushed into terms.

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

The application of a role state predicate substitution to rule and a rule collection is given below. Observe that again we rely on implicit α-conversion when pushing the substitution inside the right-hand side of a rule collection.

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

This concludes the definition of the notion of substitution that will be needed in the rules modeling the execution of a protocol. In Sections 6.4 and 6.5, when studying how execution interacts with typing and data access specification, we will need to observe the effect of performing a substitution relative to the typing and/or knowledge contexts of a judgment. For this reason, we need to define an instantiation operation for these objects as well. Given a typing context Γ and a knowledge context Δ, we will encounter the notation:

$$[t/x]\Gamma \qquad\qquad [t/x]\Delta$$

The defining equalities are given in the table below. Observe that in the case of a typing context, this operation applies only to the type part of a declaration, and not to the introduced variable: we will never be in a position where x should be substituted in a context containing the declaration $x : \tau$. Substitution over a typing context is given on the right column. Its last line is concerned with the extended notion of knowledge needed to handle active roles: it should be remembered that t' cannot be but a ground term.

$[t/x]\Gamma$	$[t/x]\Delta$
$[t/x]\Sigma = \Sigma$	$[t/x]\cdot = \cdot$
$[t/x](\Gamma, y : \tau) = ([t/x]\Gamma), y : ([t/x]\tau)$	$[t/x](\Delta, a) = ([t/x]\Delta),\ a$
$[t/x](\Gamma, L : \vec{\tau}) = ([t/x]\Gamma), L : ([t/x]\vec{\tau})$	$[t/x](\Delta, y) = \begin{cases} ([t/x]\Delta),\ t & \text{if } y = x \\ ([t/x]\Delta),\ y & \text{otherwise} \end{cases}$
	$[t/x](\Delta, t') = ([t/x]\Delta),\ t'$

6.2 Sequential Firing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We now consider execution steps resulting from the application of a fully instantiated rule of the form "$\mathit{lhs} \to \mathit{rhs}$". 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. This latter operation is performed in the premise of this rule by the right-hand side instantiation judgment "$(\mathit{rhs})_{\Sigma} \gg (\mathit{lhs}')_{\Sigma'}$" that will be discussed shortly.

$$\frac{(\mathit{rhs})_{\Sigma} \gg (\mathit{lhs}')_{\Sigma'}}{(\mathit{lhs} \to \mathit{rhs}) \triangleright [S, \mathit{lhs}]_{\Sigma} \gg [S, \mathit{lhs}']_{\Sigma'}}\ \textbf{sex\_core}$$

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

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

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

$$\frac{}{(\mathit{lhs})_{\Sigma} \gg (\mathit{lhs})_{\Sigma}}\ \textbf{sex\_seq}$$

The instantiation of an existentially quantified term variable x is handled in rule **sex_nnc** below. We generate a fresh term constant a of the appropriate type, records this fact in the signature, and replaces every occurrence of x with a before examining the body of rhs.

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

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

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

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

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

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

The following simple lemma will turn useful in the sequel.

Lemma 6.1 (*Chaining of Sequential Firings*)

If $\mathcal{P} \triangleright C \longrightarrow^* C'$ *and* $\mathcal{P} \triangleright C' \longrightarrow^* C''$, *then* $\mathcal{P} \triangleright C \longrightarrow^* C''$.

Proof: By induction on the structure of a derivation for "$\mathcal{P} \triangleright C \longrightarrow^* C'$". □

6.3 Parallel Firing

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

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

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

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

$$\frac{}{\mathcal{P} \triangleright C \Longrightarrow C}\,\mathbf{pex\_id} \qquad \frac{\mathcal{P} \triangleright [S_1]_{\Sigma}^{R_1} \longrightarrow [S_1']_{(\Sigma,\Sigma_1')}^{R_1'} \quad \mathcal{P} \triangleright [S_2]_{\Sigma}^{R_2} \Longrightarrow [S_2']_{(\Sigma,\Sigma_2')}^{R_2'}}{\mathcal{P} \triangleright [S_1, S_2]_{\Sigma}^{(R_1,R_2)} \Longrightarrow [S_1', S_2']_{(\Sigma,\Sigma_1',\Sigma_2')}^{(R_1',R_2')}}\,\mathbf{pex\_par}$$

Technically, rule $\mathbf{pex\_par}$ operates as follows: it splits the current state in two parts S_1 and S_2, and similarly partitions the active role set into R_1 and R_2. The left premise applies one atomic execution step to the partial snapshot $[S_1]_{\Sigma}^{R_1}$, producing $[S_1']_{(\Sigma,\Sigma_1')}^{R_1'}$ as a result, where Σ_1' is the amount by which the current signature Σ has been extended during this step. Rather than having an unbounded number of premises, we call our parallel firing judgment recursively in the rightmost premise: in zero or more atomic steps, we transform the remainder of the initial snapshot, $[S_2]_{\Sigma}^{R_2}$, into $[S_2']_{(\Sigma,\Sigma_2')}^{R_2'}$ where again Σ_2' is the resulting signature extension. The overall output snapshot is constructed by taking the multiset union of the returned states S_1' and S_2', juxtaposing the output active role sets R_1' and R_2', and combining the resulting signatures (Σ, Σ_1') and (Σ, Σ_2') into $(\Sigma, \Sigma_1', \Sigma_2')$.

It should be noted that the input snapshots to the premises of rule $\mathbf{pex\_par}$ have no state element or active role in common: they are independent. They both rely on the same signature Σ since unrelated state objects and active roles may in general refer to the same constants. The application of a basic step can at most extend the initial signature Σ (see Lemma 6.2 below), which justifies expressing the signature output by the two premises as (Σ, Σ_i), for $i = 1, 2$. We can always choose the newly generated names so that Σ_1 and Σ_2 do not declare the same constant: under this assumption, $(\Sigma, \Sigma_1, \Sigma_2)$ is structurally a well-formed signature.

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

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

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

$$\frac{}{\mathcal{P} \rhd C \Longrightarrow^* C}\;\mathbf{pex\_it0} \qquad\qquad \frac{\mathcal{P} \rhd C \Longrightarrow C' \quad \mathcal{P} \rhd C' \Longrightarrow^* C''}{\mathcal{P} \rhd C \Longrightarrow^* C''}\;\mathbf{pex\_itn}$$

We conclude this section with a few results concerning execution, ultimately with the admissibility of parallel firing rules. We will investigate the relationship between execution and the typing and data access specification rules in Sections 6.4 and 6.5.

We start with a lemma that asserts that, as execution proceeds, the initial signature can only be extended. As a space saver, we write $\mathcal{P} \rhd C \longrightarrow^{(*)} C'$ to indicate that we are considering the one-step ($\mathcal{P} \rhd C \longrightarrow C'$) and the multi-step ($\mathcal{P} \rhd C \longrightarrow^* C'$) versions of the sequential firing judgment at once. We adopt a similar convention for the parallel firing judgments.

Lemma 6.2 (*Signature Extension*)

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

1. *If* $(rhs)_\Sigma \gg (lhs)_{\Sigma'}$, *then* $\Sigma' = (\Sigma, \Sigma'')$ *for some signature* Σ''.
2. *If* $r \rhd [S]_\Sigma \gg [S']_{\Sigma'}$, *then* $\Sigma' = (\Sigma, \Sigma'')$ *for some signature* Σ''.
3. *If* $\mathcal{P} \rhd [S]^R_\Sigma \longrightarrow^{(*)} [S']^{R'}_{\Sigma'}$, *then* $\Sigma' = (\Sigma, \Sigma'')$ *for some signature* Σ''.

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

The first result (1) is immediate for rule **sex_seq** and follows by an appeal to the induction hypothesis for rule **sex_nnc**. The second point in this lemma (2) relies on (1) for rule **sex_core** and the induction hypothesis for rule **sex_all**. The last result (3), is immediate for rules **sex_arole**, **sex_grole**, **sex_rsp**, **sex_skp**, **sex_dot**, and **sex_it0**. It relies on (1) for rule **sex_rule** and follows by induction for rule **sex_itn**. □

We now turn to proving that parallel firing can be emulated by sequential execution. The proof of this property relies on a number of simple lemmas, the first of which, given below, states that term typing is invariant with respect to weakening: if a term is typable in a signature, it remains typable when considering additional declarations. We will extend this result in Section 6.4.

Lemma 6.3 (*Weakening Term Typing*)

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

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

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

Lemma 6.4 (*Weakening Execution*)

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

1. *if* $(rhs)_{\Sigma} \gg (lhs)_{\Sigma,\Sigma''}$, *then* $(rhs)_{\Sigma,\Sigma'} \gg (lhs)_{\Sigma,\Sigma',\Sigma''}$.
2. *if* $r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma,\Sigma''}$, *then* $r \triangleright [S]_{\Sigma,\Sigma'} \gg [S']_{\Sigma,\Sigma',\Sigma''}$.
3. *if* $\mathcal{P} \triangleright [S]^{R}_{\Sigma} \longrightarrow^{(*)} [S'']^{R''}_{\Sigma,\Sigma''}$, *then* $\mathcal{P} \triangleright [S,S']^{R,R'}_{\Sigma,\Sigma'} \longrightarrow^{(*)} [S',S'']^{R',R''}_{\Sigma,\Sigma',\Sigma''}$.
4. *if* $\mathcal{P} \triangleright [S]^{R}_{\Sigma} \Longrightarrow^{(*)} [S'']^{R''}_{\Sigma,\Sigma''}$, *then* $\mathcal{P} \triangleright [S,S']^{R,R'}_{\Sigma,\Sigma'} \Longrightarrow^{(*)} [S',S'']^{R',R''}_{\Sigma,\Sigma',\Sigma''}$.

Proof: We proceed by induction on the structure of a derivation for the antecedent of each result. In (1), the property is immediate for rule **sex_seq** and is obtained by application of the induction hypothesis for rule **sex_nnc**. We proceed similarly in the case of (2), except that we make use of (1) for rule **sex_rule** and of Lemma 6.3 for rule **sex_all**. The treatment of (3) is again similar, with the use of (2) for rule **sex_rule** and of Lemma 6.3 for rules **sex_grole**. Finally, (4) is again treated along the same lines, but it should be noted that the state and active role sets extensions (S' and R' respectively) can be split arbitrarily in the premises of rule **pex_par**. □

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

Theorem 6.5 (*Admissibility of Parallel Execution*)

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

Proof: In its forward direction, the proof of this result proceeds by induction on the structure of a given derivation $\mathcal{E}$ of the judgment $\mathcal{P} \triangleright C \Longrightarrow^{(*)} C'$. We will examine the one-step case in detail, but omit the simple proof of its multi-step extension. We proceed by cases on the last rule of $\mathcal{E}$: for single-step parallel firing, only rules **pex_id** and **pex_par** need to be considered.

$\boxed{\textbf{pex\_id}}$ $\quad \mathcal{E} = \dfrac{}{\mathcal{P} \triangleright C \Longrightarrow C}\ \textbf{pex\_id}$

with $C' = C$.

The result is obtained by instantiating rule **sex_it0** with the same parameters.

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

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

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

$\mathcal{E}'_1 :: \mathcal{P} \triangleright [S_1,S_2]^{R_1,R_2}_{\Sigma} \longrightarrow [S'_1,S_2]^{R'_1,R_2}_{(\Sigma,\Sigma'_1)}$	by the Weakening Lemma 6.4(2) on $\mathcal{E}_1$,
$\mathcal{E}^*_2 :: \mathcal{P} \triangleright [S_2]^{R_2}_{\Sigma} \longrightarrow^{*} [S'_2]^{R'_2}_{(\Sigma,\Sigma'_2)}$	by induction hypothesis on $\mathcal{E}_2$,
$\mathcal{E}'_2 :: \mathcal{P} \triangleright [S'_1,S_2]^{R'_1,R_2}_{\Sigma,\Sigma'_1} \Longrightarrow [S'_1,S'_2]^{R'_1,R'_2}_{(\Sigma,\Sigma'_1,\Sigma'_2)}$	by the Weakening Lemma 6.4(3) on $\mathcal{E}^*_2$,
$\mathcal{E}' :: \mathcal{P} \triangleright [S_1,S_2]^{R_1,R_2}_{\Sigma} \longrightarrow^{*} [S'_1,S'_2]^{R'_1,R'_2}_{(\Sigma,\Sigma'_1,\Sigma'_2)}$	by rule **sex_itn** on $\mathcal{E}'_1$ and $\mathcal{E}'_2$.

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

The reverse direction of this theorem is proved as follows: we wrap each one-step sequential firing with one application of rule $\mathbf{pex\_par}$ by means of an instance of rule $\mathbf{pex\_id}$ with empty state and active role set. The multi-step case mimics the corresponding sequential construction with parallel iteration rules. □

This result allows us to focus our efforts on the rules for sequential execution: any property proved valid for the sequential case can immediately be ported to the parallel setting by using the appropriate direction of the above theorem. This does not mean, however, that we should do without parallel execution and eliminate its rules from our formalization: this form of execution is more faithful to the behavior of protocols in a distributed environment and therefore constitutes a more adequate model of their execution. The fact that parallel execution can be serialized does not imply that the resulting sequential behavior is better and should be adopted.

6.4 Type Preservation

This section is dedicated to exploring the relationship between the simulation semantics outlined in Section 6.2 and the constraints on sensible objects provided by the typing rules in Sections 2–4 (and summarized in Appendix A.2). Our main result will be the Type Preservation Theorem 6.15, which certifies that execution preserves typability: when starting with well-typed objects, firing will always produce well-typed entities. This property is an important sanity check concerning the interaction of the typing and execution rules in our system. On the other hand, it has useful implications for actual implementations of *MSR*. It should be noted that aspects of the proof of this result concern specific constructs in our term language. Therefore, whenever extending this language for a particular application, we shall adapt the proof to the new syntax and inference rules.

The proof of the type preservation theorem makes use of a number of lemmas. Some address minor technical issue, while others are of a more general importance. We start with the Weakening Lemma, a general result that asserts that whenever a judgment is derivable, it remains so in the presence of additional assumptions in its signature and/or context.

Lemma 6.6 (*Weakening*)

Let Σ, Σ' *be a signature,* Γ *and* Γ' *typing context fragments, and* $\Gamma'' \vdash X : Y$ *a typing judgment among*

$\Gamma'' \vdash t : \tau \quad \Gamma'' \vdash \vec{t} : \vec{\tau} \quad \Gamma'' \vdash \tau \quad \Gamma'' \vdash \vec{\tau} \quad \Gamma'' \vdash P \quad \Gamma'' \vdash \mathit{lhs} \quad \Gamma'' \Vdash^{\!c} \mathit{rhs} \quad \Gamma'' \vdash r \quad \Gamma'' \vdash \rho \quad \Gamma'' \vdash \mathcal{P} \quad \Gamma'' \vdash R$

(Y is defined only for judgments with two objects on the right of the turnstile symbol.)

If $(\Sigma, \Gamma) \vdash X : Y$, *then* $(\Sigma, \Sigma', \Gamma, \Gamma') \vdash X : Y$.

Proof: This proof proceeds by induction on the structure of a derivation for the given judgment. We omit it because of its extreme simplicity. □

Our next result is quite technical: it affirms that whenever a state is well-typed, so is any substate; viceversa if a number of substates are typable with respect to a common signature, their union is typable.

Lemma 6.7 (*State Merge/Join*)

Let Σ *be a signature, and* S_1 *and* S_2 *two states. Then,* $\Sigma \vdash (S_1, S_2)$ *if and only if* $\Sigma \vdash S_1$ *and* $\Sigma \vdash S_2$.

Proof: The forward direction is a simple induction on the the structure of a derivation for the given judgment. In the reverse direction, the induction proceeds on the structure of a derivation for the judgment $\Sigma \vdash S_2$. □

The following Variable Scoping Lemma that asserts that a variable cannot occur in a context before it has been declared.

Lemma 6.8 (*Variable Scoping*)

Let Γ *and* Γ' *be contexts,* x *a variable,* τ *a type and* t *a term. If* $\Vdash^{\!c} (\Gamma, x : \tau, \Gamma')$, *then* $[t/x]\Gamma = \Gamma$ *and* $[t/x]\tau = \tau$.

Proof: Applying the context typing rules will eventually reveal the judgment $\models^{c} (\Gamma, x : \tau)$, to which only rule $\mathbf{ctp\_x}$ is applicable. Its premises are $\models^{c} \Gamma$ and $\Gamma \vdash \tau$. The conclusion of the lemma can be violated only if Γ or τ contain free occurrences of x. If this were the case, rule $\mathbf{mtp\_a}$ would eventually be applied, which entails that Γ contains a declaration for x. This would however violate our requirement that typing contexts contain at most one declaration for each variable.

A formal proof proceeds by induction on the structure of a derivation for the given judgment. □

The next lemma verifies that subtyping is invariant with respect to term substitution. This property needs to be checked over when modifying the subtyping relation.

Lemma 6.9 (*Subtyping*)

Let τ and τ' be two types. If $\tau :: \tau'$, then $[t/x]\tau :: [t/x]\tau'$ for any variable x and term τ.

Proof: This result is obtained by inspection of the subtyping rules for our language. □

The following lemma spells out the effect of compound substitutions under some assumptions about where variables occur.

Lemma 6.10 (*Compound Substitutions*)

Let Σ, t and τ be a signature, a term and a type such that $\Sigma \vdash t : \tau$ is derivable. Let moreover x and y be variables, t' be another term (possibly containing x) and, O be either a term t'', a type τ' or a tuple type $\vec{\tau}$ (possibly containing x and y free). Then the following equality holds:

$$[t/x]([t'/y]O \quad = \quad [[t/x]t'/y]([t/x]O)$$

Proof: This property is proved by induction on the structure of O and the definition of substitution for terms, types and tuple types. We will examine four representative situations: the three cases where O is a variable (x, y, or another variable z) and the case where O is a non-empty tuple type.

$\boxed{\mathbf{O = x}}$

$$\begin{array}{lcll} [t/x]([t'/y]x) & = & t & \text{by definition of substitution,} \\ & = & [[t/x]t'/y]t & \text{since } t \text{ is ground by assumption,} \\ & = & [[t/x]t'/y]([t/x]x) & \text{by definition of substitution.} \end{array}$$

$\boxed{\mathbf{O = y}}$

$$\begin{array}{lcll} [t/x]([t'/y]y) & = & [t/x]t' & \text{by definition of substitution,} \\ & = & [[t/x]t'/y]y & \text{by definition of substitution,} \\ & = & [[t/x]t'/y]([t/x]y) & \text{by definition of substitution.} \end{array}$$

$\boxed{\mathbf{O = z}}$

$$\begin{array}{lcll} [t/x]([t'/y]z) & = & z & \text{by definition of substitution,} \\ & = & [[t/x]t'/y]([t/x]z) & \text{by definition of substitution.} \end{array}$$

$\boxed{\mathbf{O} = \tau'^{(\mathbf{z})} \times \vec{\tau}}$

$$\begin{array}{lcll} [t/x]([t'/y](\tau'^{(z)} \times \vec{\tau})) & = & ([t/x]([t'/y]\tau'))^{(z)} \times ([t/x]([t'/y]\vec{\tau})) & \text{by definition of substitution,} \\ & = & ([[t/x]t'/y]([t/x]\tau'))^{(z)} \times ([[t/x]t'/y]([t/x]\vec{\tau})) & \text{by induction hypothesis,} \\ & = & [[t/x]t'/y]([t/x](\tau'^{(z)} \times \vec{\tau})) & \text{by definition of substitution.} \end{array}$$

The other possibilities are treated similarly to the last case considered. □

We now move to a fairly general result about the effect of substitutions on valid typing judgments. Whenever a judgment mentions a variable x of type τ, replacing every occurrence of x with a term of the same type maintains typability, assuming that some simple preconditions are met. The exact text of this lemma is as follows:

Lemma 6.11 (*Term Substitution*)

Let Σ and Σ' be signatures, t a term, τ a type, x a variable, Γ a context fragment, and $\Gamma' \vdash X : Y$ a typing judgment among

$$\Gamma' \vdash t' : \tau' \qquad \Gamma' \vdash \vec{t} : \vec{\tau} \qquad \Gamma' \vdash \tau' \qquad \Gamma' \vdash \vec{\tau} \qquad \Gamma' \vdash P \qquad \Gamma' \vdash \mathit{lhs} \qquad \Gamma' \Vdash^{\!x} \mathit{rhs} \qquad \Gamma' \vdash r \qquad \Gamma' \vdash \rho$$

(Y is defined only for judgments with two objects on the right of the turnstile symbol.)

If $\Sigma, \Sigma' \vdash t : \tau$, $\models^{c} (\Sigma, x : \tau, \Gamma)$ and $(\Sigma, x : \tau, \Gamma) \vdash X : Y$, then $(\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]X : [t/x]Y$.

Proof: The proof of this lemma proceeds by induction on a derivation $\mathcal{T}$ of the judgment $(\Sigma, x : \tau, \Gamma) \vdash X : Y$. Let $\mathcal{T}_t$ and $\mathcal{T}_\Gamma$ be derivations for the term and context typing judgments given in the premise of this result. We will spare the reader a detailed account of this long proof and instead concentrate on a few significant cases for the last rule applied in $\mathcal{T}$. We start with situations where rule $\mathbf{mtp\_a}$ has been applied. We must distinguish three subcases depending on whether the object being validated is a constant a declared in Σ, the variable x itself, or another variable y defined in Γ. We will then examine rules $\mathbf{mtp\_ss}$, $\mathbf{mtp\_pke}$, $\mathbf{mpt\_ext}$, and $\mathbf{rtp\_nnc}$.

$\boxed{\mathbf{mtp\_a}\ (1)}$ $\quad \mathcal{T} = \dfrac{}{(\Sigma_1, a : \tau', \Sigma_2, x : \tau, \Gamma) \vdash a : \tau'}\,\mathbf{mtp\_a}$

with $\Sigma = (\Sigma_1, a : \tau', \Sigma_2)$, $X = a$ and $Y = \tau'$.

$a = [t/x]a$	by definition of substitution,
$\tau' = [t/x]\tau'$	by the Variable Scoping Lemma 6.8 on $\mathcal{T}_\Gamma$,
$\mathcal{T}' :: (\Sigma_1, a : \tau', \Sigma_2, \Sigma', [t/x]\Gamma) \vdash a : \tau'$	by rule $\mathbf{mtp\_a}$.

$\boxed{\mathbf{mtp\_a}\ (2)}$ $\quad \mathcal{T} = \dfrac{}{(\Sigma, x : \tau, \Gamma) \vdash x : \tau}\,\mathbf{mtp\_a}$

with $X = x$ and $Y = \tau$.

$t = [t/x]x$	by definition of substitution,
$\tau = [t/x]\tau$	by the Variable Scoping Lemma 6.8 on $\mathcal{T}_\Gamma$,
$\mathcal{T}' :: (\Sigma, \Sigma', [t/x]\Gamma) \vdash t : \tau$	by the Weakening Lemma 6.6 on $\mathcal{T}_t$.

$\boxed{\mathbf{mtp\_a}\ (3)}$ $\quad \mathcal{T} = \dfrac{}{(\Sigma, x : \tau, \Gamma_1, y : \tau', \Gamma_2) \vdash y : \tau'}\,\mathbf{mtp\_a}$

with $\Gamma = (\Gamma_1, y : \tau', \Gamma_2)$ for $y \neq x$, $X = a$ and $Y = \tau'$.

$\mathcal{T}' :: (\Sigma, \Sigma', [t/x]\Gamma) \vdash y : [t/x]\tau'$	by rule $\mathbf{mtp\_a}$ and the definition of substitution.

$\boxed{\mathbf{mtp\_ss}}$ $\quad \mathcal{T} = \dfrac{\overset{\mathcal{T}_1}{(\Sigma, x : \tau, \Gamma) \vdash t' : \tau''} \qquad \overset{\mathcal{T}_2}{\tau'' :: \tau'}}{(\Sigma, x : \tau, \Gamma) \vdash t' : \tau'}\,\mathbf{mtp\_ss}$

with $X = t'$ and $Y = \tau'$.

$\mathcal{T}_1'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]t' : [t/x]\tau''$	by induction hypothesis on $\mathcal{T}_1$,
$\mathcal{T}_2'$	$:: [t/x]\tau'' :: [t/x]\tau'$	by the Subtyping Lemma 6.9 on $\mathcal{T}_2$,
$\mathcal{T}'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]t' : [t/x]\tau'$	by rule **mtp_ss** on $\mathcal{T}_1'$ and $\mathcal{T}_2'$.

mtp_pke
$$\mathcal{T} = \cfrac{\overset{\mathcal{T}_1}{(\Sigma, x:\tau, \Gamma) \vdash t' : \mathsf{msg}} \quad \overset{\mathcal{T}_2}{(\Sigma, x:\tau, \Gamma) \vdash k : \mathsf{pubK}\ A}}{(\Sigma, x:\tau, \Gamma) \vdash \{\!\{t'\}\!\}_k : \mathsf{msg}}\ \textbf{mtp\_pke}$$

with $X = \{\!\{t'\}\!\}_k$ and $Y = \mathsf{msg}$.

$\mathcal{T}_1'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]t' : \mathsf{msg}$	by induction hypothesis on $\mathcal{T}_1$,
$\mathcal{T}_2'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]k : [t/x](\mathsf{pubK}\ A)$	by induction hypothesis on $\mathcal{T}_2$,
$\mathcal{T}'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]\{\!\{t'\}\!\}_k : \mathsf{msg}$	by rule **mtp_pke** on $\mathcal{T}_1'$ and $\mathcal{T}_2'$, and definition of substitution.

mtp_ext
$$\mathcal{T} = \cfrac{\overset{\mathcal{T}_1}{(\Sigma, x:\tau, \Gamma) \vdash t' : \tau'} \quad \overset{\mathcal{T}_2}{(\Sigma, x:\tau, \Gamma) \vdash \vec{t} : [t'/y]\vec{\tau}}}{(\Sigma, x:\tau, \Gamma) \vdash (t', \vec{t}) : \tau'^{(y)} \times \vec{\tau}}\ \textbf{mtp\_ext}$$

with $X = (t', \vec{t})$ and $Y = \tau'^{(y)} \times \vec{\tau}$.

$\mathcal{T}_1'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]t' : [t/x]\tau'$	by induction hypothesis on $\mathcal{T}_1$,
$\mathcal{T}_2'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]\vec{t} : [t/x]([t'/y]\vec{\tau})$	by induction hypothesis on $\mathcal{T}_2$,
$\mathcal{T}_2'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]\vec{t} : [[t/x]t'/y]([t/x]\vec{\tau})$	by Lemma 6.10 on $\mathcal{T}_t$,
$\mathcal{T}'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x](t', \vec{t}) :$ $[t/x](\tau'^{(y)} \times \vec{\tau})$	by rule **mtp_ext** on $\mathcal{T}_1'$ and $\mathcal{T}_2'$ and definition of substitution.

rtp_nnc
$$\mathcal{T} = \cfrac{\overset{\mathcal{T}_1}{(\Sigma, x:\tau, \Gamma) \vdash \tau'} \quad \overset{\mathcal{T}_2}{(\Sigma, x:\tau, \Gamma, y:\tau') \Vdash^{\!r} rhs}}{(\Sigma, x:\tau, \Gamma) \Vdash^{\!r} \exists y : \tau'.\, rhs}\ \textbf{rtp\_nnc}$$

with $X = \exists y : \tau'.\, rhs$.

$\mathcal{T}_1'$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \vdash [t/x]\tau'$	by induction hypothesis on $\mathcal{T}_1$,
$\mathcal{T}_2'$	$:: (\Sigma, \Sigma', [t/x]\Gamma, y : [t/x]\tau') \Vdash^{\!r} [t/x]rhs$	by by induction hypothesis on $\mathcal{T}_2$,
$\mathcal{T}1$	$:: (\Sigma, \Sigma', [t/x]\Gamma) \Vdash^{\!r} [t/x](\exists y : \tau'.\, rhs)$	by rule **rtp_nnc** on $\mathcal{T}_1'$ and $\mathcal{T}_2'$, and definition of substitution,

The other possible last rules for the derivation $\mathcal{T}$ are treated similarly to the cases we just discussed. Most such rules follow the pattern seen when processing rule **mtp_pke**. □

The previous result has a counterpart regarding role state predicate parameters. We have the following lemma, where the array of judgments to consider is reduced by the limited syntactic situations in which such a parameter can occur.

Lemma 6.12 (*Role State Predicate Constant Substitution*)

Let Σ *and* Σ' *be signatures,* L_l *and* L *a role state predicate constant and variable respectively,* $\vec{\tau}$ *a tuple type,* Γ *a context fragment, and* $\Gamma' \vdash X$ *a typing judgment among*

$$\Gamma' \vdash P \qquad \Gamma' \vdash \mathit{lhs} \qquad \Gamma' \Vdash^{r} \mathit{rhs} \qquad \Gamma' \vdash r \qquad \Gamma' \vdash \rho$$

If $\vdash \Sigma, \mathsf{L}_l : \vec{\tau}, \Sigma'$, $\Vdash^{c} (\Sigma, L : \vec{\tau}, \Gamma)$ *and* $(\Sigma, L : \vec{\tau}, \Gamma) \vdash X$, *then* $(\Sigma, \mathsf{L}_l : \vec{\tau}, \Sigma', \Gamma) \vdash [\mathsf{L}_l/L]X$.

Proof: The proof of this result is similar to, but simpler than, the proof of the Term Substitution Lemma 6.11. For the sake of brevity, we refrain from expanding it. □

Our last auxiliary lemma verifies that valid executions of the right-hand side instantiation judgment respect type preservation. We have the following result:

Lemma 6.13 (*Type Preservation for Right-Hand Side Instantiation*)

Let Σ *and* Σ' *be signatures,* rhs *a rule right-hand sides, and* lhs *a predicate sequence such that*

$$(\mathit{rhs})_{\Sigma} \gg (\mathit{lhs})_{\Sigma'}$$

If the following typing judgments hold

$$\vdash \Sigma \qquad\qquad \Sigma \Vdash^{r} \mathit{rhs}$$

then the following judgments are derivable

$$1)\ \vdash \Sigma' \qquad\qquad 2)\ \Sigma' \vdash \mathit{lhs}.$$

Proof: The proof proceeds by induction on the structure of a derivation $\mathcal{E}$ for the instantiation judgment $(\mathit{rhs})_{\Sigma} \gg (\mathit{lhs})_{\Sigma'}$ and inversion on a derivation $\mathcal{T}_{\mathit{rhs}}$ for the typing judgment $\Sigma \Vdash^{r} \mathit{rhs}$. Let $\mathcal{T}_{\Sigma}$ be the given signature derivation. We verify that the property holds for the two rules that can possibly appear in this derivation.

$\boxed{\textbf{sex\_seq}}$ This situation is immediate since $\Sigma' = \Sigma$ and $\mathit{lhs} = \mathit{rhs}$.

$\boxed{\textbf{sex\_nnc}}$ $$\mathcal{E} = \cfrac{\overset{\mathcal{E}'}{([\mathsf{a}/x]\mathit{rhs})_{(\Sigma,\mathsf{a}:\tau)} \gg (\mathit{lhs})_{\Sigma'}}}{(\exists x : \tau.\, \mathit{rhs}')_{\Sigma} \gg (\mathit{lhs})_{\Sigma'}}\ \textbf{sex\_nnc}$$

with $\mathit{rhs} = \exists x : \tau.\, \mathit{rhs}'$.

$\mathcal{T}_{\tau} :: \Sigma \vdash \tau$	and
$\mathcal{T}_{\mathit{rhs}'} :: \Sigma, x : \tau \Vdash^{r} \mathit{rhs}'$	by inversion on $\mathcal{T}_{\mathit{rhs}}$ using rule **rtp_nnc**,
$\mathcal{T}_{\Sigma,\mathsf{a}} :: \vdash \Sigma, \mathsf{a} : \tau$	by rule **itp_a** on $\mathcal{T}_{\tau}$ and $\mathcal{T}_{\Sigma}$,
$\mathcal{T}_{\Sigma,x} :: \Vdash^{c} \Sigma, x : \tau$	by rule **ctp_sig** on $\mathcal{T}_{\Sigma}$ and then **ctp_x** on $\mathcal{T}_{\tau}$,
$\mathcal{T}_{\mathsf{a}} :: \Sigma, \mathsf{a} : \tau \vdash \mathsf{a} : \tau$	by rule **mtp_a**,
$\mathcal{T}'_{\mathit{rhs}'} :: \Sigma, \mathsf{a} : \tau \Vdash^{r} [\mathsf{a}/x]\mathit{rhs}'$	by the Substitution Lemma 6.11 on $\mathcal{T}_{\mathsf{a}}$, $\mathcal{T}_{\Sigma,x}$ and $\mathcal{T}_{\mathit{rhs}'}$,
$\mathcal{T}_{\Sigma'} :: \vdash \Sigma'$	and
$\mathcal{T}_{\mathit{lhs}} :: \Sigma' \vdash \mathit{lhs}$	by induction hypothesis on $\mathcal{T}_{\Sigma,\mathsf{a}}$ and $\mathcal{T}'_{\mathit{rhs}}$. □

Lemma 6.14 (*Type Preservation for Rule Application*)

Let r *be a rule ,* Σ *and* Σ' *signatures, and* S *and* S' *states such that*

$$r \rhd [S]_{\Sigma} \gg [S']_{\Sigma'}.$$

If the following typing judgments hold

$$\vdash \Sigma \qquad\qquad \Sigma \vdash r \qquad\qquad \Sigma \vdash S$$

then the following judgments are derivable

$$1)\ \vdash \Sigma' \qquad\qquad 2)\ \Sigma' \vdash r \qquad\qquad 3)\ \Sigma' \vdash S'.$$

Proof: The proof proceeds by induction on the a derivation $\mathcal{E}$ of the execution judgment $r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}$. Let $\mathcal{T}_{\Sigma}$, $\mathcal{T}_r$, and $\mathcal{T}_S$ be the given typing derivations, where the subscripts match their respective right-hand sides. We consider the following possible last rules for $\mathcal{E}$:

$$\boxed{\textbf{sex\_all}} \qquad \mathcal{E} = \cfrac{\overset{\mathcal{T}_t}{\Sigma \vdash t : \tau} \quad \overset{\mathcal{E}'}{[t/x]r' \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}}}{(\forall x : \tau.\, r') \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}}\ \textbf{sex\_all}$$

with $r = (\forall x : \tau.\, r')$.

$\mathcal{T}_{\tau} :: \Sigma \vdash \tau$	and
$\mathcal{T}_{r'} :: \Sigma, x : \tau \vdash r'$	by inversion on rule **utp_all** for $\mathcal{T}_r$,
$\mathcal{T}_{\Sigma,x} :: \vdash^{c} \Sigma, x : \tau$	by rule **ctp_x** on $\mathcal{T}_{\Sigma}$ and $\mathcal{T}_{\tau}$,
$\mathcal{T}'_r :: \Sigma \vdash [t/x]r'$	by the Substitution Lemma 6.11 on $\mathcal{T}_t$, $\mathcal{T}_{\Sigma,x}$ and $\mathcal{T}_{r'}$,
$\mathcal{T}'_{\Sigma} :: \vdash \Sigma'$	and
$\mathcal{T}'_S :: \Sigma' \vdash S'$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_{\Sigma}$, $\mathcal{T}'_r$, and $\mathcal{T}_S$,
$\mathcal{T}'_r :: \Sigma' \vdash \forall x : \tau.\, r'$	by Lemma 6.2 on $\mathcal{E}'$ and the Weakening Lemma 6.6 on $\mathcal{T}_r$.

$$\boxed{\textbf{sex\_core}} \qquad \mathcal{E} = \cfrac{\overset{\mathcal{E}'}{(rhs)_{\Sigma} \gg (lhs')_{\Sigma'}}}{(lhs \to rhs) \triangleright [S^*, lhs]_{\Sigma} \gg [S^*, lhs']_{\Sigma'}}\ \textbf{sex\_core}$$

with $r = (lhs \to rhs)$, $S = (S^*, lhs)$, and $S' = (S^*, lhs')$.

$\mathcal{T}_{S^*} :: \Sigma \vdash S^*$	and
$\mathcal{T}_{lhs} :: \Sigma \vdash lhs$	by the State Splitting Lemma 6.7 on $\mathcal{T}_S$,
$\mathcal{T}_{rhs} :: \Sigma \vdash^{r} rhs$	by inversion on rule **utp_core** for $\mathcal{T}_r$,
$\mathcal{T}'_{\Sigma} :: \vdash \Sigma'$	and
$\mathcal{T}_{lhs'} :: \Sigma' \vdash lhs'$	by the Instantiation Lemma 6.13 on $\mathcal{E}'$, $\mathcal{T}_{\Sigma}$ and $\mathcal{T}_{rhs}$,
$\mathcal{T}'_S :: \Sigma' \vdash S^*, lhs'$	by Lemma 6.6 on $\mathcal{T}_{S^*}$ and then Lemma 6.7 on $\mathcal{T}_{lhs'}$,
$\mathcal{T}'_r :: \Sigma' \vdash r$	by Lemma 6.2 on $\mathcal{E}'$ and the Weakening Lemma 6.6 on $\mathcal{T}_r$. □

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

Theorem 6.15 (*Type 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

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

If the following typing judgments hold

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

then the following judgments are derivable

$$1)\ \vdash \Sigma' \qquad\qquad 2)\ \Sigma' \vdash \mathcal{P} \qquad\qquad 3)\ \Sigma' \vdash R' \qquad\qquad 4)\ \Sigma' \vdash S'.$$

Proof: The proof proceeds by induction on the a derivation $\mathcal{E}$ of the execution judgment $\mathcal{P} \rhd [S]^{R}_{\Sigma} \longrightarrow^{(*)} [S']^{R'}_{\Sigma'}$. Let $\mathcal{T}_\Sigma$, $\mathcal{T}_\mathcal{P}$, $\mathcal{T}_R$, and $\mathcal{T}_S$ be the given typing derivations, where the subscripts match their respective right-hand sides. In the following, we will limit our analysis to the most significant cases, and sketch the proof of the simplest situations. We consider the following possible last rules for $\mathcal{E}$:

$\boxed{\textbf{sex\_arole}}$ $$\mathcal{E} = \frac{}{(\mathcal{P}^*, \rho^{\mathsf{A}}) \rhd [S]^{R}_{\Sigma} \longrightarrow [S]^{R,\rho^{\mathsf{A}}}_{\Sigma}}\ \textbf{sex\_arole}$$

with $\mathcal{P} = (\mathcal{P}^*, \rho^{\mathsf{A}})$ and $R' = (R, \rho^{\mathsf{A}})$.

Since only the active role set differs in the snapshots considered in this rule, we only need to prove the validity of point *(3)* of the theorem. This very simple proof goes as follows:

$\Sigma \ = (\Sigma_1, \mathsf{A} : \mathsf{principal}, \Sigma_2)$ and

$\mathcal{T}_\rho \ :: \ \Sigma \vdash \rho$ by inversion on rule **htp_arole** for $\mathcal{T}_\mathcal{P}$,

$\mathcal{T}'_R \ :: \ \Sigma \vdash R, \rho^{\mathsf{A}}$ by rule **atp_ext** on $\mathcal{T}_R$ and $\mathcal{T}_\rho$.

$\boxed{\textbf{sex\_rsp}}$ $$\mathcal{E} = \frac{}{\mathcal{P} \rhd [S]^{R^*,(\exists L:\vec{\tau}.\,\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S]^{R^*,([\mathsf{L}_l/L]\rho)^{\mathsf{A}}}_{\Sigma,\mathsf{L}_l:\vec{\tau}}}\ \textbf{sex\_rsp}$$

with $R = (R^*, (\exists L : \vec{\tau}.\, \rho)^{\mathsf{A}})$, $R' = (R^*, ([\mathsf{L}_l/L]\rho)^{\mathsf{A}})$ and $S' = (\Sigma, \mathsf{L}_l : \vec{\tau})$.

The changes to the resulting signature force us to prove all part of the theorem in this case.

$\mathcal{T}_{R^*} :: \Sigma \vdash R^*$ and

$\Sigma \ = (\Sigma_1, \mathsf{A} : \mathsf{principal}, \Sigma_2)$ and

$\mathcal{T}_{\vec{\tau}} \ :: \ \Sigma \vdash \vec{\tau}$ and

$\mathcal{T}_\rho \ :: \ \Sigma, L : \vec{\tau} \vdash \rho$ by inversion on rules **atp_ext** and **otp_rsp** for $\mathcal{T}_R$,

$\mathcal{T}_{\Sigma,L} :: \models \Sigma, L : \vec{\tau}$ by rule **ctp_sig** on $\mathcal{T}_\Sigma$ and then **ctp_rsp** on $\mathcal{T}_{\vec{\tau}}$,

$\mathcal{T}'_\Sigma \ :: \ \vdash \Sigma, \mathsf{L}_l : \vec{\tau}$ by rule **itp_rsp** on $\mathcal{T}_\Sigma$ and $\mathcal{T}_{\vec{\tau}}$,

$\mathcal{T}_{\rho'} \ :: \ \Sigma, \mathsf{L}_l : \vec{\tau} \vdash [\mathsf{L}_l/L]\rho$ by the Substitution Lemma 6.12 on $\mathcal{T}'_\Sigma$, $\mathcal{T}_{\Sigma,L}$ and $\mathcal{T}_\rho$,

$\mathcal{T}'_R \ :: \ \Sigma, \mathsf{L}_l : \vec{\tau} \vdash R^*, ([\mathsf{L}_l/L]\rho)^{\mathsf{A}}$ by Lemma 6.6 on $\mathcal{T}_{R^*}$ and then rule **atp_ext** on $\mathcal{T}'_\rho$,

$\mathcal{T}'_S \ :: \ \Sigma, \mathsf{L}_l : \vec{\tau} \vdash S$ by the Weakening Lemma 6.6 on $\mathcal{T}_S$,

$\mathcal{T}'_\mathcal{P} \ :: \ \Sigma, \mathsf{L}_l : \vec{\tau} \vdash \mathcal{P}$ by the Weakening Lemma 6.6 on $\mathcal{T}_\mathcal{P}$.

$\boxed{\textbf{sex\_rule}}$ $$\mathcal{E} = \frac{\begin{matrix}\mathcal{E}' \\ r \rhd [S]_\Sigma \gg [S']_{\Sigma'}\end{matrix}}{\mathcal{P} \rhd [S]^{R^*,(r,\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S']^{R^*,(\rho)^{\mathsf{A}}}_{\Sigma'}}\ \textbf{sex\_rule}$$

with $R = (R^*, (r, \rho)^{\mathsf{A}})$, $R' = (R^*, (\rho)^{\mathsf{A}})$.

This rule relies on our last lemma.

$\mathcal{T}_{R^*} :: \Sigma \vdash R^*$ and

$\Sigma \ = (\Sigma_1, \mathsf{A} : \mathsf{principal}, \Sigma_2)$ and

$\mathcal{T}_\rho \ :: \ \Sigma \vdash \rho$ and

$\mathcal{T}_r \ :: \ \Sigma \vdash r$ by inversion on rules **atp_ext** and **otp_rule** for $\mathcal{T}_R$,

$\mathcal{T}'_\Sigma \ :: \ \vdash \Sigma'$ and

$\mathcal{T}'_S \ :: \ \Sigma' \vdash S'$ by the Instantiation Lemma 6.14 on $\mathcal{E}'$, $\mathcal{T}_\Sigma$, $\mathcal{T}_r$ and $\mathcal{T}_\Sigma$,

$\mathcal{T}'_R :: \Sigma \vdash R^*, (\rho)^{\mathsf{A}}$ by Lemma 6.6 on $\mathcal{T}_{R^*}$ and $\mathcal{T}_\rho$, and then rule **atp_ext**,
$\mathcal{T}'_{\mathcal{P}} :: \Sigma' \vdash \mathcal{P}$ by Lemma 6.2 on $\mathcal{E}'$ and the Weakening Lemma 6.6 on $\mathcal{T}_{\mathcal{P}}$.

Rule **sex_grole** is treated similarly to rule **sex_all**. Rule **sex_skp** is processed along the lines of rule **sex_rsp**, although in a much simpler way. Rule **sex_dot** is immediate.

A simple inductive argument proves the cases of rules **sex_it0** and **sex_itn**, which implement the transitive closure of atomic transitions. □

By virtue of the Admissibility Theorem 6.5, a similar result clearly holds for parallel firings. For space reasons, we refrain from showing its statement.

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

These benefits of type preservation do not apply in full in the case of *MSR*. Indeed, rules **sex_grole** and **sex_all** directly invoke the typing judgment to instantiate variables with terms of the proper type. We will discuss operational improvements to these rules in Section 7, and show that in many situations, type correctness is guaranteed. We will however also see that there are circumstances where omitting a run-time type-check would result in violations of the Type Preservation Theorem.

6.5 Data Access Specification Preservation

This section explores the relationship between execution and data access specification. Our main achievement will be a preservation result for data access specification (Theorem 6.19): it states that, under reasonable typing assumptions, no execution sequence can take a snapshot that satisfies the data access specification policy to a situation that violates it. As in the case of typing, this property provides an important sanity check about the correctness of our data access specification rules. It also implies that it is sufficient to verify that the specification of a protocol satisfies the data access specification policy once before any execution is undertaken.

In order to establish this property, we will rely on the same techniques already used for the Type Preservation Theorem 6.15. In particular, we need to prove a number of auxiliary properties, that are the data access specification equivalents of some of the lemmas given in Section 6.4. We start with the following Weakening Lemma:

Lemma 6.16 (*Weakening*)

Let Σ and Σ' be signatures, Γ a typing context fragment, Δ a knowledge context, A a principal name or variable, and $\Gamma'; \Delta \Vdash_A X > Y \gg Z$ an data access specification judgment among

$$\begin{array}{ccccc} \Gamma'; \Delta \Vdash^{s}_A k \gg \Delta' & \Gamma'; \Delta \Vdash^{a}_A k \gg \Delta' & \Gamma'; \Delta \Vdash_A \vec{t} \gg \Delta' & \Gamma'; \Delta \Vdash_A \mathit{lhs} > \vec{t} \gg \Delta' \\ \Gamma' \looparrowright_A e & \Gamma; \Delta \looparrowright_A t' & \Gamma; \Delta \looparrowright_A \vec{t} & \Gamma'; \Delta \looparrowright_A \mathit{lhs} & \Gamma'; \Delta \Vdash_A \mathit{rhs} \\ \Gamma' \Vdash_A r & \Gamma' \Vdash_A \rho & \Sigma'' \Vdash \mathcal{P} & \Sigma'' \Vdash R \end{array}$$

(Y and Z are defined only for judgments with more than two (resp. three) objects to the right of the turnstile symbol; A and Δ do not appear in the last two judgments..)

If $(\Sigma, \Gamma); \Delta \Vdash_A X > Y \gg Z$, *then* $(\Sigma, \Sigma', \Gamma); \Delta \Vdash_A X > Y \gg Z$.

Proof: This routine proof proceeds by induction on the structure of a derivation of the given data access specification judgment. We will spare the reader its long and tedious development. □

The next auxiliary result we need is the data access specification equivalent of the Term Substitution Lemma 6.11 proved in the previous section. The statement of this property is rather involved in an attempt to provide a succinct characterization for the numerous judgments participating in data access specification. It should be noticed that its validity makes use of typing assumptions, used to support the informal requirement made in section 5 that all objects appearing in our data access specification rules should be well-typed.

Lemma 6.17 (*Term Substitution*)

Let Σ *and* Σ' *be signatures,* t *a term,* τ *a type,* x *a variable, and* Γ *a typing context fragment such that the judgments* $\Sigma, \Sigma' \vdash t : \tau$ *and* $\Vdash^{c} (\Sigma, x : \tau, \Gamma)$ *hold. Furthermore,*

- *let* Δ *be a knowledge context,* A *a principal constant or variable, and* $\Gamma'; \Delta \Vdash_A X > Y \gg Z$ *an data access specification judgment among*

$$\Gamma'; \Delta \Vdash^{s}_{A} k \gg \Delta' \qquad \Gamma'; \Delta \Vdash^{a}_{A} k \gg \Delta' \qquad \Gamma'; \Delta \Vdash_{A} \vec{t} \gg \Delta' \qquad \Gamma'; \Delta \Vdash_{A} lhs > \vec{t} \gg \Delta'$$

$$\Gamma' \looparrowright_{A} e \qquad \Gamma; \Delta \looparrowright_{A} t' \qquad \Gamma; \Delta \looparrowright_{A} \vec{t} \qquad \Gamma'; \Delta \looparrowright_{A} lhs \qquad \Gamma'; \Delta \Vdash_{A} rhs$$

$$\Gamma' \Vdash_{A} r \qquad \Gamma' \Vdash_{A} \rho$$

  (Y *and* Z *are defined only for judgments with more than two (resp. three) objects to the right of the turnstile symbol.)*

  If $(\Sigma, x : \tau, \Gamma); \Delta \Vdash_A X > Y \gg Z$, *then* $(\Sigma, \Sigma', [t/x]\Gamma); [t/x]\Delta \Vdash_{[t/x]A} [t/x]X > [t/x]Y \gg [t/x]Z$

- *let* Δ *and* Δ' *be knowledge contexts and* $\vec{t}$ *a tuple consisting of either ground terms or variables.*

  If $\Delta > \vec{t} > \Delta'$, *then* $[t/x]\Delta > [t/x]\vec{t} > [t/x]\Delta'$

Proof: This proof proceeds by induction on the structure of a derivation $\mathcal{A}$ for the given data access specification judgments. Let $\mathcal{T}_t$ and $\mathcal{T}_\Gamma$ be derivations for the assumed term and context typing judgments, respectively. The number of cases to examine is considerable due to the many rules that implement data access specification, and to the fact that some of these rules require a detailed analysis of several subcases. For the sake of brevity, we will describe the treatment of three significant rules: **kac_su1**, **lac_rsp**, and **rac_nnc**. The first of these has three subcases.

$\boxed{\textbf{kac\_su1}\ (1)}$ $$\mathcal{A} = \frac{}{(\Sigma, x : \tau, \Gamma_1, k : \mathsf{shK}\ A\ B, \Gamma_2); \Delta \Vdash^{s}_{A} k \gg (\Delta, k)}\ \textbf{kac\_su1}$$

with $\Gamma = (\Gamma_1, k : \mathsf{shK}\ A\ B, \Gamma_2)$ $X = k$, and $Y = (\Delta, k)$

$\mathcal{A}_x ::\ (\Sigma, \Sigma', [t/x]\Gamma_1, k : \mathsf{shK}\ ([t/x]A)\ ([t/x]B), [t/x]\Gamma_2); [t/x]\Delta \Vdash^{s}_{[t/x]A} [t/x]k \gg ([t/x]\Delta, [t/x]k)$ by rule **kac_su1**,

$\mathcal{A}_x ::\ (\Sigma, \Sigma', [t/x](\Gamma_1, k : \mathsf{shK}\ A\ B, \Gamma_2)); [t/x]\Delta \Vdash^{s}_{[t/x]A} [t/x]k \gg [t/x](\Delta, k)$ by definition of substitution.

$\boxed{\textbf{kac\_su1}\ (2)}$ $$\mathcal{A} = \frac{}{(\Sigma, x : \mathsf{shK}\ A\ B, \Gamma); \Delta \Vdash^{s}_{A} x \gg (\Delta, x)}\ \textbf{kac\_su1}$$

with $\tau = \mathsf{shK}\ A\ B$ $X = x$, and $Y = (\Delta, x)$

$\Sigma, \Sigma' = (\Sigma_1, t : \mathsf{shK}\ A\ B, \Sigma_2)$	by inversion on rule **mpt_a** for $\mathcal{T}_t$,
$A = [t/x]A$	and
$\Sigma_2 = [t/x]\Sigma_2$	by the Variable Scoping Lemma 6.8 on $\mathcal{T}_\Gamma$,
$\mathcal{A}' ::\ (\Sigma_1, t : \mathsf{shK}\ A\ B, \Sigma_2, [t/x]\Gamma); [t/x]\Delta \Vdash^{s}_{A} t \gg ([t/x]\Delta, t)$	by rule **kac_su1** and definition of substitution.

$\boxed{\textbf{kac\_su1}\ (3)}$ $$\mathcal{A} = \frac{}{(\Sigma_1, k : \mathsf{shK}\ A\ B, \Sigma_2, x : \tau, \Gamma); \Delta \Vdash^{s}_{A} k \gg (\Delta, k)}\ \textbf{kac\_su1}$$

with $\Sigma = (\Sigma_1, k : \mathsf{shK}\ A\ B, \Sigma_2)$ $X = k$, and $Y = (\Delta, k)$

This last subcase results from a combination of the reasoning performed in the last two situations.

$$\boxed{\textbf{lac\_rsp}} \qquad \mathcal{A} = \cfrac{\overset{\mathcal{A}_1}{\Delta > (A,\vec{e}) > \Delta'} \qquad \overset{\mathcal{A}_2}{(\Sigma, x:\tau,\Gamma);\Delta' \Vdash_A \mathit{lhs} > \vec{t}' \gg \Delta''}}{(\Sigma, x:\tau,\Gamma);\Delta \Vdash_A (L(A,\vec{e}),\mathit{lhs}) > \vec{t}' \gg \Delta''}\ \textbf{lac\_rsp}$$

with $X = (L(A,\vec{e}),\mathit{lhs})$, $Y = \vec{t}$, and $Z = \Delta''$

$\mathcal{A}'_1 :: [t/x]\Delta > [t/x](A,\vec{e}) > [t/x]\Delta'$ by induction hypothesis on $\mathcal{A}_1$,
$\mathcal{A}'_2 :: (\Sigma,\Sigma',[t/x]\Gamma);[t/x]\Delta' \Vdash_{[t/x]A} [t/x]\mathit{lhs} > [t/x]\vec{t}' \gg [t/x]\Delta''$ by induction hypothesis on $\mathcal{A}_2$,
$\mathcal{A}' :: (\Sigma,\Sigma',[t/x]\Gamma);[t/x]\Delta \Vdash_{[t/x]A} [t/x](L(A,\vec{e}),\mathit{lhs}) > [t/x]\vec{t}' \gg [t/x]\Delta''$ by rule **rac_nnc** on $\mathcal{A}'_1$ and $\mathcal{A}'_2$ and definition of substitution.

$$\boxed{\textbf{rac\_nnc}} \qquad \mathcal{A} = \cfrac{\overset{\mathcal{A}_{rhs}}{(\Sigma, x:\tau,\Gamma, y:\tau');(\Delta,y) \Vdash_A \mathit{rhs}}}{(\Sigma, x:\tau,\Gamma);\Delta \Vdash_A \exists y:\tau'.\,\mathit{rhs}}\ \textbf{rac\_nnc}$$

with $X = (\exists y:\tau'.\,\mathit{rhs})$

$\mathcal{A}'_{rhs} :: (\Sigma,\Sigma',[t/x](\Gamma,y:\tau'));[t/x](\Delta,y) \Vdash_{[t/x]A} [t/x]\mathit{rhs}$ by induction hypothesis on $\mathcal{A}_{rhs}$,
$\mathcal{A}' :: (\Sigma,\Sigma',[t/x]\Gamma);[t/x]\Delta \Vdash_{[t/x]A} [t/x](\exists y:\tau'.\,\mathit{rhs})$ by rule **rac_nnc** on $\mathcal{A}'_{rhs}$ and definition of substitution.

The remaining rules are treated similarly and will not be discussed further. □

We need a similar result in order to make substitutions to role state predicate parameters. This property is formally captured in the following lemma:

Lemma 6.18 (*Role State Predicate Constant Substitution*)

Let Σ and Σ' be signatures, L_l and L a role state predicate constant and variable respectively, $\vec{\tau}$ a tuple type, Γ a typing context fragment, Δ a knowledge context, and $\Gamma';\Delta \Vdash_A X > Y \gg Z$ an data access specification judgment among

$$\Gamma';\Delta \Vdash_A \mathit{lhs} > \vec{t} \gg \Delta' \qquad \Gamma';\Delta \looparrowright_A \mathit{lhs} \qquad \Gamma';\Delta \Vdash_A \mathit{rhs} \qquad \Gamma' \Vdash_A r \qquad \Gamma' \Vdash_A \rho$$

(Y and Z are defined only in the first of these judgments; Δ does not appear.)

If $\vdash \Sigma, \mathsf{L}_l : \vec{\tau}, \Sigma'$, $\vdash (\Sigma, L:\vec{\tau},\Gamma)$ and $(\Sigma, L:\vec{\tau},\Gamma);\Delta \Vdash_A X > Y \gg Z$, then $(\Sigma,\Sigma',\Gamma);\Delta \Vdash_A [\mathsf{L}_l/L]X > Y \gg Z$.

Proof: This result is proved similarly to the Term Substitution Lemma 6.17. There are fewer cases to consider since role state predicate parameters can occur only in a limited number of syntactic constructions. □

At this point, we have all the necessary tools to produce a direct proof of the main result of this section. The Data Access Specification Preservation Theorem below states that, given a protocol theory and an active role set that satisfies data access specification (and are well-typed) in the current signature, every execution sequence will produce a snapshot whose active role state is compliant with the data access specification policy.

Theorem 6.19 (*Data Access Specification 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

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

If the following data access specification judgments hold

$$\Sigma \Vdash \mathcal{P} \qquad \Sigma \Vdash R$$

then the following judgments are derivable

1) $\Sigma' \Vdash \mathcal{P}$ *2)* $\Sigma' \Vdash R'$

Proof: To carry out this proof, it is convenient to use a presentation of the execution rules that bypasses the rule application judgment $r \rhd [S]_\Sigma \gg [S']_{\Sigma'}$. To do so, we simply replace rules **sex_all** and **sex_core** from Section 6.2 with the following variants:

$$\frac{\Sigma \vdash t : \tau}{\mathcal{P} \rhd [S]_\Sigma^{R,((\forall x:\tau.\, r,)\rho)^{\mathsf{A}}} \longrightarrow [S]_\Sigma^{R,(([t/x]r),\rho)^{\mathsf{A}}}}\,\textbf{sex\_all}' \qquad \frac{(rhs)_\Sigma \gg (lhs')_{\Sigma'}}{\mathcal{P} \rhd [S, lhs]_\Sigma^{R,((lhs \to rhs),\rho)^{\mathsf{A}}} \longrightarrow [S, lhs']_{\Sigma'}^{R,(\rho)^{\mathsf{A}}}}\,\textbf{sex\_core}'$$

Proving that these two rules are derivable in our original system and that rules **sex_all** and **sex_core** are admissible in their presence is left as an easy exercise to the reader.

With this in place, the proof of the desired result is similar to the proof of the Type Preservation Theorem 6.15. We will limit its illustration to the cases where the last rule applied in the given execution derivation, say $\mathcal{E}$, is either **sex_all** or **sex_core**. We will use the names $\mathcal{T}_\Sigma$, $\mathcal{T}_\mathcal{P}$, $\mathcal{T}_R$, $\mathcal{A}_\mathcal{P}$, and $\mathcal{A}_R$, for the assumed derivations of the other judgments, in the order they are given.

$$\boxed{\textbf{sex\_all}} \quad \mathcal{E} = \frac{\begin{array}{c}\mathcal{T}_t \\ \Sigma \vdash t : \tau\end{array}}{\mathcal{P} \rhd [S]_\Sigma^{R^*,((\forall x:\tau.\, r,)\rho)^{\mathsf{A}}} \longrightarrow [S]_\Sigma^{R^*,(([t/x]r),\rho)^{\mathsf{A}}}}\,\textbf{sex\_all}$$

with $R = (R^*, ((\forall x : \tau.\, r), \rho)^{\mathsf{A}})$ and $R' = (R^*, (([t/x]r), \rho)^{\mathsf{A}})$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R^*$	and
$\mathcal{A}_\rho :: \Sigma \Vdash_{\mathsf{A}} \rho$	and
$\mathcal{A}_r :: \Sigma, x : \tau \Vdash_{\mathsf{A}} r$	by inversion on rules **aac_ext**, **oac_rule**, and **uac_all** for $\mathcal{A}_R$,
$\mathcal{T}_\tau :: \Sigma \vdash \tau$	by inversion on rules **atp_ext**, **otp_rule**, and **utp_all** for $\mathcal{T}_R$,
$\mathcal{T}_{\Sigma,x} :: \models^{\mathsf{c}} \Sigma, x : \tau$	by rule **ctp_x** on $\mathcal{T}_\Sigma$ and $\mathcal{T}_\tau$,
$\mathcal{A}'_r :: \Sigma \Vdash_{\mathsf{A}} [t/x]r$	by the Substitution Lemma 6.17 on $\mathcal{T}_t$, $\mathcal{T}_{\Sigma,x}$ and $\mathcal{A}_r$,
$\mathcal{A}'_R :: \Sigma \Vdash_{\mathsf{A}} R^*, (([t/x]r), \rho)^{\mathsf{A}}$	by rules **oac_rule** and **aac_ext** on $\mathcal{A}_{R^*}$ and $\mathcal{A}'_r$.

$$\boxed{\textbf{sex\_core}} \quad \mathcal{E} = \frac{\begin{array}{c}\mathcal{E}' \\ (rhs)_\Sigma \gg (lhs')_{\Sigma'}\end{array}}{\mathcal{P} \rhd [S^*, lhs]_\Sigma^{R^*,((lhs \to rhs),\rho)^{\mathsf{A}}} \longrightarrow [S^*, lhs']_{\Sigma'}^{R^*,(\rho)^{\mathsf{A}}}}\,\textbf{sex\_core}$$

with $S = (S^*, lhs)$, $R = (R^*, ((lhs \to rhs), \rho)^{\mathsf{A}})$, $S' = (S^*, lhs')$ and $R' = (R^*, (\rho)^{\mathsf{A}})$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R^*$	and
$\mathcal{A}_\rho :: \Sigma \Vdash_{\mathsf{A}} \rho$	by inversion on rules **aac_ext**, **oac_rule** and **uac_core** for $\mathcal{A}_R$,
$\mathcal{A}'_R :: \Sigma' \Vdash R^*, (\rho)^{\mathsf{A}}$	by Lemma 6.16 on $\mathcal{A}_{R^*}$ and $\mathcal{A}_\rho$ and then rule **aac_ext**,
$\mathcal{A}'_\mathcal{P} :: \Sigma' \Vdash \mathcal{P}$	by the Weakening Lemma 6.16 on $\mathcal{A}_\mathcal{P}$.

The treatment of the other possible final steps for $\mathcal{E}$ is adapted from the proof of the Type Preservation Theorem 6.15. In particular, rules **sex_grole** and **sex_rsp** make use of the Substitution Lemmas 6.17 and 6.18, respectively. The other cases are immediate. □

Thanks to the Admissibility Theorem 6.5, this property can be extended to the parallel firing judgments. For space reasons, we omit further developments.

This theorem and the fact that the execution rules do not depend on any data access specification judgment makes data access specification verification a purely static check, that need to be perform only once before any execution is undertaken. Additional aspects, mostly related to implementation issues, will be touched in Section 7.

6.6 Changes

As done at the end of the previous major subdivisions of this report, we conclude this section with a discussions of the changes in the execution semantics of *MSR* with respect to previous versions of this formalism, as presented in [27, 30].

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

2. In [27, 30], the universal variables (then implicitly quantified) were instantiated by pattern matching at the time a rule was applied. Our current model is apparently more non-deterministic in that it relies on some guesswork to instantiated these objects before the current state is accessed. As already said, this model is purposefully abstract and not intended as the basis for an implementation. We will discuss a more concrete proposal in Section 7. We should however observe that [27, 30] did not discuss how variables that appear only in the consequent of a rule ought to be instantiated, while our current proposal treats them transparently.

3. As mentioned in Section 4, our earlier work on *MSR* demanded that the graph obtained by chasing the role state predicate names in a specification be acyclic. This eliminated the possibility of roles executions that required an unbounded number of steps, a major simplification in the proofs of the decidability results of [27, 46]. Our present syntax admits roles where the corresponding graphs contain cycles. It should however be noted that our execution model prevents taking advantage of them to implement a looping behavior: parametricity makes role state predicates unique to each individual invocation of a role. The rules in an active role, which realize this property, are however processed sequentially without any possibility of jumping back. Still, iteration is possible in *MSR* thanks to the introduction of memory predicates. This will be demonstrated in Section 9.

4. In the previous versions of *MSR*, the notion of execution was limited to sequential firing of rules. Parallel execution is therefore a novelty of this report.

7 Pragmatics

The given typing, data access specification and execution semantics for *MSR* provide a foundation for proving useful properties about this language, as we did in Section 6. However, they are not optimal either in terms of implementability or usability. In this section, we briefly examine three pragmatic enhancements, none of which is a prerequisite for or otherwise impacts the validity of the results examined in this document. In Section 7.1, we discuss opportunities for type reconstruction and outline an approach to realizing it. In Section 7.2, we examine the basis for an implementation strategy that relies on matching for selecting appropriate terms to instantiate universal variables with. Finally, we present a simple check that verifies that an active role set effectively came from the protocol theory at hand in Section 7.3.

7.1 Type Reconstruction

Each constant occurring in an *MSR* protocol is to be interpreted as having been introduced by a declaration in a dependent prefix, or in a role (as the leading $\forall$ or through $\exists$), or inside the rule itself as a $\forall$ or $\exists$ declaration. As a matter of readability, convenience to the user and usability, a number of declarations and typing information can be omitted and automatically reconstructed from the context in which variables are used and from a limited number of type declarations. An *MSR* implementation is expected to either reconstruct this omitted data, when this can be done unambiguously, or issue detailed error messages so that the user can add sufficient text to help reconstruct the remaining omitted parts. We will now examine these opportunities in more detail, although we will not display rules to achieve this. A detailed account of type reconstruction in the presence of dependent types can be found in [56].

The acceptable omissions fall into the following two classes:

- Implicit bindings comprise the $\forall$ declarations at the head of a rule and the dependent prefixes of a type or subtype.
- Type part of a declaration.

The omitted dependent prefixes of a type declaration also lead to the omission of the corresponding parameters when these symbols are used.

Reconstruction of Universal Bindings

The binders $\forall x : \tau$ occurring in a rule can be omitted as long as the variable x is not explicitly bound by another declaration and the type τ can be reconstructed. Upon encountering an undeclared symbol x, the compiler shall extend the rule it occurs in with the prefix $\forall x$ (the still implicit type is reconstructed at a later stage, or a type variable α, to be instantiated subsequently, can be inserted).

The reconstruction algorithms outlined in this section apply only within a rule, and not elsewhere. It shall also be noted that, in general, this simple syntactic binder reconstruction needs to be alternated with the more complex reconstruction of omitted types, as an omitted type may make use of a yet undeclared variable.

Reconstruction of Dependent Prefixes

Every symbols occurring in a base type should be declared before its first use. However, since many of these declarations are bookkeeping dependent prefixes of the type or subtype they appear in, it is convenient to omit them whenever they can be reconstructed. We adopt the same strategy as in the case of omitted $\forall x : \tau$ declarations. The missing dependent type binder for symbol x is added at the front of the type or subtype as an object declaration $\alpha^{(x)}$ with its type temporarily expressed as the type variable α. If only the variable declaration (but not the type or subtype) has been omitted, we extend a type τ into $\tau^{(x)}$ and a subtype σ into $\sigma^{(x)}$.

Whenever an undeclared symbol occurs in a type within a rule, it may need to be expanded as either a $\forall$ declaration or a dependent prefix. In general, the latter is preferred if it occurs within a single type after all the reconstruction steps have been performed. A $\forall$ declaration is necessary only when this symbol occurs in two different types. Other considerations are as in the case of implicit $\forall$ declarations.

Reconstruction of Omitted Types

Object declarations "$x : \tau$" can be abbreviated as "x" whenever the type τ can be reconstructed from the context. Type reconstruction information are obtained from the following sources:

Previous declarations. For example, if a signature contains a declaration $f : \tau \to \tau'$ for a function symbol f, and a type $\_^{(x)} \times (\ldots f x \ldots)$ where the type of x has been omitted as $\_$, then it can be deduced that the type of x shall be τ.

Data Access Specification information. Data access information imposes constraints on the identity of principals or the ownership of keys. Often times, only one value allows a rule to pass the data access specification test. In these cases, it is admissible to omit typing information, even if reconstruction by the sole means of the surrounding declarations is not possible.

Type reconstruction is usually done by inserting type variables in place of the omitted types, and collecting and propagating constraints until a single solution emerges (*success*), the constraints are found to be inconsistent (*fatal error*), or all possible constraint simplifications have been attempted but constraints remain (*underspecification error*). In the latter case, the author of the specification shall be invited to add typing information and run the checker again.

Subtypes

Whenever a type subject to reconstruction belongs to a subtype hierarchy, the result shall be the lower bound of all types occurring in the constraints, subject to the usual variance and contravariance conditions. For example, given a subtype declaration $\tau :: \tau'$ and a variable whose type can be reconstructed as either τ or τ', τ shall be preferred. If this lower bound is not unique, then an underspecification error shall be returned.

Implicit Arguments

Whenever an identifier declaration relies on omitted typing information, any use of this identifier shall omit all arguments corresponding to this omitted typing information. For example, if c is declared of type $a\ x$ (by means of the declaration $c : a\ x$) where the type of x is kept implicit, then every use of c shall take the form $c\ t$ for term t of type $a\ t'$ for an appropriate term t'. If instead c is declared of type $\tau^{(x)} \times a\ x$ (by means of the declaration $c : \tau^{(x)} \times a\ x$) with the type of x given explicitly, then every use of c can only be of the form $c\ (t', t)$ where t' has type τ and t has type $a\ t'$.

During reconstruction, these omitted arguments shall be reconstructed as well.

7.2 Lazy instantiation

In Section 6, the execution rules concerning a universal quantifier, **sex_all** and **sex_grole**, operate by guessing an appropriate term t or principal name A and then substituting it for the bound variable. This is clearly not an effective approach to instantiating universal variables.

In logic programming, where a similar form of instantiation needs to be performed in order to use a clause, the choice of the instantiating term is delayed until later execution forces it. This is achieved by replacing universally quantified variables with *logical variables* when rules such as **sex_all** and **sex_grole** are encountered, and by using unification to instantiate them when the execution needs to commit to specific terms — matching is sufficient in the case of *MSR*. Logical variables form a new syntactic category in the internal language used during execution. They are defined as follows:

$$\textit{Logical variable:} \quad X \quad ::= \quad X_\Sigma^\tau$$

Each logical variable X_Σ^τ is parametrized by its expected type, τ, and a signature Σ that lists the signature constants that can legally be used in a substitution term for this variable. We write X for a logical variable when the type and signature are unimportant or can easily be reconstructed from the context. Note that logical variables are distinct from the variables we have encountered so far.

During execution, we will instantiate logical variables to ground terms by matching rule left-hand sides (where logical variables may occur) with states (which shall be ground). A successful match is witnessed by a substitution, written θ, which associates each logical variable in the rule with a ground term. Applying a substitution θ to an object O (a term, rule fragment or role) produces an object O' where each logical variable mentioned in θ has been replaced with the associated ground term. We write $[\theta]O$ for this operation.

Given these premises, the following rules upgrade the operational semantics given in Section 6 to make use of logical variables and matching. For convenience, we use a formulation that bypasses the rule application judgment (see the proof of Theorem 6.19).

$$\frac{}{(\mathcal{P}, \rho^{\mathsf{A}}) \rhd [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,\rho^{\mathsf{A}}}}\ \textbf{sex\_arole}{*} \qquad \frac{}{(\mathcal{P}, (\rho)^{\forall A}) \rhd [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,[X_\Sigma^{\mathsf{principal}}/A](\rho^{A})}}\ \textbf{sex\_grole}{*}$$

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

$$\frac{([\theta]rhs)_{[\theta]\Sigma} \gg (rhs')_{\Sigma'}}{\mathcal{P} \triangleright [S, [\theta]lhs]^{R,((lhs\rightarrow rhs),\rho)^A}_{\Sigma} \longrightarrow [S, rhs']^{R,([\theta]\rho)^{[\theta]A}}_{\Sigma'}}\ \textbf{sex\_core}*$$

$$\frac{}{\mathcal{P} \triangleright [S]^{R,(r,\rho)^A}_{\Sigma} \longrightarrow [S]^{R,(\rho)^A}_{(\Sigma,\mathsf{L}_l:\vec{\tau})}}\ \textbf{sex\_skp}* \qquad \frac{}{\mathcal{P} \triangleright [S]^{R,(\cdot)^A}_{\Sigma} \longrightarrow [S]^{R}_{\Sigma}}\ \textbf{sex\_dot}*$$

As mentioned earlier, rules **sex_all**$*$ and **sex_grole**$*$, which handle universal quantifiers, now replace the bound variables with a new logical variable of the appropriate type and in the current signature. The interesting action takes place in rule **sex_core**$*$. The rule being applied has the form $lhs \rightarrow rhs$. To use it, we check if the current state, call it S^*, can be partitioned into a part S that will be left alone and a part S' that matches lhs by means of some substitution θ, i.e., $S' = [\theta]lhs$. If so, we invoke the right-hand side instantiation judgment on the corresponding instantiation of rhs and of the signature Σ, thereby obtaining some partial state rhs' and updated signature Σ'. We then replace $[\theta]lhs$ in S^* with rhs' and propagate the substitution θ to the remaining rules in the current active roles set and to the role owner. Other rules do not change, in particular logical variables play no role in handling existential quantifiers.

It should be noted that a signature Σ can contain logical variables. Moreover, the notation $(S, [\theta]lhs)$ calls for pattern matching of lhs with the state at hands, which will yield the substitution θ. Pattern matching is done using standard procedures, but, because we are in a dependent setting, every match t found for each variable X^{τ}_{Σ} should be type-checked so that $\Sigma \vdash t : \tau$.

7.3 Legal Active Roles

As we saw, a snapshot of the execution of a protocol theory $\mathcal{P}$ consists of a state S, a signature Σ, and an active role set R. A natural question to answer is whether R is actually an instance of $\mathcal{P}$, in the sense described above. We model it by means of the judgments:

$\Sigma \vdash \mathcal{P} \lhd \rho^{\mathsf{A}}$	ρ^{A} *stems from* $\mathcal{P}$ *in* Σ
$\Sigma \vdash \rho'^{\alpha} \lhd \rho^{\mathsf{A}}$	ρ^{A} *stems from* ρ'^{α} *in* Σ
$\Sigma \vdash \rho' \lhd \rho$	ρ *stems from* ρ' *in* Γ
$\Gamma \vdash \theta$	θ *is well-formed in* Σ

The first simply checks that active role set element ρ^{A} comes from protocol theory $\mathcal{P}$ by picking an appropriate role in there. It is defined by the following two rules:

$$\frac{\Sigma \vdash \rho' \lhd \rho^{\mathsf{A}}}{\Sigma \vdash (\mathcal{P}, \rho') \lhd \rho^{\mathsf{A}}} \qquad \frac{\Sigma \vdash \mathcal{P} \lhd \rho^{\mathsf{A}}}{\Sigma \vdash (\mathcal{P}, \rho') \lhd \rho^{\mathsf{A}}}$$

The second judgment makes sure that the owner of ρ and ρ' corresponds. This is achieved through instantiation in the case of a generic role, as expressed by the following rules.

$$\frac{(\Sigma, \mathsf{A} : \mathsf{principal}, \Sigma') \vdash [\mathsf{A}/A]\rho' \lhd \rho}{(\Sigma, \mathsf{A} : \mathsf{principal}, \Sigma') \vdash (\rho')^{\forall A} \lhd \rho^{\mathsf{A}}} \qquad \frac{(\Sigma, \mathsf{A} : \mathsf{principal}, \Sigma') \vdash \rho' \lhd \rho}{(\Sigma, \mathsf{A} : \mathsf{principal}, \Sigma') \vdash (\rho')^{\mathsf{A}} \lhd \rho^{\mathsf{A}}}$$

The bulk of the work is done by the third judgment. It follows the structure of the candidate role ρ' by inserting quantified variables in the context and possibly skipping rules until it matches the target role ρ. The rightmost rule below handles this case by finding an appropriate substitution θ and ensuring that $[\theta]\rho'$ coincides with ρ up to the renaming of bound variables. This is expressed by the premise $[\theta]\rho' =_{\alpha} \rho$ — we do not show the rules for this standard notion.

$$\frac{(\Gamma, L : \vec{\tau}) \vdash \rho' \lhd \rho}{\Gamma \vdash (\exists L : \vec{\tau}.\, \rho') \lhd \rho} \qquad \frac{(\Gamma, x : \tau) \vdash (r, \rho') \lhd \rho}{\Gamma \vdash ((\forall x : \tau.\, r), \rho') \lhd \rho} \qquad \frac{\Gamma \vdash \rho' \lhd \rho}{\Gamma \vdash (r, \rho') \lhd \rho} \qquad \frac{\Gamma \vdash \theta \quad [\theta]\rho' =_{\alpha} \rho}{\Gamma \vdash \rho' \lhd \rho}$$

The judgment $\Gamma \vdash \theta$ appearing in the premise of this rule checks that a substitution is well-formed in a signature. It is defined as follows:

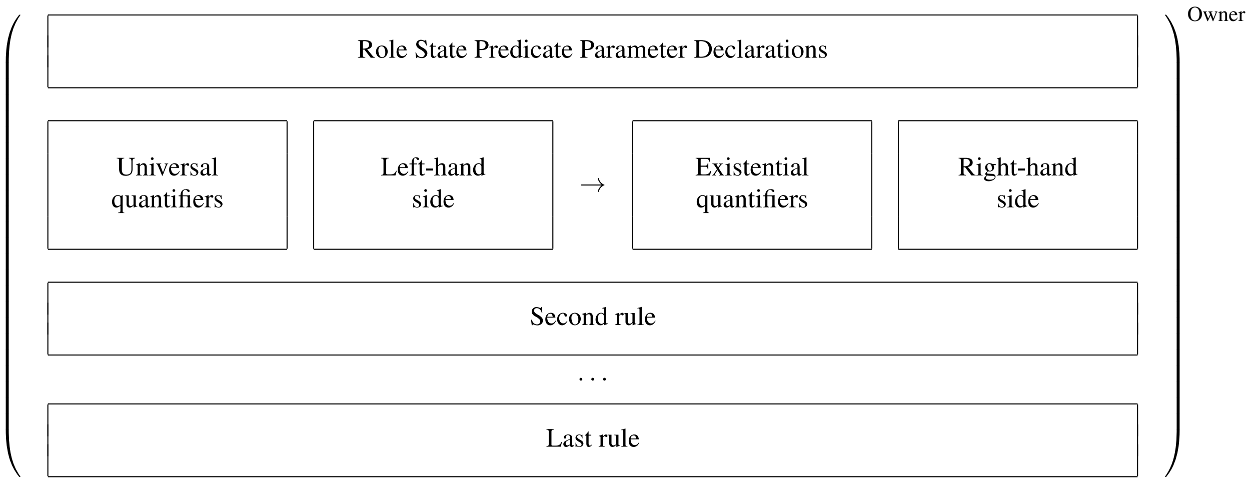
$$\frac{}{\Sigma \vdash \cdot} \qquad \frac{\Gamma \vdash \theta \quad \Sigma \vdash \tau \quad \Sigma \vdash t : \tau}{(\Sigma, \Gamma, x : \tau) \vdash \theta, t/x} \qquad \frac{(\Sigma, \mathsf{L}_l : \vec{\tau}, \Gamma) \vdash \theta}{(\Sigma, \mathsf{L}_l : \vec{\tau}, \Gamma, L : \vec{\tau}) \vdash \theta, \mathsf{L}_l/L}$$

Notice that these judgments implement a necessary condition for an active role to come from a protocol theory, but not a sufficient one since the occurrence constraints on fresh data placed by the existential quantifiers may not be met. In order to guarantee this, we would essentially need to undo the computation from the target state to an initial state.

8 Intruder

In this section, we turn to the *MSR* specification of a potential attacker, against which we may want to analyze a given protocol. We start in Section 8.1 by presenting an *MSR* formalization of what has come to be accepted as the standard abstraction of the attacker: the Dolev-Yao intruder [52, 44]. This will assume the form of a suite of roles anchored on a specific principal, the intruder that we denote I, that altogether capture the capabilities usually associated with this attacker. Our strongly typed framework will shed some light on what these capabilities actually are, and we will show that they are closely related to the notion of data access specification defined in Section 5. Since *MSR* allows expressing the intruder as any other role, we can write arbitrary complex specifications for I. In Section 8.2, we show that any such roles can be emulated by the protocol suite introduced in 8.1. Therefore, the Dolev-Yao intruder is the most powerful attacker.

It will be convenient to format roles as in the following diagram:



The header lists the roles state predicate declarations used by the role. It is followed by the rules that constitute it (we will never need to alternate declarations and rules). Each rules is represented by the four blocks in the diagram. We find it convenient to list their element in columns. Whenever a role consists of more than one rule, the universal quantifier prefix grows monotonically when passing from one rule to the next. We acknowledge this fact by abbreviating the repeated quantifiers as "$\forall \ldots$". Finally, we mark types that can be reconstructed from the other information present in a rule by denoting them in a shaded font.

8.1 The Dolev-Yao Intruder

The *Dolev-Yao abstraction* of a crypto-protocol appears to be drawn from positions taken in [52] and from a simplified model presented in [44]. It assumes that such elementary data as principal names, keys and nonces are atomic rather than strings of bits, as implemented in practice. Furthermore, it views the operations needed to assemble messages, i.e. concatenation, encryption and digital signature, as pure constructors in an initial algebra. Therefore, if n is a nonce and k a key, $\{n\}_k$ is a composite object whose structure is clearly recognizable. This means for example that a term of the form $\{t\}_k$ cannot be mistaken for a concatenation $(t_1\ t_2)$, and that $\{t\}_k = \{t'\}_{k'}$ if and only if $t = t'$ and $k = k'$. This also means that the Dolev-Yao model abstracts away the details of the cryptographic algorithms in use, reducing in this way encryption and decryption to atomic operations. Indeed, it is often said to adopt a *black box* view on cryptography.

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

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

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

MSR is a clear instance of the the Dolev-Yao abstraction. Elementary data are indeed atomic, message are constructed by applying symbolic operators, and the criterion for identifying terms is plain syntactic equality. We will now give a specification of the Dolev-Yao intruder in *MSR*. In Section 8.3, we give an encoding of an optimized version of this attacker proposed in [50] and formalized using a previous version of *MSR* in [27].

It has been proved that there is no advantage in considering more than one Dolev-Yao adversary in any given system [59]. Therefore, we select a principal, I say, and endow it with the powers of the Dolev-Yao intruder.

Since the intruder can learn and manipulate information, he must be able to store data out of sight from other principals. This is easily achieved in *MSR* by associating I with a memory predicate $\mathit{I}(\_)$ whose single argument can hold a message (the subscript "$_\mathsf{I}$" is kept implicit). The *standard Dolev-Yao intruder signature* Σ_{DY} for this model is defined as:

$$\Sigma_{DY} = \mathsf{I} : \mathsf{principal},\quad \mathit{I}(\_) : \mathsf{principal} \times \mathsf{msg}$$

On the basis of this declarations, we will express each of the intruder's capabilities as one or more roles consisting of a single rule. We give them a name (written in bold to its left) that will be referred to in Section 8.2. Each of these roles will be anchored at I and altogether model the actions that the Dolev-Yao intruder can undertake to mount an attack. They constitute the *standard Dolev-Yao intruder theory* that we denote $\mathcal{P}_{DY}$. Thus, the knowledge of the intruder is represented in a distributed fashion as a collection of memory predicates of the form $\mathit{I}(t)$ for all known terms t.

The first line of the description of the Dolev-Yao intruder is then expressed by the following two roles, anchored on I. The rule on the left 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. The rule on the right emits a memorized message out in the public network.

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

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

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

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

$$\mathbf{SDC}: \begin{pmatrix} \forall A, B : \mathsf{principal}. & & & \\ \forall k : \mathsf{shK}\ A\ B. & \begin{matrix} I(\{t\}_k) \\ I(k) \end{matrix} & \rightarrow & I(t) \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}} \qquad \mathbf{SEC}: \begin{pmatrix} \forall A, B : \mathsf{principal}. & & & \\ \forall k : \mathsf{shK}\ A\ B. & \begin{matrix} I(t) \\ I(k) \end{matrix} & \rightarrow & I(\{t\}_k) \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}}$$

Both for taking apart and constructing a shared-key encrypted message, the intruder must know the key. Observe that most typing information can be inferred.

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

$$\mathbf{PDC}: \begin{pmatrix} \forall A : \mathsf{principal}. & & & \\ \forall k : \mathsf{pubK}\ A. & I(\{\!\{t\}\!\}_k) & \rightarrow & I(t) \\ \forall k' : \mathsf{privK}\ k. & I(k') & & \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}} \qquad \mathbf{PEC}: \begin{pmatrix} \forall A : \mathsf{principal}. & & & \\ \forall k : \mathsf{pubK}\ A. & \begin{matrix} I(t) \\ I(k) \end{matrix} & \rightarrow & I(\{\!\{t\}\!\}_k) \\ \forall t : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{I}}$$

We now tackle the often overlooked fourth line of the Dolev-Yao intruder specification above: the ability to access public information. He should clearly be entitled to look up the name and public keys of principals, but any attempted access to more sensitive information such as private keys should be forbidden. Our data access specification 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 our data access specification policy. We have the following five anchored roles:

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

$$\mathbf{IS1}: \begin{pmatrix} \forall A : \mathsf{principal}. & & & \\ \forall k : \mathsf{shK}\ \mathsf{I}\ A. & \cdot & \rightarrow & I(k) \end{pmatrix}^{\mathsf{I}} \qquad \mathbf{IS2}: \begin{pmatrix} \forall A : \mathsf{principal}. & & & \\ \forall k : \mathsf{shK}\ A\ \mathsf{I}. & \cdot & \rightarrow & I(k) \end{pmatrix}^{\mathsf{I}}$$

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

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

$$\mathbf{GNC}: \big(\quad \cdot \quad \rightarrow \quad \exists n : \mathsf{nonce}. \quad I(n)\big)^{\mathsf{I}} \qquad \mathbf{GMS}: \big(\quad \cdot \quad \rightarrow \quad \exists m : \mathsf{msg}. \quad I(m)\big)^{\mathsf{I}}$$

Observe that the rationale behind these two rules, although reasonable, may conflict with idiosyncrasies of individual protocols. For example, the full version of the Needham-Schroeder protocol discussed in Section 9.1.2 may be more accurately validated in the presence of an intruder who can create public keys (but not the corresponding private keys).

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

$$\mathbf{DEL}: \big(\forall t : \mathsf{msg}. \quad I(t) \quad \rightarrow \quad \cdot\big)^{\mathsf{I}} \qquad \mathbf{DUP}: \begin{pmatrix} \forall t : \mathsf{msg}. & I(t) & \rightarrow & \begin{matrix} I(t) \\ I(t) \end{matrix} \end{pmatrix}^{\mathsf{I}}$$

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

1. This specification lies completely within *MSR* and can therefore be adapted, were the protocol at hand require it. This differs from many specification languages which either hardwire the intruder, or express it in a language different from the protocol under examination. It should be observed that our previous version of *MSR* belonged to this latter class: although the specification of the intruder was subject to the same execution semantics as the other roles, it relied on a slightly different language.

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

3. All but the fresh data generation rules can be automatically generated from the term language, and their typing and data access specification rules.

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

Property 8.1 (*Typing and Data Access Specification Validity of the Dolev-Yao Model*)

Let Σ_{DY} and $\mathcal{P}_{DY}$ be the signature and protocol theory for the Dolev-Yao intruder. Then, the judgments

$$\vdash \Sigma_{DY} \qquad\qquad \Sigma_{DY} \vdash \mathcal{P}_{DY} \qquad\qquad \Sigma_{DY} \Vdash \mathcal{P}_{DY}$$

are derivable.

Proof: We omit this easy inspection. □

The validation of the judgment "$\Sigma_{DY} \Vdash \mathcal{P}_{DY}$" makes use of all the data access specification rules in Section 5, except the ones dealing with role state predicates.

8.2 The Most Powerful 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 adversary that plays by the rules of the Dolev-Yao abstraction is built into the most successful security protocol verification systems [10, 43, 49, 51, 54, 57, 58]. 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 8.1. Our precise definition of data access specification 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 data access specification valid execution judgment $\mathcal{P} \triangleright [S]\,^{R}_{\Sigma} \longrightarrow^* [S']\,^{R'}_{\Sigma'}$. 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 beyond $\mathcal{P}_{DY}$; 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 $\mathsf{I}(\_)$; and 3) the judgment $\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY} \triangleright [\ulcorner S \urcorner]\,^{\ulcorner R \urcorner}_{\ulcorner \Sigma \urcorner} \longrightarrow^* [\ulcorner S' \urcorner]\,^{\ulcorner R' \urcorner}_{\ulcorner \Sigma' \urcorner}$ is derivable.

8.2.1 Encoding Protocol Theories, States, and Signatures

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 $(\rho)^{\mathsf{I}}$:

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

The Dolev-Yao intruder model does not refer to any role state or memory predicate beside $\mathsf{I}(\_)$. 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 $\mathsf{I}(\_)$ for each of its arguments, as specified by the following definition:

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

The encoding of a signature Σ is obtained by including any part of the Dolev-Yao intruder signature Σ_{DY} that may be missing. We highlight this fact by writing $\ulcorner\Sigma\urcorner$ as $\Sigma\oplus\Sigma_{DY}$, which is defined as $\Sigma\setminus(\Sigma\cap\Sigma_{DY})\cup\Sigma_{DY}$. 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 8.17.

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 simple-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 $\mathbf{IS1}$ and $\mathbf{IS2}$ 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 data access specification restrictions, we can circumvent this difficulty by basing the encoding of an active role set R on a derivation $\mathcal{A}$ of the data access specification 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 next three sections show how this is achieved. They show how to determine a specific set $\ulcorner\mathcal{A}\urcorner$ of Dolev-Yao intruder roles from any given derivation $\mathcal{A}$ of $\Sigma\ \Vdash\ R$. It does so by mapping each rule instance occurring in $\mathcal{A}$ to zero or more intruder roles from Section 8.1. We then 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 $\mathcal{P}_{DY}$.

8.2.2 Dolev-Yao Emulation of the Antecedent of a Rule

We begin by compiling the list of Dolev-Yao rules needed to emulate the actions performed in the right-hand side of a fully instantiated rule for a generic intruder. For each relevant data access specification judgment, we draw a table that associates each rule instance (in the left column) to the roles in Σ_{DY} that will emulate it in our encoding. We will be using the same device in the next two sections.

We begin with the judgment $\Gamma;\Delta\ \Vdash^{a}_{A}\ k\gg\Delta'$ that manages public keys. Its four rules yield the following table.

$\dfrac{}{(\Gamma, k : \mathsf{pubK}\ B, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k, k') \Vdash^{a}_{A} k \gg (\Delta, k, k')}\ \textbf{kac\_pss}$	**DUP**
$\dfrac{}{(\Gamma, k : \mathsf{pubK}\ B, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k') \Vdash^{a}_{A} k \gg (\Delta, k, k')}\ \textbf{kac\_pus}$	**DUP**
$\dfrac{}{(\Gamma, k : \mathsf{pubK}\ A, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); (\Delta, k) \Vdash^{a}_{A} k \gg (\Delta, k, k')}\ \textbf{kac\_psu}$	**IPB**, **IPV**
$\dfrac{}{(\Gamma, k : \mathsf{pubK}\ A, \Gamma', k' : \mathsf{privK}\ k, \Gamma''); \Delta \Vdash^{a}_{A} k \gg (\Delta, k')}\ \textbf{kac\_puu}$	**IPV**, **DUP**

Before we can prove that the encoding given by the Dolev-Yao rules on the right-hand column can emulate the left-hand side decryption actions, we need to provide an encoding $\ulcorner\Delta\urcorner$ of the intruder knowledge Δ. We do so by simply embedding every item in it into the memory predicate $\mathsf{I}(\_)$.

$$\begin{array}{rcl} \ulcorner\cdot\urcorner & = & \cdot \\ \ulcorner\Delta, t\urcorner & = & \ulcorner\Delta\urcorner,\ \mathsf{I}(t) \end{array}$$

The desired correctness lemma is then as follows.

Lemma 8.2 (*Dolev-Yao Emulation of Public-Key Decryptability*)

Let $\Sigma = (\Sigma_1, k : \mathsf{pubK}\ A, \Sigma_2, k' : \mathsf{privK}\ k, \Sigma_3)$ *be a signature with* A *a principal name,* k *and* k' *keys, and* Σ_1, Σ_2 *and* Σ_3 *signature fragments. Let moreover* Δ *and* Δ' *be knowledge contexts compatible with* Σ.

If $\mathcal{A} :: \Sigma; \Delta \Vdash^{a}_{\mathsf{I}} k \gg \Delta'$, *then* $\cdot \rhd [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^{*} [\ulcorner\Delta'\urcorner, \mathsf{I}(k')]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$.

Proof: The proof proceeds by induction on the structure of $\mathcal{A}$. There are two cases to examine:

$\boxed{\textbf{kac\_puu}}\quad \mathcal{A} = \dfrac{}{(\Sigma_1, k : \mathsf{pubK}\ B, \Sigma_2, k' : \mathsf{privK}\ k, \Sigma_3); \Delta \Vdash^{a}_{A} k \gg (\Delta, k')}\ \textbf{kac\_puu}$

with $\Delta' = (\Delta, k')$. Recall that $\ulcorner\mathcal{A}\urcorner = (\mathbf{IPV}, \mathbf{DUP})$.

$$\begin{array}{lll}
\mathcal{E}_0 :: \cdot \rhd [\ulcorner\Delta\urcorner]^{\mathbf{IPV},\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow [\ulcorner\Delta\urcorner]^{(\cdot\to\mathsf{I}(k'))^{!},\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rule } \textbf{sex\_all} \text{ on } \mathbf{IPV}, \\
\mathcal{E}_1 :: \cdot \rhd [\ulcorner\Delta\urcorner]^{(\cdot\to\mathsf{I}(k'))^{!},\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow [\ulcorner\Delta\urcorner, \mathsf{I}(k')]^{(\cdot)^{!},\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rules } \textbf{sex\_seq} \text{ and } \textbf{seq\_core}, \\
\mathcal{E}_2 :: \cdot \rhd [\ulcorner\Delta\urcorner, \mathsf{I}(k')]^{(\cdot)^{!},\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow [\ulcorner\Delta\urcorner, \mathsf{I}(k')]^{\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rule } \textbf{sex\_dot}. \\
\mathcal{E}_3 :: \cdot \rhd [\ulcorner\Delta\urcorner, \mathsf{I}(k')]^{\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow [\ulcorner\Delta\urcorner, \mathsf{I}(k')]^{(\mathsf{I}(k')\to\mathsf{I}(k'),\mathsf{I}(k'))^{!}}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rule } \textbf{sex\_all} \text{ on } \mathbf{DUP}, \\
\mathcal{E}_4 :: \cdot \rhd [\ulcorner\Delta\urcorner, \mathsf{I}(k')]^{(\mathsf{I}(k')\to\mathsf{I}(k'),\mathsf{I}(k'))^{!}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow [\ulcorner\Delta\urcorner, \mathsf{I}(k'), \mathsf{I}(k')]^{(\cdot)^{!}}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rules } \textbf{sex\_seq} \text{ and } \textbf{sex\_core}, \\
\mathcal{E}_5 :: \cdot \rhd [\ulcorner\Delta\urcorner, \mathsf{I}(k'), \mathsf{I}(k')]^{(\cdot)^{!}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow [\ulcorner\Delta\urcorner, \mathsf{I}(k'), \mathsf{I}(k')]^{\cdot}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rule } \textbf{sex\_dot}. \\
\mathcal{E} :: \cdot \rhd [\ulcorner\Delta\urcorner]^{\mathbf{IPV},\mathbf{DUP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^{*} [\ulcorner\Delta'\urcorner, \mathsf{I}(k')]^{\cdot}_{\Sigma\oplus\Sigma_{DY}} & & \text{by rules } \textbf{sex\_it0} \text{ and } \textbf{sex\_itn} \text{ on } \mathcal{E}_0\text{–}\mathcal{E}_5.
\end{array}$$

Rule **kac_pus** is processed in a similar but simpler way. □

The use of shared key encryption in the antecedent of a rule is handled similarly. The relevant rules and their encoding is as follows:

$\dfrac{}{\Gamma; (\Delta, k) \Vdash^{s}_{A} k \gg (\Delta, k)}\ \textbf{kac\_ss}$	**DUP**
$\dfrac{}{(\Gamma, k : \mathsf{shK}\ A\ B, \Gamma'); \Delta \Vdash^{s}_{A} k \gg (\Delta, k)}\ \textbf{kac\_su1}$	**IS1**
$\dfrac{}{(\Gamma, k : \mathsf{shK}\ B\ A, \Gamma'); \Delta \Vdash^{s}_{A} k \gg (\Delta, k)}\ \textbf{kac\_su2}$	**IS2**

They yield the following emulation lemma.

Lemma 8.3 (*Dolev-Yao Emulation of Shared-Key Decryptability*)

Let Σ *be a signature,*k *a key, and* Δ *and* Δ' *be knowledge contexts compatible with* Σ.
If $\mathcal{A} :: \Sigma; \Delta \Vdash^{s}_{\mathsf{I}} k \gg \Delta'$, *then* $\cdot \rhd [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner, I(k)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$.

Proof: The proof proceeds by induction on $\mathcal{A}$ similarly to the public-key variant of this property (Lemma 8.2). □

The encoding of the left-hand side term decomposition judgment $\Gamma; \Delta \Vdash_{A} \vec{t} \gg \Delta'$ is given by the following table.

$\dfrac{}{\Gamma; \Delta \Vdash_A \cdot \gg \Delta}\,\mathbf{tac\_dot}$	
$\dfrac{\Gamma; (\Delta, e) \Vdash_A \vec{t} \gg \Delta'}{\Gamma; (\Delta, e) \Vdash_A e, \vec{t} \gg \Delta'}\,\mathbf{tac\_kn}$ $\quad$ $\dfrac{\Gamma; (\Delta, t) \Vdash_A \vec{t} \gg \Delta'}{\Gamma; (\Delta, t) \Vdash_A t, \vec{t} \gg \Delta'}\,\mathbf{tac\_kn*}$	**DEL**
$\dfrac{(\Gamma, e : \tau, \Gamma'); (\Delta, e) \Vdash_A \vec{t} \gg \Delta'}{(\Gamma, e : \tau, \Gamma'); \Delta \Vdash_A e, \vec{t} \gg \Delta'}\,\mathbf{tac\_ukn}$	
$\dfrac{\Gamma; \Delta \Vdash_A t_1, t_2, \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A (t_1\, t_2), \vec{t} \gg \Delta'}\,\mathbf{tac\_cnc}$	**DCM**
$\dfrac{\Gamma; \Delta \Vdash^{s}_A k \gg \Delta' \quad \Gamma; \Delta' \Vdash_A t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \{t\}_k, \vec{t} \gg \Delta''}\,\mathbf{tac\_ske}$	**SDC**
$\dfrac{\Gamma; \Delta \Vdash^{p}_A k \gg \Delta' \quad \Gamma; \Delta' \Vdash_A t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \{\!\{t\}\!\}_k, \vec{t} \gg \Delta''}\,\mathbf{tac\_pke}$	**PDC**

The following lemma ascertains the correctness of the emulation.

Lemma 8.4 (*Dolev-Yao Emulation of Term Decomposability*)

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

Proof: The proof proceeds by induction on the structure of $\mathcal{A}$. We will expand the cases of rule **tac_kn** and **tac_ske**, which rely on techniques that have not yet been used.

$$\boxed{\mathbf{tac\_kn}} \qquad \mathcal{A} = \dfrac{\overset{\mathcal{A}_1}{\Sigma; (\Delta^*, e) \Vdash_A \vec{t}' \gg \Delta'}}{\Sigma; (\Delta^*, e) \Vdash_A e, \vec{t}' \gg \Delta'}\,\mathbf{tac\_kn}$$

with $\Delta = (\Delta^*, e)$ and $\vec{t} = (e, \vec{t}')$. Recall that $\ulcorner\mathcal{A}\urcorner = (\ulcorner\mathcal{A}_1\urcorner, \mathbf{DEL})$.

$\mathcal{E}_0 :: \cdot \rhd [\ulcorner\Delta^*\urcorner, I(e), \ulcorner\vec{t}'\urcorner]^{\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$ by induction hypothesis on $\mathcal{A}_1$,

$\mathcal{E}_1 :: \cdot \rhd [\ulcorner\Delta^*\urcorner, I(e), I(e), \ulcorner\vec{t}'\urcorner]^{\ulcorner\mathcal{A}_1\urcorner, DEL}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta^*\urcorner, I(e), \ulcorner\vec{t}'\urcorner]^{\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}}$ by using role $\mathbf{DEL}$, *i.e.* by rules **sex_all**, **sex_seq**, **sex_core** on $\mathbf{DEL}$, and then **sex_itn**,

$\mathcal{E} :: \cdot \rhd [\ulcorner\Delta^*\urcorner, I(e), I(e), \ulcorner\vec{t}'\urcorner]^{\ulcorner\mathcal{A}_1\urcorner, DEL}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$ by the Chaining Lemma 6.1 on $\mathcal{E}_0$–$\mathcal{E}_1$.

$$\boxed{\textbf{tac\_ske}} \qquad \mathcal{A} = \cfrac{\overset{\mathcal{A}_1}{\Sigma;\Delta \Vdash^{s}_{A} k \gg \Delta'} \qquad \overset{\mathcal{A}_2}{\Sigma;\Delta' \Vdash_{A} t,\vec{t}' \gg \Delta''}}{\Sigma;\Delta \Vdash_{A} \{t\}_k,\vec{t}' \gg \Delta''}\ \textbf{tac\_kn}$$

with $\vec{t} = (\{t\}_k, \vec{t}')$. Recall that $\ulcorner\mathcal{A}\urcorner = (\ulcorner\mathcal{A}_1\urcorner, \ulcorner\mathcal{A}_2\urcorner, \mathbf{SDC})$.

$\mathcal{E}_0' :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner, \mathsf{I}(k)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by the Shared-Key Lemma 8.3 on $\mathcal{A}_1$,
$\mathcal{E}_0 :: \cdot \triangleright [\ulcorner\Delta\urcorner, \ulcorner\vec{t}'\urcorner, \mathsf{I}(\{t\}_k)]^{\ulcorner\mathcal{A}_1\urcorner,\ulcorner\mathcal{A}_2\urcorner,SDC}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner, \ulcorner\vec{t}'\urcorner, \mathsf{I}(\{t\}_k), \mathsf{I}(k)]^{\ulcorner\mathcal{A}_2\urcorner,SDC}_{\Sigma\oplus\Sigma_{DY}}$	by the Weakening Lemma 6.4 on $\mathcal{E}_0'$,
$\mathcal{E}_1 :: \cdot\triangleright [\ulcorner\Delta'\urcorner, \ulcorner\vec{t}'\urcorner, \mathsf{I}(\{t\}_k), \mathsf{I}(k)]^{\ulcorner\mathcal{A}_2\urcorner,SDC}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner, \ulcorner\vec{t}'\urcorner, \mathsf{I}(t)]^{\ulcorner\mathcal{A}_2\urcorner}_{\Sigma\oplus\Sigma_{DY}}$	by using role $\mathbf{SDC}$,
$\mathcal{E}_3 :: \cdot \triangleright [\ulcorner\Delta'\urcorner, \ulcorner\vec{t}'\urcorner, \mathsf{I}(t)]^{\ulcorner\mathcal{A}_2\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta''\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by induction hypothesis on $\mathcal{A}_2$,
$\mathcal{E} :: \cdot \triangleright [\ulcorner\Delta\urcorner, \ulcorner\vec{t}'\urcorner, \mathsf{I}(\{t\}_k)]^{\ulcorner\mathcal{A}_1\urcorner,\ulcorner\mathcal{A}_2\urcorner,SDC}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta''\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by the Chaining Lemma 6.1 on $\mathcal{E}_0$–$\mathcal{E}_3$.

The other rules are handled similarly. □

The knowledge merging judgment $\Delta > \vec{t} > \Delta'$ is translated as per the following table.

$\cfrac{}{\Delta > \cdot > \Delta}\ \textbf{mac\_dot}$		
$\cfrac{\Delta > \vec{e} > \Delta'}{\Delta > e,\vec{e} > (\Delta', e)}\ \textbf{mac\_ukn}$	$\cfrac{\Delta > \vec{t} > \Delta'}{\Delta > t,\vec{t} > (\Delta', t)}\ \textbf{mac\_ukn}*$	
$\cfrac{\Delta > \vec{e} > \Delta'}{(\Delta, e) > e,\vec{e} > (\Delta', e)}\ \textbf{mac\_kn}$	$\cfrac{\Delta > \vec{t} > \Delta'}{(\Delta, t) > t,\vec{t} > (\Delta', t)}\ \textbf{mac\_kn}*$	**DEL**

The corresponding correctness lemma is as follows.

Lemma 8.5 (*Dolev-Yao Emulation of Knowledge Merging*)

Let Σ be a signature, $\vec{e}$ a term tuple, and Δ and Δ' be knowledge contexts compatible with Σ.
If $\mathcal{A} :: \Delta > \vec{e} > \Delta'$, *then* $\cdot \triangleright [\ulcorner\Delta\urcorner, \ulcorner\vec{e}\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta'\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$.

Proof: The proof proceeds again by induction on the structure of $\mathcal{A}$. Rules $\textbf{mac\_ukn}$ and $\textbf{mac\_kn}$ make use of the Execution Weakening Lemma 6.4. □

Finally, we have the following translation of the data access specification rules for the left-hand side of an MSR rule.

$\dfrac{\Delta > (A,\vec{e}) > \Delta' \quad \Gamma;\Delta' \Vdash_A \mathit{lhs} > \vec{t}' \gg \Delta''}{\Gamma;\Delta \Vdash_A L(A,\vec{e}), \mathit{lhs} > \vec{t}' \gg \Delta''}\,\textbf{lac\_rsp}$ $\dfrac{\Delta > (A,\vec{t}) > \Delta' \quad \Gamma;\Delta' \Vdash_A \mathit{lhs} > \vec{t}' \gg \Delta''}{\Gamma;\Delta \Vdash_A L(A,\vec{t}), \mathit{lhs} > \vec{t}' \gg \Delta''}\,\textbf{lac\_rsp}{*}$ $\quad$ $\dfrac{\Delta > (A,\vec{t}) > \Delta' \quad \Gamma;\Delta' \Vdash_A \mathit{lhs} > \vec{t}' \gg \Delta''}{\Gamma;\Delta \Vdash_A \mathsf{L}_l(A,\vec{t}), \mathit{lhs} > \vec{t}' \gg \Delta''}\,\textbf{lac\_rsp}{**}$	
$\dfrac{\Gamma;\Delta \Vdash_A \mathit{lhs} > (t,\vec{t}') \gg \Delta''}{\Gamma;\Delta \Vdash_A \mathsf{N}(t), \mathit{lhs} > \vec{t}' \gg \Delta''}\,\textbf{lac\_net}$	**INT**
$\dfrac{\Gamma;\Delta \Vdash_A \mathit{lhs} > (\vec{t},\vec{t}') \gg \Delta'}{\Gamma;\Delta \Vdash_A \mathsf{M}_A(\vec{t}), \mathit{lhs} > \vec{t}' \gg \Delta'}\,\textbf{lac\_mem}$	
$\dfrac{\Gamma;\Delta \Vdash_A \vec{t} \gg \Delta'}{\Gamma;\Delta \Vdash_A \cdot > \vec{t} \gg \Delta'}\,\textbf{lac\_dot}$	

In order to state and prove their correctness, we need to define the encoding $\ulcorner \mathit{lhs} \urcorner$ of a rule antecedent lhs. It leaves network predicates intact, but translates local and global memory predicates into the intruder predicate $\mathsf{I}(\_)$ (see earlier for the definition of $\ulcorner \vec{t} \urcorner$).

$$\begin{aligned}
\ulcorner . \urcorner &= \cdot \\
\ulcorner \mathit{lhs}, \mathsf{N}(t) \urcorner &= \ulcorner \mathit{lhs} \urcorner, \mathsf{N}(t) \\
\ulcorner \mathit{lhs}, \mathsf{M}_{\mathsf{I}}(\vec{t}) \urcorner &= \ulcorner \mathit{lhs} \urcorner,\ \ulcorner \vec{t} \urcorner \\
\ulcorner \mathit{lhs}, L(\vec{t}) \urcorner &= \ulcorner \mathit{lhs} \urcorner,\ \ulcorner \vec{t} \urcorner
\end{aligned}$$

We are now in a position to ascertain that the antecedent of an arbitrary intruder rule with a valid data access specification derivation behaves correctly.

Lemma 8.6 (*Dolev-Yao Emulation of Left-Hand Side Decomposability*)

Let Σ be a signature, lhs a predicate sequence, $\vec{t}$ a term tuple, and Δ and Δ' be knowledge contexts compatible with Σ. If $\mathcal{A} :: \Sigma;\Delta \Vdash_{\mathsf{I}} \mathit{lhs} > \vec{t} \gg \Delta'$, then $\cdot \triangleright [\ulcorner \Delta \urcorner, \ulcorner \mathit{lhs} \urcorner, \ulcorner \vec{t} \urcorner]^{\ulcorner \mathcal{A} \urcorner}_{\Sigma \oplus \Sigma_{DY}} \longrightarrow^* [\ulcorner \Delta' \urcorner]^{\cdot}_{\Sigma \oplus \Sigma_{DY}}$.

Proof: The proof proceeds by induction on the structure of $\mathcal{A}$. It uses the techniques already deployed for proving Lemma 8.4. Rules **lac_mem** and **lac_dot** simply rely on the induction hypothesis and the definition of $\ulcorner \mathit{lhs} \urcorner$ above. Rule **lac_net** additionally fires role **INT**. Finally, rule **lac_rsp** chains a use of Lemma 8.5 and an appeal to the induction hypothesis, as in the case of rule **kac_ske** in Lemma 8.4. □

8.2.3 Dolev-Yao Emulation of the Consequent of a Rule

We now turn to the emulation of the right-hand side of an arbitrary intruder rule on the basis of Dolev-Yao rules. We start from the judgment $\Gamma \looparrowright_A e$ that captures access to elementary information.

$\dfrac{}{(\Gamma, e : \textsf{principal}, \Gamma') \looparrowright_A \; e}\;\textbf{eac\_pr}$	**IPR**
$\dfrac{}{(\Gamma, e : \textsf{shK}\; A\; B, \Gamma') \looparrowright_A \; e}\;\textbf{eac\_s1}$	**IS1**
$\dfrac{}{(\Gamma, e : \textsf{shK}\; B\; A, \Gamma') \looparrowright_A \; e}\;\textbf{eac\_s2}$	**IS2**
$\dfrac{}{(\Gamma, e : \textsf{privK}\; k, \Gamma', k : \textsf{pubK}\; A, \Gamma'') \looparrowright_A \; e}\;\textbf{eac\_pp}$	**IPV**
$\dfrac{}{(\Gamma, e : \textsf{pubK}\; B, \Gamma') \looparrowright_A \; e}\;\textbf{eac\_p}$	**IPB**

The correctness of this encoding is given by the following lemma.

Lemma 8.7 (*Dolev-Yao Emulation of Elementary Information Access on the Right-Hand Side*)

Let Σ be a signature and e an elementary term. If $\mathcal{A} :: \Sigma \looparrowright_{\rceil} e$, *then* $\cdot \rhd [\cdot]^{\ulcorner A \urcorner}_{\Sigma \oplus \Sigma_{DY}} \longrightarrow^* [\ulcorner e \urcorner]^{\cdot}_{\Sigma \oplus \Sigma_{DY}}$.

Proof: This proof proceeds by cases on the structure of $\mathcal{A}$. It relies on the techniques already seen in the proof of Lemma 8.2, and on the implicit assumption that all the objects in $\mathcal{A}$ are well-typed. □

In most of the rest of this section, we will need to simulate the deletion and duplication of the entire knowledge context Δ. The operations $\mathbf{DEL}(\Delta)$ and $\mathbf{DUP}(\Delta)$ achieve this effect. They are defined as follows:

$$\begin{aligned} \mathbf{DEL}(\cdot) &= \cdot & \qquad \mathbf{DUP}(\cdot) &= \cdot \\ \mathbf{DEL}(\Delta, t) &= \mathbf{DEL}(\Delta),\ \mathbf{DEL} & \qquad \mathbf{DUP}(\Delta, t) &= \mathbf{DUP}(\Delta),\ \mathbf{DUP} \end{aligned}$$

The following lemmas ascertains that they actually work as expected. Given Δ, picking $\mathbf{DEL}(\Delta)$ as our active role set consisting of returns an empty state, while using $\mathbf{DUP}(\Delta)$ yields a state with two copies of Δ.

Lemma 8.8 (*Dolev-Yao Emulation of Knowledge Deletion and Duplication*)

Let Σ be a signature and Δ a knowledge context compatible with Σ.

1. $\cdot \rhd [\ulcorner \Delta \urcorner]^{\mathbf{DEL}(\Delta)}_{\Sigma \oplus \Sigma_{DY}} \longrightarrow^* [\cdot]^{\cdot}_{\Sigma \oplus \Sigma_{DY}}$;
2. $\cdot \rhd [\ulcorner \Delta \urcorner]^{\mathbf{DUP}(\Delta)}_{\Sigma \oplus \Sigma_{DY}} \longrightarrow^* [\ulcorner \Delta \urcorner, \ulcorner \Delta \urcorner]^{\cdot}_{\Sigma \oplus \Sigma_{DY}}$.

Proof: In both parts of this property, the proof proceeds by induction on the structure of the knowledge context Δ. □

The term constructability judgment $\Gamma; \Delta \looparrowright_A \; e$ is emulated according to the following table.

$\dfrac{}{\Gamma; (\Delta, e) \looparrowright_A \; e}\;\textbf{cac\_kn} \qquad \dfrac{}{\Gamma; (\Delta, t) \looparrowright_A \; t}\;\textbf{cac\_kn}{*}$	$\mathbf{DEL}(\Delta)$
$\dfrac{\Gamma \looparrowright_A \; e}{\Gamma; \Delta \looparrowright_A \; e}\;\textbf{cac\_ukn}$	$\mathbf{DEL}(\Delta)$
$\dfrac{\Gamma; \Delta \looparrowright_A \; t_1 \quad \Gamma; \Delta \looparrowright_A \; t_2}{\Gamma; \Delta \looparrowright_A \; t_1\, t_2}\;\textbf{cac\_cnc}$	$\mathbf{DUP}(\Delta)$, **CMP**
$\dfrac{\Gamma; \Delta \looparrowright_A \; t \quad \Gamma; \Delta \looparrowright_A \; k}{\Gamma; \Delta \looparrowright_A \; \{t\}_k}\;\textbf{cac\_ske}$	$\mathbf{DUP}(\Delta)$, **SEC**
$\dfrac{\Gamma; \Delta \looparrowright_A \; t \quad \Gamma; \Delta \looparrowright_A \; k}{\Gamma; \Delta \looparrowright_A \; \{\!\{t\}\!\}_k}\;\textbf{cac\_pke}$	$\mathbf{DUP}(\Delta)$, **PEC**

The following lemma shows that this translation is indeed correct.

Lemma 8.9 (*Dolev-Yao Emulation of Term Constructability*)

Let Σ *be a signature,* Δ *a knowledge context compatible with* Σ*, and* t *a term. If* $\mathcal{A} :: \Sigma;\Delta \looparrowright t$*, then* $\cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$.

Proof: This proof proceeds again by induction on the structure of $\mathcal{A}$. We will examine two of the most significant cases.

$\boxed{\textbf{cac\_ukn}}$ $$\mathcal{A} = \dfrac{\overset{\mathcal{A}_1}{\Sigma \looparrowright_A e}}{\Sigma;\Delta \looparrowright_A e}\,\textbf{cac\_ukn}$$

with $t = e$. Recall that $\ulcorner\mathcal{A}\urcorner = \ulcorner\mathcal{A}_1\urcorner, \mathbf{DEL}(\Delta)$.

$\mathcal{E}_0 :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\mathbf{DEL}(\Delta),\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\cdot]^{\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}}$	by Lemmas 8.8 on Δ and then the Weakening Lemma 6.4,
$\mathcal{E}_1 :: \cdot \triangleright [\cdot]^{\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(e)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by the Access Lemma 8.7 on $\mathcal{A}_1$,
$\mathcal{E} :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\mathbf{DEL}(\Delta),\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(e)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by the Chaining Lemma 6.1 on $\mathcal{E}_0$–$\mathcal{E}_1$.

$\boxed{\textbf{cac\_cnc}}$ $$\mathcal{A} = \dfrac{\overset{\mathcal{A}_1}{\Gamma;\Delta \looparrowright_A t_1} \quad \overset{\mathcal{A}_2}{\Gamma;\Delta \looparrowright_A t_2}}{\Gamma;\Delta \looparrowright_A t_1\, t_2}\,\textbf{cac\_cnc}$$

with $t = t_1\, t_2$. Recall that $\ulcorner\mathcal{A}\urcorner = \ulcorner\mathcal{A}_1\urcorner, \ulcorner\mathcal{A}_2\urcorner, \mathbf{DUP}(\Delta), \mathbf{CMP}$.

$\mathcal{E}_0' :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\mathbf{DUP}(\Delta)}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta\urcorner, \ulcorner\Delta\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by the Duplication Lemmas 8.8 on Δ
$\mathcal{E}_0 :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\mathbf{DUP}(\Delta),\ulcorner\mathcal{A}_1\urcorner,\ulcorner\mathcal{A}_2\urcorner,\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\Delta\urcorner, \ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_1\urcorner,\ulcorner\mathcal{A}_2\urcorner,\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}}$	by the Weakening Lemma 6.4 on $\mathcal{E}_0'$,
$\mathcal{E}_1' :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_1\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t_1)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by induction hypothesis on $\mathcal{A}_1$,
$\mathcal{E}_1 :: \cdot \triangleright [\ulcorner\Delta\urcorner, \ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_1\urcorner,\ulcorner\mathcal{A}_2\urcorner,\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t_1), \ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_2\urcorner,\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}}$	by the Weakening Lemma 6.4 on $\mathcal{E}_1'$,
$\mathcal{E}_2' :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_2\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t_2)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by induction hypothesis on $\mathcal{A}_2$,
$\mathcal{E}_2 :: \cdot \triangleright [I(t_1), \ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}_2\urcorner,\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t_1), I(t_2)]^{\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}}$	by the Weakening Lemma 6.4 on $\mathcal{E}_2'$,
$\mathcal{E}_3 :: \cdot \triangleright [I(t_1), I(t_2)]^{\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t_1\, t_2)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by using of role $\mathbf{CMP}$,
$\mathcal{E} :: \cdot \triangleright [\ulcorner\Delta\urcorner]^{\mathbf{DUP}(\Delta),\ulcorner\mathcal{A}_1\urcorner,\ulcorner\mathcal{A}_2\urcorner,\mathbf{CMP}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [I(t_1\, t_2)]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$	by the Chaining Lemma 6.1 on $\mathcal{E}_0$–$\mathcal{E}_3$.

The other possibilities are treated similarly □

This translation is trivially extended to the term tuple constructability judgment. It is formally defined next and followed by a proof of its correctness.

$\dfrac{}{\Gamma;\Delta \looparrowright_A \cdot}\,\textbf{cac\_dot}$	$\mathbf{DEL}(\Delta)$
$\dfrac{\Gamma;\Delta \looparrowright_A t \quad \Gamma;\Delta \looparrowright_A \vec{t}}{\Gamma;\Delta \looparrowright_A (t, \vec{t})}\,\textbf{cac\_ext}$	$\mathbf{DUP}(\Delta)$

Lemma 8.10 (*Dolev-Yao Emulation of Term Tuple Constructability*)

Let Σ be a signature, Δ a knowledge context compatible with Σ, and $\vec{t}$ a term tuple.
If $\mathcal{A} :: \Sigma; \Delta \looparrowright_{\urcorner} \vec{t}$, *then* $\cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner\vec{t}\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$.

Proof: Similarly to the previous result, the proof of this lemma proceeds by induction on the structure of $\mathcal{A}$. □

We now turn to emulating the constructability of predicate sequences using Dolev-Yao rules. The transformation is given by the following table.

$\dfrac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A lhs}{\Gamma; \Delta \looparrowright_A \mathsf{N}(t), lhs}\,\textbf{rac\_net}$	**DUP**(Δ), **TRN**
$\dfrac{\Gamma; \Delta \looparrowright_A \vec{t} \quad \Gamma; \Delta \looparrowright_A lhs}{\Gamma; \Delta \looparrowright_A \mathsf{M}_A(\vec{t}), lhs}\,\textbf{rac\_mem}$	**DUP**(Δ)
$\dfrac{\Gamma; \Delta \looparrowright_A (A, \vec{e}) \quad \Gamma; \Delta \looparrowright_A lhs}{\Gamma; \Delta \looparrowright_A L(A, \vec{e}), lhs}\,\textbf{rac\_rsp}$ $\dfrac{\Gamma; \Delta \looparrowright_A (A, \vec{t}) \quad \Gamma; \Delta \looparrowright_A lhs}{\Gamma; \Delta \looparrowright_A L(A, \vec{t}), lhs}\,\textbf{rac\_rsp}*$ $\dfrac{\Gamma; \Delta \looparrowright_A (A, \vec{t}) \quad \Gamma; \Delta \looparrowright_A lhs}{\Gamma; \Delta \looparrowright_A \mathsf{L}_l(A, \vec{t}), lhs}\,\textbf{rac\_rsp}{*}{*}$	**DUP**(Δ)
$\dfrac{}{\Gamma; \Delta \looparrowright_A \cdot}\,\textbf{rac\_dot}$	**DEL**(Δ)

The correctness of this translation is given by the following lemma.

Lemma 8.11 (*Dolev-Yao Emulation of Predicate Sequence Constructability*)

Let Σ be a signature, Δ a knowledge context compatible with Σ, and lhs a predicate sequence.
If $\mathcal{A} :: \Sigma; \Delta \looparrowright_{\urcorner} lhs$, *then* $\cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner lhs\urcorner]^{\cdot}_{\Sigma\oplus\Sigma_{DY}}$.

Proof: Once more, this proof proceeds by induction on the structure of $\mathcal{A}$. The case in which $\mathcal{A}$ ends in rule **rac_dot** is handled similarly to rule **cac_ukn** in Lemma 8.9. Rules **rac_mem** and **rac_rsp** are treated similarly to rule **cac_cnc** in the proof of same result. Finally, rule **rac_net** follows again this pattern, but it should be prefixed by an application of the role **TRN** that appears in the translation. □

Finally, we give the translation of a data access specification derivation for an entire rule consequent.

$\dfrac{(\Gamma, x : \mathsf{nonce}); (\Delta, x) \Vdash_A rhs}{\Gamma; \Delta \Vdash_A \exists x : \mathsf{nonce}.\, rhs}\,\textbf{rac\_nnc}$	**GNC**
$\dfrac{(\Gamma, x : \mathsf{msg}); (\Delta, x) \Vdash_A rhs}{\Gamma; \Delta \Vdash_A \exists x : \mathsf{msg}.\, rhs}\,\textbf{rac\_msg}$	**GMS**
$\dfrac{\Gamma; \Delta \looparrowright_A lhs}{\Gamma; \Delta \Vdash_A lhs}\,\textbf{rac\_ps}$	

The correctness of this translation follows.

Lemma 8.12 (*Dolev-Yao Emulation of Right-Hand Side Constructability*)

Let Σ and Σ' be signatures, Δ a knowledge context ,compatible with Σ rhs a rule's right-hand side, and lhs a predicate sequence such that the judgment "$(rhs)_\Sigma \gg (lhs)_{\Sigma'}$" is derivable.
If $\mathcal{A} :: \Sigma; \Delta \Vdash_{\urcorner} rhs$, *then* $\cdot \triangleright [\ulcorner\Delta\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner lhs\urcorner]^{\cdot}_{\Sigma'\oplus\Sigma_{DY}}$.

Proof: The proof proceeds by induction on the structure of $\mathcal{A}$. Let $\mathcal{E}_{rhs}$ be the given derivation of "$(rhs)_\Sigma \gg (lhs)_{\Sigma'}$". We will examine only the case in which it ends in rule $\mathbf{rac\_nnc}$.

$$\boxed{\mathbf{rac\_nnc}} \qquad \mathcal{A} = \cfrac{\overset{\mathcal{A}_1}{(\Gamma, x : \mathsf{nonce}); (\Delta, x) \ \Vdash_A \ rhs'}}{\Gamma; \Delta \ \Vdash_A \ \exists x : \mathsf{nonce}.\, rhs'} \,\mathbf{rac\_nnc}$$

with $rhs = \exists x : \mathsf{nonce}.\, rhs'$. Recall that $\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}_1 \urcorner, \mathbf{GNC}$.

$$\begin{array}{lll}
\mathcal{E}'_{rhs} :: ([\mathsf{n}/x]rhs')_{\Sigma,\mathsf{n}:\mathsf{nonce}} \gg (lhs)_{\Sigma'} & & \text{by inversion on } \mathcal{E}_{rhs}, \\
\mathcal{E}_0 \ :: \ \cdot \triangleright [\ulcorner \Delta \urcorner]^{\mathbf{GNC}, \ulcorner \mathcal{A}_1 \urcorner}_{\Sigma \oplus \Sigma_{DY}} \ \longrightarrow^* \ [\ulcorner \Delta \urcorner, \mathsf{I}(\mathsf{n})]^{\ulcorner \mathcal{A}_1 \urcorner}_{\Sigma \oplus \Sigma_{DY}, \mathsf{n}:\mathsf{nonce}} & & \text{by using role } \mathbf{GNC}, \\
\mathcal{E}_1 \ :: \ \cdot \triangleright [\ulcorner \Delta \urcorner, \mathsf{I}(\mathsf{n})]^{\ulcorner \mathcal{A}_1 \urcorner}_{(\Sigma,\mathsf{n}:\mathsf{nonce}) \oplus \Sigma_{DY}} \ \longrightarrow^* \ [\ulcorner rhs' \urcorner]^{\cdot}_{\Sigma' \oplus \Sigma_{DY}} & & \text{by induction hypothesis on } \mathcal{E}'_{rhs} \text{ and } \mathcal{A}_1, \\
\mathcal{E} \ :: \ \cdot \triangleright [\ulcorner \Delta \urcorner]^{\mathbf{GNC}, \ulcorner \mathcal{A}_1 \urcorner}_{\Sigma \oplus \Sigma_{DY}} \ \longrightarrow^* \ [\ulcorner rhs' \urcorner]^{\cdot}_{\Sigma' \oplus \Sigma_{DY}} & & \text{by the Chaining Lemma 6.1 on } \mathcal{E}_0\text{–}\mathcal{E}_1.
\end{array}$$

Observe that firing role $\mathbf{GNC}$ involves selecting a constant of type nonce that does not appear in $\Sigma \oplus \Sigma_{DY}$. We choose the constant n that appears in $\mathcal{E}'_{rhs}$. Rule $\mathbf{rac\_msg}$ is treated identically to rule $\mathbf{rac\_nnc}$, while rule $\mathbf{rac\_ps}$ relies on the Constructability Lemma 8.11 for predicate sequences. □

8.2.4 Dolev-Yao Emulation of Rule and Roles

We will now glue the translations of data access specification derivations of the antecedent and of the consequent of a rule together and use them to translate roles. As we will see, most judgments will not produce any additional Dolev-Yao rules.

We begin with the judgment $\Gamma \ \Vdash_A \ r$ which handles MSR rules. It is translated according to the following (vacuous) table (the real work is being done by the translation of the premises).

$\cfrac{\Gamma; \cdot \ \Vdash_A \ lhs > \cdot \gg \Delta \quad \Gamma; \Delta \ \Vdash_A \ rhs}{\Gamma \ \Vdash_A \ lhs \to rhs}\,\mathbf{uac\_core}$	
$\cfrac{(\Gamma, x : \tau) \ \Vdash_A \ r}{\Gamma \ \Vdash_A \ \forall x : \tau.\, r}\,\mathbf{uac\_all}$	

The correctness of this strategy will be established later, when we look at entire protocols. Instead, we show that the transformation of all the judgments we have analyzed so far is invariant with respect to term substitution.

Lemma 8.13 (*Emulation Invariance under Substitution*)

Let Σ and Σ' be signatures, t a term, τ a type, x a variable, and Γ a typing context fragment such that the judgments $\Sigma, \Sigma' \ \vdash \ t : \tau$ and $\models^{c} \ (\Sigma, x : \tau, \Gamma)$ hold. Furthermore,

- *let Δ be a knowledge context and $\Gamma'; \Delta \ \Vdash_{\ulcorner} \ X > Y \gg Z$ an data access specification judgment among*

$$\begin{array}{ccccc}
\Gamma'; \Delta \ \Vdash^{s}_{\ulcorner} \ k \gg \Delta' & \Gamma'; \Delta \ \Vdash^{a}_{\ulcorner} \ k \gg \Delta' & \Gamma'; \Delta \ \Vdash_{\ulcorner} \ \vec{t} \gg \Delta' & \Gamma'; \Delta \ \Vdash_{\ulcorner} \ lhs > \vec{t} \gg \Delta' & \\
\Gamma' \looparrowright_{\ulcorner} \ e & \Gamma; \Delta \looparrowright_{\ulcorner} \ t' & \Gamma; \Delta \looparrowright_{\ulcorner} \ \vec{t} & \Gamma'; \Delta \looparrowright_{\ulcorner} \ lhs & \Gamma'; \Delta \ \Vdash_{\ulcorner} \ rhs \\
& \Gamma' \ \Vdash_{\ulcorner} \ r & & \Gamma' \ \Vdash_{\ulcorner} \ \rho &
\end{array}$$

  (Y and Z are defined only for judgments with more than two (resp. three) objects to the right of the turnstile symbol.)

  For every derivation $\mathcal{A}$ of $(\Sigma, x : \tau, \Gamma); \Delta \Vdash_{\ulcorner} \ X > Y \gg Z$, there exists a derivation $\mathcal{A}_{[t/x]}$ of $(\Sigma, \Sigma', [t/x]\Gamma); [t/x]\Delta \Vdash_{\ulcorner}$ $[t/x]X > [t/x]Y \gg [t/x]Z$ such that

$$\ulcorner \mathcal{A} \urcorner \ = \ \ulcorner \mathcal{A}_{[t/x]} \urcorner$$

- *let Δ and Δ' be knowledge contexts and $\vec{t}$ a tuple consisting of either ground terms or variables.*

  For every derivation $\mathcal{A}$ of $\Delta > \vec{t} > \Delta'$, there exists a derivation $\mathcal{A}_{[t/x]}$ of $[t/x]\Delta > [t/x]\vec{t} > [t/x]\Delta'$ such that

$$\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}_{[t/x]} \urcorner$$

Proof: By the Term Substitution Lemma 6.17, we know that there is a derivation $\mathcal{A}_{[t/x]}$ of the postulated judgments. A close inspection of the proof of this property reveals that the specific derivation $\mathcal{A}_{[t/x]}$ it constructs has the same structure as $\mathcal{A}$, *i.e.* consists of the application of the exact same rules in the same order, although to different instances of their meta-variables.

We shall now observe that $\ulcorner \mathcal{A} \urcorner$ is constructed on the basis of the structure of $\mathcal{A}$ and independently from the actual instances of the meta-variables it accesses (except for the presence of correlations that are preserved in $\mathcal{A}_{[t/x]}$). Therefore, since $\mathcal{A}$ and $\mathcal{A}_{[t/x]}$ share the same structure, we deduce that $\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}_{[t/x]} \urcorner$.

A formal proof of this result proceeds by induction on the proof of the Term Substitution Lemma 6.17. □

The emulation of a role simply accumulates the Dolev-Yao rules that result from the emulation of its constituents.

$\dfrac{}{\Gamma \Vdash_A \cdot}\ \textbf{oac\_dot}$	
$\dfrac{(\Gamma, L : \vec{\tau}) \Vdash_A \rho}{\Gamma \Vdash_A \exists L : \vec{\tau}.\, \rho}\ \textbf{oac\_rsp}$	
$\dfrac{\Gamma \Vdash_A r \quad \Gamma \Vdash_A \rho}{\Gamma \Vdash_A r, \rho}\ \textbf{oac\_rule}$	

Again, the correctness of this translation will be established at the protocol level. We have a result similar to Lemma 8.13 concerning the instantiation of role stat predicate symbols.

Lemma 8.14 (*Emulation Invariance under Role State Predicate Symbol Substitution*)

Let Σ and Σ' be signatures, L_l and L a role state predicate constant and variable respectively, $\vec{\tau}$ a tuple type, Γ a typing context fragment, Δ a knowledge context compatible with Σ, and $\Gamma'; \Delta \Vdash_A X > Y \gg Z$ an data access specification judgment among

$$\Gamma'; \Delta \Vdash_A lhs > \vec{t} \gg \Delta' \qquad \Gamma'; \Delta \looparrowright_A lhs \qquad \Gamma'; \Delta \Vdash_A rhs \qquad \Gamma' \Vdash_A r \qquad \Gamma' \Vdash_A \rho$$

(Y and Z are defined only in the first of these judgments; Δ does not appear in the last two.) Assume that $\vdash (\Sigma, \mathsf{L}_l : \vec{\tau}, \Sigma')$ and $\vdash (\Sigma, L : \vec{\tau}, \Gamma)$.

Then for every derivation $\mathcal{A}$ of $(\Sigma, L : \vec{\tau}, \Gamma); \Delta \Vdash_A X > Y \gg Z$, there exists a derivation $\mathcal{A}_{[\mathsf{L}_l/L]}$ of $(\Sigma, \Sigma', \Gamma); \Delta \Vdash_A [\mathsf{L}_l/L]X > Y \gg Z$ such that

$$\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}_{[\mathsf{L}_l/L]} \urcorner$$

Proof: This result relies on the same proof technique already sketched for the Term Substitution Lemma 8.13. □

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 $\ulcorner \mathcal{A} \urcorner$.

$\dfrac{}{\Sigma \Vdash \cdot}\ \textbf{aac\_dot}$	
$\dfrac{\Sigma \Vdash R \quad \Sigma \Vdash_{\mathsf{A}} \rho}{\Sigma \Vdash R, \rho^{\mathsf{A}}}\ \textbf{aac\_ext}$	ρ^{A} if $A \neq \mathsf{I}$

It is instructive to unfold this definition to the level of active roles:

$$\left[\begin{array}{lcl} \ulcorner \cdot \urcorner & = & \cdot \\ \ulcorner R, (\rho)^{\mathsf{A}} \urcorner & = & \begin{cases} \ulcorner R \urcorner\ \ulcorner \mathcal{A} \urcorner & \text{if } \mathsf{A} \neq \mathsf{I}, \text{ where } \mathcal{A} \text{ is a derivation of } \Sigma \Vdash_{\mathsf{I}} \rho \\ \ulcorner R \urcorner\ (\rho)^{\mathsf{A}} & \text{otherwise} \end{cases} \end{array}\right.$$

Here, $\mathcal{A}$ consists exclusively of Dolev-Yao roles from $\mathcal{P}_{DY}$.

Given a protocol theory $\mathcal{P}$ and a derivation $\mathcal{A}$ of $\Sigma \Vdash \mathcal{P}$, we translate $\ulcorner \mathcal{A} \urcorner$ by collecting the translation of every role occurring in $\mathcal{P}$.

$\dfrac{}{\Sigma \Vdash \cdot}\ \textbf{hac\_dot}$	
$\dfrac{\Sigma \Vdash \mathcal{P} \quad (\Sigma, A : \mathsf{principal}) \Vdash_A \rho}{\Sigma \Vdash \mathcal{P}, \rho^{\forall A}}\ \textbf{hac\_grole}$	$\rho^{\forall A}$
$\dfrac{\Sigma \Vdash \mathcal{P} \quad \Sigma \Vdash_{\mathsf{A}} \rho}{\Sigma \Vdash \mathcal{P}, \rho^{\mathsf{A}}}\ \textbf{hac\_arole}$	ρ^{A} if $A \neq \mathsf{I}$

The following lemma ensures that the emulation introduced in the last three sections transforms well-typed objects into well-typed entities.

Lemma 8.15 (*Typability of the Dolev-Yao Emulation*)

Let Σ be a signature, $\mathcal{P}$ a protocol theory, S a state and R an active role set. Let moreover Σ_{DY} and $\mathcal{P}_{DY}$ be the signature and protocol theory for the Dolev-Yao intruder.

1. *If $\vdash \Sigma$, then $\vdash \Sigma, \Sigma_{DY}$;*
2. *If $\Sigma \vdash \mathcal{P}$, then $\Sigma, \Sigma_{DY} \vdash \ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}$;*
3. *If $\Sigma \vdash S$, then $\Sigma, \Sigma_{DY} \vdash \ulcorner S \urcorner$;*
4. *If $\Sigma \vdash R$ and $\mathcal{A} :: \Sigma \Vdash R$, then $\Sigma, \Sigma_{DY} \vdash \ulcorner \mathcal{A} \urcorner$.*

Proof: By the Dolev-Yao Encoding Validity Lemma 8.1, we know that $\vdash \Sigma_{DY}$ and $\Sigma_{DY} \vdash \mathcal{P}_{DY}$ have derivations. By the Weakening Lemma 6.6, $\vdash \Sigma, \Sigma_{DY}$ is therefore derivable. Parts (2) and (3) rely on the same lemma and on a simple induction on the structure of a derivation for their premise. Point (4) is proved similarly; it uses the fact that $\ulcorner \mathcal{A} \urcorner$ contains elements from R (all the active roles that are not owned by the intruder) and possibly multiple occurrences of roles from $\mathcal{P}_{DY}$. □

We will need the following technical lemma to delete leftover role state predicate symbols.

Lemma 8.16 (*Deletion of Used Role State Predicate Symbols*)

Let Σ, Σ_{L} and Σ' be signature fragments such that Σ_{L} consists of role state predicate constants declarations only. Let moreover $\mathcal{P}$ be a protocol theory, S and S' two states, and R, R' two active role sets such that

$$\vdash \Sigma \qquad\qquad \Sigma \vdash S \qquad\qquad \Sigma \vdash R \qquad\qquad \mathcal{P} \rhd [S]^{R}_{\Sigma,\Sigma_{\mathsf{L}}} \longrightarrow^{*} [S']^{R'}_{\Sigma,\Sigma_{\mathsf{L}},\Sigma'}$$

Then $\mathcal{P} \rhd [S]^{R}_{\Sigma} \longrightarrow^{*} [S']^{R'}_{\Sigma,\Sigma'}$.

Proof: The proof proceeds by induction on a derivation $\mathcal{E}$ of "$\mathcal{P} \rhd [S]^{R}_{\Sigma,\Sigma_{\mathsf{L}}} \longrightarrow^{*} [S']^{R'}_{\Sigma,\Sigma_{\mathsf{L}},\Sigma'}$". Intuitively, the typing derivations affirm that none of the role state predicate symbols in Σ_{L} appear in S or R (they clearly cannot appear in $\mathcal{P}$). Consequently, no transition in $\mathcal{E}$ can make use of them. Therefore, they can be dropped. □

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 **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 **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 **ex_all** to instantiate a universal variable in an active intruder rule do not correspond to any action: we have proved that data access specification is preserved under substitution [17] and that this process does not affect the encoding of an data access specification derivation. Finally, the application of a fully instantiated intruder rule (**ex_core**) relies on results such as the above that specify the behavior of its constituents.

Theorem 8.17 (*Dolev-Yao Emulation of 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 $\quad \mathcal{P} \rhd [S]_{\Sigma}^{R} \longrightarrow^{(*)} [S']_{\Sigma,\Sigma'}^{R'}, \quad$ *then* $\quad (\ulcorner\mathcal{P}\urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}\urcorner} \longrightarrow^{*} [\ulcorner S'\urcorner]_{\Sigma\oplus\Sigma_{DY},\Sigma^{*}}^{\ulcorner\mathcal{A}'\urcorner}$

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: The proof of this theorem proceeds by induction on the structure of a derivation $\mathcal{E}$ of the given execution judgment "$\mathcal{P} \rhd [S]_{\Sigma}^{R} \longrightarrow^{*} [S']_{\Sigma,\Sigma'}^{R'}$". We distinguish cases on the basis of the last rule appearing in $\mathcal{E}$. With the exception of **sex_it0** and **sex_itn**, we must consider two subcases for every rule: whether the action applies to an object with owner I, or not.

$\boxed{\textbf{sex\_arole},\ \mathsf{A} \neq \mathsf{I}} \qquad \mathcal{E} = \dfrac{}{(\mathcal{P}^{*}, \rho^{\mathsf{A}}) \rhd [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,\rho^{\mathsf{A}}}}\ \textbf{sex\_arole}$

with $\mathcal{P} = (\mathcal{P}^{*}, \rho^{\mathsf{A}}),\ R' = R, \rho^{\mathsf{A}},\ S' = S$ and $\Sigma' = \cdot$.

$\ulcorner\mathcal{P}\urcorner = \ulcorner\mathcal{P}^{*}\urcorner, \rho^{\mathsf{A}}$ by definition of the Dolev-Yao translation,

$\mathcal{A}_{\rho} :: \Sigma \Vdash_{\mathsf{A}} \rho$ by inversion on rule **hac_arole** for $\mathcal{A}_{\mathcal{P}}$,

$\mathcal{A}' :: \Sigma \Vdash R, \rho^{\mathsf{A}}$ by rule **aac_ext** on $\mathcal{A}_{\rho}$ and $\mathcal{A}$,

$\ulcorner\mathcal{A}'\urcorner = \ulcorner\mathcal{A}\urcorner, \rho^{\mathsf{A}}$ by definition of the Dolev-Yao translation,

$\mathcal{E}' :: (\ulcorner\mathcal{P}^{*}\urcorner, \rho^{\mathsf{A}}, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}\urcorner} \longrightarrow^{*} [\ulcorner S'\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}\urcorner,\rho^{\mathsf{A}}}$ by rule **sex_arole**.

$\boxed{\textbf{sex\_arole},\ \mathsf{A} = \mathsf{I}} \qquad$ Here $\mathcal{P} = (\mathcal{P}^{*}, \rho^{\mathsf{I}})$ and $R' = R, \rho^{\mathsf{I}}$.

$\mathcal{A}_{\rho} :: \Sigma \Vdash_{\mathsf{I}} \rho$ by inversion on rule **hac_arole** for $\mathcal{A}_{\mathcal{P}}$,

By definition of the Dolev-Yao translation, $\ulcorner\mathcal{A}_{\rho}\urcorner$ is composed of a finite number of copies of intruder roles from $\mathcal{P}_{DY}$. With this theory at our disposal, we can individually copy them to the current active role set by means of rule **sex_arole**. The formal proof proceeds by induction on the structure of $\ulcorner\mathcal{A}_{\rho}\urcorner$. We omit it for the sake of conciseness.

Therefore,

$\mathcal{E}' :: (\ulcorner\mathcal{P}^{*}, \rho^{\mathsf{I}}\urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}\urcorner} \longrightarrow^{*} [\ulcorner S\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}\urcorner,\ulcorner\mathcal{A}_{\rho}\urcorner}$ by several applications of rule **sex_arole**.

Observe that $\ulcorner\mathcal{P}^{*}, \rho^{\mathsf{I}}\urcorner = \ulcorner\mathcal{P}^{*}\urcorner$.

$\boxed{\textbf{sex\_grole},\ \mathsf{A} \neq \mathsf{I}} \qquad \mathcal{E} = \dfrac{\begin{array}{c}\mathcal{T} \\ \Sigma \vdash \mathsf{A} : \mathsf{principal}\end{array}}{(\mathcal{P}^{*}, \rho^{\forall A}) \rhd [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,([\mathsf{A}/A]\rho)^{\mathsf{A}}}}\ \textbf{sex\_arole}$

with $\mathcal{P} = (\mathcal{P}^{*}, \rho^{\forall A}),\ R' = R, ([\mathsf{A}/A]\rho)^{\mathsf{A}},\ S' = S$ and $\Sigma' = \cdot$.

$\mathcal{A}_\rho :: \Sigma, A : \mathsf{principal} \Vdash_A \rho$ by inversion on rule **hac_grole** for $\mathcal{A}_\mathcal{P}$,

$\mathcal{A}'_\rho :: \Sigma \Vdash_{\mathsf{A}} [\mathsf{A}/A]\rho$ by the Substitution Lemma 6.17 on $\mathcal{A}_\rho$ and $\mathcal{T}$,

$\mathcal{A}' :: \Sigma \Vdash R, ([\mathsf{A}/A]\rho)^{\mathsf{A}}$ by rule **aac_ext** on $\mathcal{A}'_\rho$ and $\mathcal{A}$,

$\ulcorner\mathcal{A}'\urcorner = \ulcorner\mathcal{A}\urcorner, ([\mathsf{A}/A]\rho)^{\mathsf{A}}$ by definition of the Dolev-Yao translation,

$\mathcal{E}' :: (\ulcorner\mathcal{P}^*\urcorner, \rho^{\forall A}, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner S'\urcorner]^{\ulcorner\mathcal{A}\urcorner,([\mathsf{A}/A]\rho)^{\mathsf{A}}}_{\Sigma\oplus\Sigma_{DY}}$ by rule **sex_grole**.

$\boxed{\textbf{sex\_grole},\ \mathsf{A}=\mathsf{I}}$ Here $\mathcal{P} = (\mathcal{P}^*, \rho^{\forall A})$ and $R' = R, ([\mathsf{I}/A]\rho)^{\mathsf{I}}$.

$\mathcal{A}_A :: \Sigma, A : \mathsf{principal} \Vdash_A \rho$ by inversion on rule **hac_grole** for $\mathcal{A}_\mathcal{P}$,

$\mathcal{A}_\rho :: \Sigma \Vdash_{\mathsf{I}} [\mathsf{I}/A]\rho$ by the Substitution Lemma 6.17 on $\mathcal{A}'_\rho$ and $\mathcal{T}$,

As in the case of rule **sex_arole** (with $\mathsf{A} = \mathsf{I}$), the translation $\ulcorner\mathcal{A}_\rho\urcorner$ consists of a finite number of copies of intruder roles. We copy them from $\mathcal{P}_{DY}$ to the active role set by means of rule **sex_arole**. Therefore,

$\mathcal{E}' :: (\ulcorner\mathcal{P}^*, \rho^{\forall A}\urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}\urcorner,\ulcorner\mathcal{A}_\rho\urcorner}_{\Sigma\oplus\Sigma_{DY}}$ by several applications of rule **sex_arole**.

$\boxed{\textbf{sex\_rsp},\ \mathsf{A}\neq\mathsf{I}}$

$$\mathcal{E} = \dfrac{}{\mathcal{P} \rhd [S]^{R^*,(\exists L:\vec{\tau}.\,\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S]^{R^*,([\mathsf{L}_l/L]\rho)^{\mathsf{A}}}_{(\Sigma,\mathsf{L}_l:\vec{\tau})}}\ \textbf{sex\_rsp}$$

with $R = R^*, (\exists L : \vec{\tau}.\,\rho)^{\mathsf{A}}$, $R' = R^*, ([\mathsf{L}_l/L]\rho)^{\mathsf{A}}$, $S' = S$ and $\Sigma' = \mathsf{L}_l : \vec{\tau}$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R^*$ by inversion on rule **aac_ext** for $\mathcal{A}$ or $\mathcal{A}'$,

$\ulcorner\mathcal{A}\urcorner = \ulcorner\mathcal{A}_{R^*}\urcorner, (\exists L : \vec{\tau}.\,\rho)^{\mathsf{A}}$ by definition of the Dolev-Yao translation,

$\ulcorner\mathcal{A}'\urcorner = \ulcorner\mathcal{A}_{R^*}\urcorner, ([\mathsf{L}_l/L]\rho)^{\mathsf{A}}$ by definition of the Dolev-Yao translation,

$\mathcal{E}' :: (\ulcorner\mathcal{P}\urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}_{R^*}\urcorner,(\exists L:\vec{\tau}.\,\rho)^{\mathsf{A}}}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}_{R^*}\urcorner,([\mathsf{L}_l/L]\rho)^{\mathsf{A}}}_{\Sigma\oplus\Sigma_{DY},\mathsf{L}_l:\vec{\tau}}$ by rule **sex_rsp**.

$\boxed{\textbf{sex\_rsp},\ \mathsf{A}=\mathsf{I}}$ Here $R = R^*, (\exists L : \vec{\tau}.\,\rho)^{\mathsf{I}}$ and $R' = R^*, ([\mathsf{L}_l/L]\rho)^{\mathsf{I}}$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R^*$ and

$\mathcal{A}_L :: \Sigma \Vdash_{\mathsf{I}} \exists L : \vec{\tau}.\,\rho$ by inversion on rule **aac_ext** for $\mathcal{A}$,

$\ulcorner\mathcal{A}\urcorner = (\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_L\urcorner)$ by definition of the Dolev-Yao translation,

$\mathcal{A}_\rho :: \Sigma, L : \vec{\tau} \Vdash_{\mathsf{I}} \rho$ by inversion on rule **oac_rsp** for $\mathcal{A}_L$,

$\mathcal{A}'_\rho :: \Sigma, \mathsf{L}_l : \vec{\tau} \Vdash_{\mathsf{I}} [\mathsf{L}_l/L]\rho$ by the Substitution Lemma 6.18 on $\mathcal{T}_\Sigma$ and $\mathcal{A}_\rho$,

$\ulcorner\mathcal{A}'_\rho\urcorner = \ulcorner\mathcal{A}_\rho\urcorner = \ulcorner\mathcal{A}_L\urcorner$ by the Invariance Lemma 8.14 on $\mathcal{T}_\Sigma$ and $\mathcal{A}_L$,

$\ulcorner\mathcal{A}'\urcorner = (\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}'_\rho\urcorner)$ by definition of the Dolev-Yao translation,

$\mathcal{E} :: \mathcal{P} \rhd [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}_{R^*}\urcorner,\ulcorner\mathcal{A}_L\urcorner}_{\Sigma\oplus\Sigma_{DY}} \longrightarrow^* [\ulcorner S\urcorner]^{\ulcorner\mathcal{A}_{R^*}\urcorner,\ulcorner\mathcal{A}'_\rho\urcorner}_{\Sigma\oplus\Sigma_{DY}}$ by rule **sex_it0**.

$\boxed{\textbf{sex\_all},\ \mathsf{A}\neq\mathsf{I}}$

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{T}\\ \Sigma \vdash t : \tau\end{array}}{\mathcal{P} \rhd [S]^{R^*,((\forall x:\tau.\,r),\rho)^{\mathsf{A}}}_{\Sigma} \longrightarrow [S]^{R^*,(([t/x]r),\rho)^{\mathsf{A}}}_{\Sigma}}\ \textbf{sex\_all}$$

with $R = R^*, ((\forall x : \tau.\,r), \rho)^{\mathsf{A}}$, $R' = R^*, (([t/x]r), \rho)^{\mathsf{A}}$, $S' = S$ and $\Sigma' = \cdot$.

Similarly to the case of rule **sex_rsp** (with $\mathsf{A} \neq \mathsf{I}$), the Dolev-Yao translation does not directly affect the applicability of the rule. We proceed as in that case.

$\boxed{\textbf{sex\_all},\ \mathsf{A}=\mathsf{I}}$ Here $R = R^*, ((\forall x:\tau.\, r), \rho)^{\mathsf{I}}$ and $R' = R^*, (([t/x]r), \rho)^{\mathsf{I}}$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R$	and
$\mathcal{A}_\rho :: \Sigma \Vdash_{\mathsf{I}} \rho$	and
$\mathcal{A}_\forall :: \Sigma \Vdash_{\mathsf{I}} \forall x:\tau.\, r$	by inversion on rules **aac_ext** and **oac_rule** for $\mathcal{A}$,
$\ulcorner\mathcal{A}\urcorner = (\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_\rho\urcorner, \ulcorner\mathcal{A}_\forall\urcorner)$	by definition of the Dolev-Yao translation,
$\mathcal{A}_r :: \Sigma, x:\tau \Vdash_{\mathsf{I}} r$	by inversion on rule **uac_all** for $\mathcal{A}_\forall$,
$\mathcal{A}'_r :: \Sigma \Vdash_{\mathsf{I}} [t/x]r$	by the Term Substitution Lemma 6.17 on $\mathcal{T}_\Sigma$, $\mathcal{T}$ and $\mathcal{A}_r$,
$\ulcorner\mathcal{A}'_r\urcorner = \ulcorner\mathcal{A}_r\urcorner = \ulcorner\mathcal{A}_\forall\urcorner$	by the Invariance Lemma 8.13 on $\mathcal{T}_\Sigma$, $\mathcal{T}$ and $\mathcal{A}_\forall$,
$\ulcorner\mathcal{A}'\urcorner = (\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_\rho\urcorner, \ulcorner\mathcal{A}_r\urcorner)$	by definition of the Dolev-Yao translation,
$\mathcal{E} :: \mathcal{P} \rhd [\ulcorner S\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_\rho\urcorner, \ulcorner\mathcal{A}_\forall\urcorner} \longrightarrow^* [\ulcorner S\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_\rho\urcorner, \ulcorner\mathcal{A}_r\urcorner}$	by rule **sex_it0**.

$\boxed{\textbf{sex\_core},\ \mathsf{A}\neq\mathsf{I}}$

$$\mathcal{E} = \cfrac{\begin{array}{c}\mathcal{E}' \\ (rhs)_\Sigma \gg (lhs')_{\Sigma,\Sigma'}\end{array}}{\mathcal{P} \rhd [S^*, lhs]_{\Sigma}^{R^*,((lhs\to rhs),\rho)^{\mathsf{A}}} \longrightarrow [S^*, lhs']_{\Sigma,\Sigma'}^{R^*,(\rho)^{\mathsf{A}}}}\ \textbf{sex\_core}$$

with $R = R^*, ((lhs \to rhs), \rho)^{\mathsf{A}}$, $R' = R^*, (\rho)^{\mathsf{A}}$, $S = S^*, lhs$ and $S' = S^*, lhs'$.

We proceed again as in the case of rule **sex_rsp** (for $\mathsf{A}\neq\mathsf{I}$). It should be noted that $\ulcorner lhs\urcorner = lhs$ since lhs cannot contain role state or memory predicates whose first argument (resp. index) is I. This is due to the existence of $\mathcal{A}$, which entails that "$\Sigma \Vdash_{\mathsf{A}} (lhs \to rhs)$" is derivable. The formal argument proceeds by induction on the structure of a derivation for this judgment. By a similar argument, $\ulcorner lhs'\urcorner = lhs'$.

$\boxed{\textbf{sex\_core},\ \mathsf{A}=\mathsf{I}}$ Here $R = R^*, ((lhs \to rhs), \rho)^{\mathsf{I}}$ and $R' = R^*, (\rho)^{\mathsf{I}}$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R^*$	and
$\mathcal{A}_\rho :: \Sigma \Vdash_{\mathsf{I}} \rho$	and
$\mathcal{A}_{lhs} :: \Sigma; \cdot \Vdash_{\mathsf{I}} lhs > \cdot \gg \Delta$	and
$\mathcal{A}_{rhs} :: \Sigma; \Delta \Vdash_{\mathsf{I}} rhs$	by inversion on $\mathcal{A}$,
$\ulcorner\mathcal{A}\urcorner = (\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_\rho\urcorner, \ulcorner\mathcal{A}_{lhs}\urcorner, \ulcorner\mathcal{A}_{rhs}\urcorner)$	by definition of the Dolev-Yao translation,
$\mathcal{E}_{lhs} :: \cdot \rhd [\ulcorner lhs\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{lhs}\urcorner} \longrightarrow^* [\ulcorner\Delta\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\cdot}$	by the Left-Hand Side Decomposability Lemmas 8.6 on $\mathcal{A}_{lhs}$,
$\mathcal{E}'_{lhs} :: \cdot \rhd [\ulcorner lhs\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{lhs}, \mathcal{A}_{rhs}\urcorner} \longrightarrow^* [\ulcorner\Delta\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{rhs}\urcorner}$	by the Execution Weakening Lemma 6.4 on $\mathcal{E}_{lhs}$,
$\mathcal{E}_{rhs} :: \cdot \rhd [\ulcorner\Delta\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{lhs}\urcorner} \longrightarrow^* [\ulcorner lhs'\urcorner]_{\Sigma'\oplus\Sigma_{DY}}^{\cdot}$	by the Right-Hand Side Constructability Lemma 8.12 on $\mathcal{A}_{rhs}$ and $\mathcal{E}'_r$,
$\mathcal{E}^* :: \cdot \rhd [\ulcorner lhs\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{lhs}, \mathcal{A}_{rhs}\urcorner} \longrightarrow^* [\ulcorner lhs'\urcorner]_{\Sigma'\oplus\Sigma_{DY}}^{\cdot}$	by the Chaining Lemma 6.1 on $\mathcal{E}'_{lhs}$ and $\mathcal{E}_{rhs}$,
$\mathcal{E}' :: (\ulcorner\mathcal{P}\urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S^*\urcorner, \ulcorner lhs\urcorner]_{\Sigma\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}\urcorner} \longrightarrow^* [\ulcorner S^*\urcorner, \ulcorner lhs'\urcorner]_{\Sigma'\oplus\Sigma_{DY}}^{\ulcorner\mathcal{A}_{R^*}\urcorner, \ulcorner\mathcal{A}_\rho\urcorner}$	by the Execution Weakening Lemma 6.4 on $\mathcal{E}^*$.

$\boxed{\textbf{sex\_skp},\ \mathsf{A}\neq\mathsf{I}}$

$$\mathcal{E} = \cfrac{}{\mathcal{P} \rhd [S]_{\Sigma}^{R^*,(r,\rho)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R^*,(\rho)^{\mathsf{A}}}}\ \textbf{sex\_skp}$$

with $R = R^*, (r, \rho)^{\mathsf{A}}$, $R' = R^*, (\rho)^{\mathsf{A}}$, $S' = S$ and $\Sigma' = \cdot$.

We proceed again as in the of rule **sex_rsp** (for $\mathsf{A} \neq \mathsf{I}$).

$\boxed{\textbf{sex\_skp},\ \mathsf{A} = \mathsf{I}}$ Here $R = R^*, (r, \rho)^{\mathsf{I}}$ and $R' = R^*, (\rho)^{\mathsf{I}}$.

$\mathcal{A}_{R^*} :: \Sigma \Vdash R^*$ and

$\mathcal{A}_\rho :: \Sigma \Vdash_{\mathsf{I}} \rho$ and

$\mathcal{A}_r :: \Sigma \Vdash_{\mathsf{I}} r$ by inversion on rules **aac_ext** and **oac_rule** for $\mathcal{A}$,

$\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}_{R^*} \urcorner, \ulcorner \mathcal{A}_\rho \urcorner, \ulcorner \mathcal{A}_r \urcorner$ by definition of the Dolev-Yao translation,

Observe that $\ulcorner \mathcal{A}_r \urcorner$ is composed of a finite number of copies of roles taken from $\mathcal{P}_{DY}$. In particular, each of these roles consists of a single rule, and no role state predicate declarations are present. Assume therefore that $\ulcorner \mathcal{A}_r \urcorner$ is given by the following sequence of one-rule roles $(r_1, \cdot)^{\mathsf{I}}, \ldots, (r_n, \cdot)^{\mathsf{I}}$, where we have included the trailing "$\cdot$" for completeness.

Intuitively, we emulate the action of rule **sex_skp** on r as follows: for each component $(r_i, \cdot)^{\mathsf{I}}$ in $\ulcorner \mathcal{A} \urcorner$, we apply rule **sex_skp** to produce the empty role $(\cdot)^{\mathsf{I}}$, which we immediately eliminate thanks to rule **sex_dot**. The formal proof proceeds by induction on the structure of $\ulcorner \mathcal{A}_r \urcorner$. We omit it for the sake of conciseness.

Therefore,

$\mathcal{E}' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \triangleright [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A}_{R^*} \urcorner, \ulcorner \mathcal{A}_\rho \urcorner, \ulcorner \mathcal{A}_r \urcorner} \longrightarrow^* [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A}_{R^*} \urcorner, \ulcorner \mathcal{A}_\rho \urcorner}$ by applications of rules **sex_skp** and **sex_dot**.

$\boxed{\textbf{sex\_dot},\ \mathsf{A} \neq \mathsf{I}}$ $\mathcal{E} = \dfrac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R', (\cdot)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R'}}\,\textbf{sex\_dot}$

with $R = R', (\cdot)^{\mathsf{A}}$, $S' = S$ and $\Sigma' = \cdot$.

$\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}' \urcorner, (\cdot)^{\mathsf{A}}$ by definition of the Dolev-Yao translation,

$\mathcal{E}' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \triangleright [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A} \urcorner, (\cdot)^{\mathsf{A}}} \longrightarrow^* [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A} \urcorner}$ by rule **sex_dot**.

$\boxed{\textbf{sex\_skp},\ \mathsf{A} = \mathsf{I}}$ Here $R = R', (\cdot)^{\mathsf{I}}$.

$\ulcorner \mathcal{A} \urcorner = \ulcorner \mathcal{A}' \urcorner, \ulcorner (\cdot)^{\mathsf{I}} \urcorner = \ulcorner \mathcal{A}' \urcorner$ by definition of the Dolev-Yao translation,

$\mathcal{E}' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \triangleright [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A} \urcorner} \longrightarrow^* [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A}' \urcorner}$ by rule **sex_it0**.

$\boxed{\textbf{sex\_it0}}$ $\mathcal{E} = \dfrac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R} \longrightarrow^* [S]_{\Sigma}^{R}}\,\textbf{sex\_it0}$

with $R' = R$, $S' = S$ and $\Sigma' = \cdot$.

$\mathcal{E}' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \triangleright [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A} \urcorner} \longrightarrow^* [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A} \urcorner}$ by rule **sex_it0**.

$\boxed{\textbf{sex\_itn}}$ $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\mathcal{P} \triangleright [S]_{\Sigma}^{R} \longrightarrow [S^*]_{\Sigma, \Sigma_1'}^{R^*}} \quad \overset{\mathcal{E}_1}{\mathcal{P} \triangleright [S^*]_{\Sigma, \Sigma_1'}^{R^*} \longrightarrow^* [S']_{\Sigma, \Sigma_1', \Sigma_2'}^{R'}}}{\mathcal{P} \triangleright [S]_{\Sigma}^{R} \longrightarrow^* [S']_{\Sigma, \Sigma_1', \Sigma_2'}^{R'}}\,\textbf{sex\_itn}$

with $\Sigma' = \Sigma_1', \Sigma_2'$.

$\mathcal{E}_1' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \triangleright [\ulcorner S \urcorner]_{\Sigma \oplus \Sigma_{DY}}^{\ulcorner \mathcal{A} \urcorner} \longrightarrow^* [\ulcorner S^* \urcorner]_{\Sigma \oplus \Sigma_{DY}, \Sigma_1^*}^{\ulcorner \mathcal{A}_{R^*} \urcorner}$ by induction hypothesis on $\mathcal{E}_1$,

$\mathcal{E}_2' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \triangleright [\ulcorner S^* \urcorner]_{\Sigma \oplus \Sigma_{DY}, \Sigma_1'}^{\ulcorner \mathcal{A}_{R^*} \urcorner} \longrightarrow^* [\ulcorner S' \urcorner]_{\Sigma \oplus \Sigma_{DY}, \Sigma_1', \Sigma_2^*}^{\ulcorner \mathcal{A}' \urcorner}$ by induction hypothesis on $\mathcal{E}_2$,

$$\begin{array}{ll}
\mathcal{T}_{\Sigma_{DY}} ::\vdash \Sigma \oplus \Sigma_{DY} & \text{by the Typability Lemma 8.15 on } \mathcal{T}_{\Sigma}, \\
\mathcal{T}_{\ulcorner S \urcorner} :: \Sigma \oplus \Sigma_{DY} \vdash \ulcorner S \urcorner & \text{by the Typability Lemma 8.15 on } \mathcal{T}_{S}, \\
\mathcal{T}_{\ulcorner \mathcal{A} \urcorner} :: \Sigma \oplus \Sigma_{DY} \vdash \ulcorner \mathcal{A} \urcorner & \text{by the Typability Lemma 8.15 on } \mathcal{A}, \\
\mathcal{T}^*_{\Sigma_{DY}} ::\vdash \Sigma \oplus \Sigma_{DY}, \Sigma^*_1 & \text{and} \\
\mathcal{T}^*_{\ulcorner S \urcorner} :: \Sigma \oplus \Sigma_{DY} \vdash \ulcorner S^* \urcorner & \text{and} \\
\mathcal{T}^*_{\ulcorner \mathcal{A} \urcorner} :: \Sigma \oplus \Sigma_{DY} \vdash \ulcorner \mathcal{A}_{R^*} \urcorner & \text{by the Type Preservation Theorem 6.15 on } \mathcal{T}_{\Sigma_{DY}}, \mathcal{T}_{\ulcorner S \urcorner} \text{ and } \mathcal{T}_{\ulcorner \mathcal{A} \urcorner}, \\
\mathcal{E}^*_2 :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S^* \urcorner]^{\ulcorner \mathcal{A}_{R^*} \urcorner}_{\Sigma \oplus \Sigma_{DY}, \Sigma^*_1} \longrightarrow^* [\ulcorner S' \urcorner]^{\ulcorner \mathcal{A}' \urcorner}_{\Sigma \oplus \Sigma_{DY}, \Sigma^*_1, \Sigma^*_2} & \text{by the Deletion Lemma 8.16 on } \mathcal{T}^*_{\Sigma_{DY}}, \mathcal{T}^*_{\ulcorner S \urcorner}, \mathcal{T}^*_{\ulcorner \mathcal{A} \urcorner} \text{ and } \mathcal{E}'_2, \\
\mathcal{E}' :: (\ulcorner \mathcal{P} \urcorner, \mathcal{P}_{DY}) \rhd [\ulcorner S \urcorner]^{\ulcorner \mathcal{A} \urcorner}_{\Sigma \oplus \Sigma_{DY}} \longrightarrow^* [\ulcorner S' \urcorner]^{\ulcorner \mathcal{A}' \urcorner}_{\Sigma \oplus \Sigma_{DY}, \Sigma^*_1, \Sigma^*_2} & \text{by the Chaining Lemma 6.1 on } \mathcal{E}'_1 \text{ and } \mathcal{E}^*_2.
\end{array}$$

This concludes the proof of this theorem. □

Since, in models that relies on black-box cryptography, an attack of any kind is ultimately an execution sequence between two snapshots, 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 [10, 43, 49, 51, 54, 57, 58], 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 [16, 42, 50, 57] and on per-protocol specializations, for example allowing the intruder to construct only message patterns actually used in the protocol [51, 54]. 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 [59] to reduce them to a single attacker I

8.3 An Optimized Dolev-Yao Intruder

We will now present an optimized variant of the Dolev-Yao intruder discussed above. Whenever the adversary intercepts a message, we will have him decompose it in its most elementary bits, store them in a dedicated memory predicate, and construct any message intended for transmission from stored elementary terms. We can therefore partition the actions of the intruder in three distinct activities: message decomposition, storage of elementary information, and message construction. This idea was first proposed in [50] and analyzed in an earlier version of *MSR* in [27].

In order to formalize this idea in *MSR*, it is convenient to slightly revise the subsorting relation presented in Section 2.2. Namely, we introduce the sort atm which stands for atomic messages. We make all types classifying elementary information a subsort of atm:

$$\frac{}{\mathsf{principal} :: \mathsf{atm}}\,\mathbf{ss'\_pr} \qquad \frac{}{\mathsf{nonce} :: \mathsf{atm}}\,\mathbf{ss'\_nnc}$$

$$\frac{}{\mathsf{shK}\ A\ B :: \mathsf{atm}}\,\mathbf{ss'\_shK} \qquad \frac{}{\mathsf{pubK}\ A :: \mathsf{atm}}\,\mathbf{ss'\_pbK}$$

while atm is made a subsort of msg:

$$\frac{}{\mathsf{atm} :: \mathsf{msg}}\,\mathbf{ss'\_atm}$$

We leave it as an exercise to the reader to update the typing rules presented in Section 2.3 and the proof of the various results in this report.

We replace the single memory predicate $\mathsf{M_I}(\_)$ used in Section 8.1 with three predicates, $\mathsf{D_I}(\_)$, $\mathsf{A_I}(\_)$ and $\mathsf{C_I}(\_)$. The first is intended to contain messages while they are decomposed into their elementary constituents. The second holds the atomic terms learned in this way. The third is used in the message construction phase. Our signature shall therefore contain the following four declarations:

$$\mathsf{I} : \mathsf{principal} \quad , \quad \mathsf{D\_} : \mathsf{principal} \times \mathsf{msg} \quad , \quad \mathsf{A\_} : \mathsf{principal} \times \mathsf{atm} \quad \text{and} \quad \mathsf{C\_} : \mathsf{principal} \times \mathsf{msg}$$

The interception and transmission rules are updated as follows:

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

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

The rules that dealt with composite message in Section 8.1 are adapted by replacing the generic M_ predicate with the appropriate refinement. When decomposing a message, its components may need further disassembling. Dually, messages intended for transmission are built by putting together constructable pieces. Keys constitute an exception to this rule: they are clearly atomic and therefore can be accessed from the A_ predicate where such information is stored. We also need to copy them to the consequent of rules so that the intruder can use them again for other encryptions (as we will see shortly, this optimized scheme does without an explicit duplication rule).

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

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

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

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

Once a captured message has been reduced to its atomic constituents, they are memorized in individual A_ predicates by the following rule:

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

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

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

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

It should be observed that this rule provides a loophole in the scheme discussed in this section since it allows any message to transit from the decomposition pool to the construction queue without accessing its atomic components. To avoid this, it would be tempting to specialize this rule to the situations where t is indeed an encrypted message. This would however violate the data access specification policy.

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

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

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

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

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

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

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

It should be noted that we have structured the above rules in such a way that no explicit copying rule is ever needed: whenever atomic or decomposable information is accessed for constructing an outgoing message, we always leave a copy for future use. On a similar note, we omit the deletion rule of Section 8.1: this version of the Dolev-Yao intruder only accumulate information, never eliminates it.

Proving the correctness of this optimized intruder model with respect to the role set presented in Section 8.1 is a rather simple task: indeed, mapping the specialized predicates $\mathsf{D_I}(\_)$, $\mathsf{A_I}(\_)$ and $\mathsf{C_I}(\_)$ back to $\mathsf{M_I}(\_)$ yields a set of rules that is almost identical to our original role set for the Dolev-Yao intruder. The minor discrepancies brought about by this translation are corrected by uses of the structural rule of deletion, and the elimination of one redundantly produced rule, whose antecedent and consequent are identical.

The proof of the corresponding completeness result, which shows that our optimized model is powerful enough to simulate the original Dolev-Yao intruder, is a direct consequence of the Most Powerful Attacker Theorem 8.17.

9 Examples

In this section, we will demonstrate the expressive power of *MSR* by formalizing a number of examples. We start in Section 9.1 with what is probably the most popular case-study in the security protocol analysis community: the Needham-Schroeder public-key authentication protocol [52]. Our second example, in Section 9.2, studies a slight simplification of the Neuman-Stubblebine repeated authentication protocol [53], which is particularly interesting since it consists of two phases, the second of which can be repeated arbitrarily many times. Our final and largest example, in Section 9.3, formalizes the *OFT* key management algorithms [6], a proposed standard for the hierarchical management of keys in large multicast groups.

In all our examples, we will rely on the visual layout for rules and roles that we already used in Section 8. In particular, recall that we mark types that can be reconstructed from the other information present in a rule by denoting them in a shaded font.

9.1 The Needham-Schroeder Public-Key Authentication Protocol

As our first example using *MSR* as a specification language, we will formalize the infamous Needham-Schroeder public-key authentication protocol [52]. We familiarize the reader with *MSR* by first considering the two-party nucleus of this protocol in Section 9.1.1. Then, in Section 9.1.2, we tackle a variant of the full protocol, which relies on a server to generate session keys.

9.1.1 Simplified Protocol

The server-less variant of the Needham-Schroeder public-key protocol [52] 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 following expected run in the "usual notation" (where we have used our syntax for messages):

$$\begin{array}{ll}
1. & A \rightarrow B\text{: } \{\!\{n_A\ A\}\!\}_{k_B}\\
2. & B \rightarrow A\text{: } \{\!\{n_A\ n_B\}\!\}_{k_A}\\
3. & A \rightarrow B\text{: } \{\!\{n_B\}\!\}_{k_B}
\end{array}$$

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.

MSR and most modern security protocol specification languages describe the sequence of actions that each principal involved in a protocol executes. We called such sequences roles. Strand spaces [47] are a simple and intuitive notation that emphasize this notion. The following strand representation of this protocol is given by the following picture:

$$\begin{tikzcd}
 & \textit{Initiator} & & & \textit{Responder} & \\
 & \{\!\{n_A\ A\}\!\}_{k_B} \arrow[d, Rightarrow] & \longrightarrow & \longrightarrow & \{\!\{n_A\ A\}\!\}_{k_B} \arrow[d, Rightarrow] & \\
\longrightarrow & \{\!\{n_A\ n_B\}\!\}_{k_A} \arrow[d, Rightarrow] & & & \{\!\{n_A\ n_B\}\!\}_{k_A} \arrow[d, Rightarrow] & \longrightarrow \\
 & \{\!\{n_B\}\!\}_{k_B} & \longrightarrow & \longrightarrow & \{\!\{n_B\}\!\}_{k_B} &
\end{tikzcd}$$

Here incoming and outgoing single arrows respectively denote the reception and transmission of a message. The double arrows assign a temporal ordering on these actions.

We will now express each role in turn in the syntax of *MSR*. The initiator's actions are represented by the following role:

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

Clearly, since any principal can engage in this protocol as an initiator (or a responder), our encoding should be structured as a generic role. Let A be its owner. The first rule formalizes of the first line 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. 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: it corresponds to the topmost double arrow in the strand specification on the left. The arguments of this predicate record all the variables used in this rule.

The second rule encodes the last two lines 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. The presence of both a message receptions and transmission in the same rule corresponds to the second double arrow in the strand specification of the initiator of this role.

Our notation provides a specific type for 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. Observe that most declarations are grayed out, meaning that they can be reconstructed automatically: 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 rationale behind the constructable types in this rule are as follows. The universal declarations for B, k_B, and n_A and the type of the existential declaration for n_A in the first rule can be deduced from the declaration of the role state predicate L. The declarations for k_A and k'_A can be omitted since the data access specification policy requires that k_A be the public key of A, and k'_A be the corresponding private key. The only universal declaration that cannot be reconstructed is "$\forall n_B$: nonce": n_B is clearly a universally quantified variable in this rule, but there is no hint that it should be a nonce. Let us now examine the declaration for L: the first argument is always the rule owner, which is a principal. The third argument must be the public key of some principal B because of the way k_B is used. Therefore, we only need to indicate that B is bound in the second argument of L.

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 no consequent.

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

Again, observe that most typing information has been grayed out since it can be reconstructed from the way variables are used and the few types left.

9.1.2 Full Protocol

We will now specify a variant of the full version of the Needham-Schroeder public-key authentication protocol [52], which relies on a server S to generate the keys k_A and k_B used in the fragment discussed in the previous section. This protocol is written as follows in the "usual notation":

$$\begin{array}{rl}
1. & A \rightarrow \mathsf{S}\colon\ A\, B \\
2. & \mathsf{S} \rightarrow A\colon\ \{k_B\, B\}_{k_{A\mathsf{S}}} \\
3. & A \rightarrow B\colon\ \{\!\!\{n_A\, A\}\!\!\}_{k_B} \\
4. & B \rightarrow \mathsf{S}\colon\ B\, A \\
5. & \mathsf{S} \rightarrow B\colon\ \{k_A\, A\}_{k_{B\mathsf{S}}} \\
6. & B \rightarrow A\colon\ \{\!\!\{n_A\, n_B\}\!\!\}_{k_A} \\
7. & A \rightarrow B\colon\ \{\!\!\{n_B\}\!\!\}_{k_B}
\end{array}$$

The simplified version discussed in Section 9.1.1 corresponds to lines (3), (6) and (7) of this protocol. In line (1), A asks the server for a key to communicate with B, which is obtained in line (2). The responder issues and is granted a similar request in lines (4) and (5), respectively. This protocol differs from the original published in [52] in that we used shared-key encryption rather than digital signatures as the format of the server's responses. This small deviation allows us to treat this example while remaining within the bounds of the syntax presented in this report. A simple extension of our notion of terms would allow expressing the original form of this protocol.

The actions of the initiator are expressed in *MSR* by the following generic role, which consists of three rules that have to fire in sequence, and consequently mentions two role state predicate declarations. The first rule corresponds to line (1) in the "usual notation", the second to lines (2) and (3), and the third to lines (6) and (7).

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

Observe again that most declarations and types can be reconstructed. Notice in particular that, since in the second rule the arguments of L form a prefix of the arguments of L', the entire declaration for L can be synthesized from the type of L'.

The responder's actions are expressed in the following generic role. The first rule corresponds to lines (3) and (4) in the "usual notation", the second to lines (5) and (6), and the third to line (7). Observe again that most declarations can be automatically reconstructed.

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

The last role in this protocol encompasses the actions of the server. Assuming that there is a single server, S, they can conveniently be expressed by the following anchored role, which consists of a single rule. The "usual notation" specification of this protocol makes use of this role twice: in lines (1) and (2) to create B's keys for A, and then in lines (4) and (5) for the dual operation.

$$\left(\begin{array}{lccc}
\begin{array}{l}\forall A, B : \mathsf{principal}.\\ \forall k_{A\mathsf{S}} : \mathsf{shK}\ A\,\mathsf{S}.\\ \forall k_B : \mathsf{pubK}\ B.\end{array} & \mathsf{N}(A\,B) & \rightarrow & \mathsf{N}(\{k_B\ B\}_{k_{A\mathsf{S}}})
\end{array}\right)^{\mathsf{S}}$$

Upon receiving a message of the form $\mathsf{N}(A\,B)$, the server retrieves the public key k_B of the principal B and notifies A by sending the message $\mathsf{N}(\{k_B\ B\}_{k_{A\mathsf{S}}})$. The binding between B and k_B is elegantly achieved through the use of dependent types. This approach allows us to keep the details of the retrieval abstract.

Notice that very little information can be reconstructed in this role.

9.2 The Neuman-Stubblebine Repeated Authentication Protocol

In this section, we provide an *MSR* of a fragment of the Neuman-Stubblebine repeated authentication protocol [53]. Similarly to Kerberos, this protocol consists of two phases. In a first phases, a principal A negotiates a "ticket" with a server in order to use services provided by another principal B. In the second phases, A can reuse the ticket over and over to request this same service from B. The original protocol includes timestamps that we ignore in this specification. These entities require extensions to *MSR* that go beyond the scope of this report. The two phases of this protocol are examined in Sections 9.2.1 and 9.2.2 respectively.

9.2.1 Initialization Subprotocol

The Neuman-Stubblebine protocol [53] is intended to enable a principal A to repeatedly authenticate herself to another principal B. Typically, B provides a service that A is interested in using over and over. Each time A intends to use this service, she authenticates herself to B by presenting a ticket he is expected to honor.

The initialization phase of this protocol involves an interaction with a server S to obtain the ticket, as well as its first use to request the service provided by B. We have the following expected trace in the usual notation:

$$\begin{array}{ll}
1. & A \rightarrow B\colon\ A\, n_A \\
2. & B \rightarrow \mathsf{S}\colon\ B\, \{A\, n_A\, T_B\}_{k_{B\mathsf{S}}}\, n_B \\
3. & \mathsf{S} \rightarrow A\colon\ \{B\, n_A\, k_{AB}\, T_B\}_{k_{A\mathsf{S}}}\, \{A\, k_{AB}\, T_B\}_{k_{B\mathsf{S}}}\, n_B \\
4. & A \rightarrow B\colon\ \{A\, k_{AB}\, T_B\}_{k_{B\mathsf{S}}}\, \{n_B\}_{k_{AB}}
\end{array}$$

In the first line, A manifests her intention to use B's service by sending him her identity and a nonce n_A. In the second line, B forwards this information to the server S together with his identity, a nonce of his own n_B, and a time stamp T_B. In the third line, the server constructs the ticket $\{A k_{AB} T_B\}_{k_{B\mathsf{S}}}$ by combining A's name, a freshly generated key for communication between A and B, and B's time stamp. It is encrypted with the key $k_{B\mathsf{S}}$ that B shares with S so that A cannot modify it. The server also informs A of the extremes of the ticket in the message $\{B\, n_A\, k_{AB}\, T_B\}_{k_{A\mathsf{S}}}$ and forwards her B's nonce. In the last line, A identifies herself to B by sending him the ticket and his nonce n_B encrypted with the newly created k_{AB}.

This initialization subprotocol is encoded in *MSR* by means of three roles, one for A, one for B, and one for S. We start by giving a specification of A's actions, reported in the following role:

$$\left(\begin{array}{llcll}
\multicolumn{5}{l}{\exists L : \mathsf{principal} \times \mathsf{nonce}.} \\
 & \cdot & \rightarrow & \exists n_A\colon \mathsf{nonce}. & \begin{array}{l}\mathsf{N}(A\, n_A) \\ L(A, n_A)\end{array} \\
\begin{array}{l}\forall B\colon \mathsf{principal}. \\ \forall n_A, n_B : \mathsf{nonce}. \\ \forall k_{AB} : \mathsf{shK}\ A\, B. \\ \forall k_{A\mathsf{S}} : \mathsf{shK}\ A\, \mathsf{S}. \\ \forall X : \mathsf{msg}.\end{array} & \begin{array}{l}\mathsf{N}(\{B\, n_A\, k_{AB}\}_{k_{A\mathsf{S}}}\, X\, n_B) \\ L(A, n_A)\end{array} & \rightarrow & & \begin{array}{l}\mathsf{N}(X\, \{n_B\}_{k_{AB}}) \\ \mathsf{M}_A(B, X, k_{AB})\end{array}
\end{array}\right)^{\forall A}$$

The first rule is a straightforward encoding of line (1) of the "usual notation" description of this subprotocol. The more interesting second rule corresponds to lines (3) and (4). Observe that A is not entitled to observe the inner structure of the ticket. We express this fact by placing the variable X in the second component of the received message. Expanding this object as $\{A\, k_{AB}\, T_B\}_{k_{B\mathsf{S}}}$ to expose its structure would violate the data access specification policy. In the consequent of this same rule, A sends the message on line (4) of the informal presentation to B. She also needs to memorize some information to be able to reuse the ticket in the future, namely the ticket itself, the associated key k_{AB}, and B's identity. This is achieved by means of the memory predicate $\mathsf{M}\_$. The type of this predicate is $\mathsf{principal}^{(A)} \times \mathsf{principal}^{(B)} \times \mathsf{msg} \times \mathsf{shK}\ A\, B$.

The responder's actions in this subprotocol are specified by the following role. Its two rules correspond to lines (1) and

(2), and line (4) of the “usual notation” specification above.

$$\left(\begin{array}{llll}
\multicolumn{4}{l}{\exists L : \mathsf{principal}^{(B)} \times \mathsf{principal} \times \mathsf{nonce} \times \mathsf{shK}\ B\ \mathsf{S} \times \mathsf{nonce} \times \mathsf{nonce}.} \\
\begin{array}{l} \forall A : \mathsf{principal}. \\ \forall n_A : \mathsf{nonce}. \\ \forall k_{B\mathsf{S}} : \mathsf{shK}\ B\ \mathsf{S}. \end{array} & \mathsf{N}(A\ n_A) & \to\ \exists n_B, T_B : \mathsf{nonce}. & \begin{array}{l} \mathsf{N}(B\ \{A\ n_A\ T_B\}_{k_{B\mathsf{S}}}\ n_B) \\ L(B, A, n_A, k_{B\mathsf{S}}, n_B, T_B) \end{array} \\
\begin{array}{l} \forall \ldots \\ \forall k_{AB} : \mathsf{shK}\ A\ B \\ \forall n_B, T_B : \mathsf{nonce}. \end{array} & \begin{array}{l} \mathsf{N}(\{A\ k_{AB}\}_{k_{B\mathsf{S}}}\ \{n_B\}_{k_{AB}}) \\ L(B, A, n_A, k_{B\mathsf{S}}, n_B, T_B) \end{array} & \to & \cdot
\end{array}\right)^{\forall B}$$

Observe that we have treated the timestamp T_B as if it were a nonce.

The responder B does not need to memorize any data to provide further instances of the service needed by A. Indeed, the ticket A holds contain all the necessary information to re-authenticate her to B.

Finally, we have a single rule that formalizes the actions of the server. Upon receiving a request from B, the server generates the shared key k_{AB}, constructs the ticket and the notification message for A, and transmits this information.

$$\left(\begin{array}{llll}
\begin{array}{l} \forall A, B : \mathsf{principal}. \\ \forall k_{A\mathsf{S}} : \mathsf{shK}\ A\ \mathsf{S}. \\ \forall k_{B\mathsf{S}} : \mathsf{shK}\ B\ \mathsf{S}. \\ \forall n_A, n_B, T_B : \mathsf{nonce}. \end{array} & \mathsf{N}(B\ \{A\ n_A\ T_B\}_{k_{B\mathsf{S}}}\ n_B) & \to\ \exists k_{AB} : \mathsf{shK}\ A\ B. & \mathsf{N}(\{B\ n_A\ k_{AB}\ T_B\}_{k_{A\mathsf{S}}}\ \{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}\ n_B)
\end{array}\right)^{\mathsf{S}}$$

9.2.2 Repeated Authentication Subprotocol

The second phase of the Neuman-Stubblebine protocol allows the initiator A to repeatedly use the ticket she has acquired in the first phase to access the service provided by B. It is expressed in the “usual notation” by the following three-step subprotocol:

1. $A \to B$: $n'_A\ \{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}$
2. $B \to A$: $n'_B\ \{n'_A\}_{k_{AB}}$
3. $A \to B$: $\{n'_B\}_{k_{AB}}$

In the first line, A generates a nonce n'_A and sends it to B together with the ticket $\{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}$. Upon receiving this message, B creates a nonce of his own n'_B and transmits it to A in line (2) together with the encryption of n'_A with the key k_{AB} embedded in the ticket. In the last line, A sends B's nonce back after encrypting it with their shared key k_{AB}.

This subprotocol is formalized in *MSR* by means of the two roles below. The initiator's actions are expressed by the following generic role. In the first rule, corresponding to line (1) of the informal specification, A accesses the ticket she has stored in the memory predicate M_ during the initialization phase of this protocol. The second rule corresponds to the remaining lines of the “usual notation” specification.

$$\left(\begin{array}{llll}
\multicolumn{4}{l}{\exists L : \mathsf{principal}^{(A)} \times \mathsf{principal}^{(B)} \times \mathsf{msg} \times \mathsf{shK}\ A\ B \times \mathsf{nonce}.} \\
\begin{array}{l} \forall B : \mathsf{principal}. \\ \forall X : \mathsf{msg}. \\ \forall k_{AB} : \mathsf{shK}\ A\ B. \end{array} & \mathsf{M}_A(B, X, k_{AB}) & \to\ \exists n'_A : \mathsf{nonce}. & \begin{array}{l} \mathsf{N}(n'_A\ X) \\ \mathsf{M}_A(B, X, k_{AB}) \\ L(A, B, X, k_{AB}, n'_A) \end{array} \\
\begin{array}{l} \forall \ldots \\ \forall n'_A, n'_B : \mathsf{nonce}. \end{array} & \begin{array}{l} \mathsf{N}(n'_B\ \{n'_A\}_{k_{AB}}) \\ L(A, B, X, k_{AB}, n'_A) \end{array} & \to & \mathsf{N}(\{n'_B\}_{k_{AB}})
\end{array}\right)^{\forall A}$$

The actions of the service provider B are given by the following generic role. Its first rule captures lines (1) and (2) of the informal specification, while the second rule formalizes the remaining line.

$$\left(\begin{array}{llcll}
\multicolumn{5}{l}{\exists L : \mathsf{principal}^{(B)} \times \mathsf{principal}^{(A)} \times \mathsf{nonce} \times \mathsf{shK}\ B\ \mathsf{S} \times \mathsf{shK}\ A\ B \times \mathsf{nonce} \times \mathsf{nonce}.} \\
\begin{array}{l}\forall n'_A, T_B : \mathsf{nonce}.\\ \forall k_{B\mathsf{S}} : \mathsf{shK}\ B\ \mathsf{S}.\\ \forall A : \mathsf{principal}.\\ \forall k_{AB} : \mathsf{shK}\ A\ B.\end{array} & \mathsf{N}(n'_A\ \{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}) & \to & \exists n'_B : \mathsf{nonce}. & \begin{array}{l}\mathsf{N}(n'_B\ \{n'_A\}_{k_{AB}})\\ L(B, A, n'_A, k_{B\mathsf{S}}, k_{AB}, n'_B, T_B)\end{array} \\
\begin{array}{l}\forall \ldots\\ \forall n'_B : \mathsf{nonce}.\end{array} & \begin{array}{l}\mathsf{N}(\{n'_B\}_{k_{AB}})\\ L(B, A, n'_A, k_{B\mathsf{S}}, k_{AB}, n'_B, T_B)\end{array} & \to & & .
\end{array}\right)^{\forall B}$$

This concludes our *MSR* specification of the Neuman-Stubblebine repeated authentication protocol. The two phases that constitute it have been modeled by providing two sets of roles. The connection between them is given by the memory predicate M_ used by the client A. It should be noted that this protocol lies outside of the scope of the previous version of *MSR* [27, 30], which did not provide any means to share data across different roles.

9.3 Group Key Management

Our last and most complex example will involve using *MSR* to formalize aspects of the *OFT* group key management protocol [6]. This case study will demonstrate working with complex data structures such as trees and lists, realizing primitive recursion in *MSR*, and defining non-standard cryptographic operators.

We describe *OFT* in Section 9.3.1, define the syntax we will use for it as well as its typing rules in Section 9.3.2, and explain how we represent trees and related data structures in an *MSR* state in Section 9.3.3. We will formalize two subprotocols of *OFT*: the joining protocol in Section 9.3.4 and the eviction protocol in Section 9.3.5.

9.3.1 Problem Description

The *OFT* protocol suite [6] was designed for establishing a common key shared among all the members of a large group of principals, and maintaining it as members join and leave. Intended applications include pay TV, secure teleconferencing, and military command and control. These applications have several characteristics in common: they are real-time, they involve potentially large groups, and membership is highly dynamic as members join and leave the group frequently. Therefore, the group key may change several times per second, which requires providing members with the most current key frequently. Recomputing and redistributing the group key must be done efficiently, even for very large groups, or the key management overhead may be a bottleneck for applications.

OFT relies on two devices to achieve high efficiency. First it logically arranges the members of a group and the needed auxiliary keys in a tree, which allows it to carry out key updates in logarithmic time in the size of the group. Second, it relies on multicast to send key updates to all members of the group with a single message. Group members maintain some state, which is also logarithmic in the size of the group.

Each group is managed by a *group manager*, which maintains a *one-way function tree* for the group (hence the name *OFT*). This is a binary tree with each leaf associated with a group member. Each node n, whether a leaf or an inner node, has an unblinded key k_n. In the case of leafs, this key is communicated directly by the group manager to the corresponding member using a shared key. In the case of an inner node n, the key k_n is computed on the basis of the keys k_{left} and k_{right} of its two children using the mixing function f and the one-way function g as follows:

$$k \;=\; \mathsf{f}(\mathsf{g}(k_{left}), \mathsf{g}(k_{right}))$$

The group key is the unblinded key associated with the root of the tree. While the group manager has a complete view of the tree and of all the keys in it, each member A knows only the unblinded key k_A of her own leaf node, and the blinded key $k'_s = \mathsf{g}(k_s)$ of the sibling of every node n_s on a path from her leaf node to the root of the tree. This allows A to recursively compute the unblinded key of every node from n_A to the root. Indeed, given the unblinded key k_A of her own leaf node and the blinded key k'_s of her sibling, she can compute the unblinded key of her parent node as $k_p = \mathsf{f}(\mathsf{g}(k_A), k'_s)$.

Similarly, for any node n on this path for which she has computed the unblinded key k_n, she can compute the unblinded key $k_p = \mathsf{f}(\mathsf{g}(k_n), k'_s)$ of its parent knowing the blinded key k'_s of its sibling. In particular, this procedure gives A a way to learn the group key. Note however that she has no way of learning the keys of any other node in the tree.

In this section, we will assume that there is a single group, although *OFT* is more general, and write GM for its group manager. We will focus on the main activities of GM: processing joining requests and evicting members. Both involve computing a new group key. This will be done by giving one or more members a new unblinded key, which will trigger changes to the unblinded keys associated to every node along the path to the root. Using a multicast message, GM will send the corresponding sequence of unblinded keys to all members of the group so that they can update their view of the tree and correctly compute the new group key.

9.3.2 Terms

Because *OFT* relies on non-standard data structures, at least as far as cryptographic protocols are concerned, we will need to extend our notion of terms and their typing rules. We will not need to modify *MSR*'s execution rules. In this section, we formalize their syntax and define their typing semantics.

Syntax

To formalize *OFT*, we need to extend allowed atomic messages with a limited number of *tags*, which we will use to signal the meaning of messages sent to group members. We also include *node symbols*, written n, which we will use only in a few of the memory predicates maintained by the group manager. Three new syntactic categories are needed: natural numbers (d) written in unary notation, blinded keys (k'), and unblinded keys (k). A blinded key k' is obtained by applying a fixed one-way function g to an unblinded key k. Unblinded keys are either atomic symbols, which we write k, or are obtained by applying a fixed *mixing function* f to two unblinded keys.

$$\begin{array}{llcll}
\textit{Atomic messages:} & a & ::= & \mathsf{A} & \textit{(Principal)} \\
 & & | & \mathsf{k}_{\mathsf{AB}} & \textit{(Shared key)} \\
 & & | & \mathsf{join} \mid \mathsf{welcome} \mid \mathsf{newkey} \mid \mathsf{rekey} \mid \mathsf{evict} \mid \mathsf{newkey}' \mid \mathsf{shift} & \textit{(Tag)} \\
 & & | & \mathsf{n} & \textit{(Node)} \\
\textit{Natural numbers:} & d & ::= & \mathsf{z} & \textit{(Zero)} \\
 & & | & \mathsf{s}(d) & \textit{(Successor)} \\
\textit{Unblinded keys:} & k & ::= & \mathsf{k} & \textit{(Elementary key)} \\
 & & | & \mathsf{f}(k'_1, k'_2) & \textit{(Mixing)} \\
\textit{Blinded keys:} & k' & ::= & \mathsf{g}(k) & \textit{(Blinding)}
\end{array}$$

For convenience, we will assume that the mixing function f is commutative and that it has a unit that we will write as $\star$. Although more specific than the *OFT* documentation [6], it is consistent with the practice of using XOR as f, so that $\star$ can be taken to be the zero bitstring.

Besides atomic messages, keys (now including blinded and unblinded keys), concatenation and symmetric-key encryption, we allow a term to be obtained by encrypting another term using a blinded key. Types are extended with primitive types to classify tags, natural numbers, nodes, and blinded and unblinded keys. The resulting syntax for terms and types is as follows:

$$\begin{array}{llcll}
\textit{Messages:} & t & ::= & a & \textit{(Atomic messages)} \\
 & & | & k & \textit{(Keys)} \\
 & & | & d & \textit{(Natural numbers)} \\
 & & | & t_1\, t_2 & \textit{(Concatenation)} \\
 & & | & \{t\}_{\mathsf{k}_{AB}} & \textit{(Symmetric-key encryption)} \\
 & & | & \{\!|t|\!\}_{\mathsf{k}} & \textit{(Blinded-key encryption)}
\end{array}$$

$$\begin{array}{lcll} \textit{Types:}\ \ \tau & ::= & \mathsf{principal} \mid \mathsf{shK}\ A\ B & \textit{(Principals and shared keys)} \\ & \mid & \mathsf{tag} & \textit{(Tags)} \\ & \mid & \mathsf{nat} & \textit{(Natural numbers)} \\ & \mid & \mathsf{node} & \textit{(Nodes)} \\ & \mid & \mathsf{uKey} & \textit{(Unblinded keys)} \\ & \mid & \mathsf{bKey} & \textit{(Blinded keys)} \end{array}$$

We introduce a new state predicate $!\mathsf{N}(t)$ to express the multicast of messages to multiple group members. Just like regular network messages, we will use it in the state of the computation and inside rules.

$$\begin{array}{lcll} \textit{States:}\ \ S & ::= & \cdot \mid S,\ \mathsf{N}(t) \mid S,\ \mathsf{L}_l(\vec{t}) \mid S,\ \mathsf{M_A}(\vec{t}) & \\ & \mid & S,\ !\mathsf{N}(t) & \textit{(Multicast network predicate)} \end{array}$$

Typing

The typing rules for the syntactic entities added to the language are easily defined. Tags, natural numbers and both types of new keys are subsorts of msg (nodes are not because they are used in internal data structures and never sent on the network).

$$\frac{}{\mathsf{tag} :: \mathsf{msg}}\,\textbf{ss\_tag} \qquad \frac{}{\mathsf{nat} :: \mathsf{msg}}\,\textbf{ss\_nat} \qquad \frac{}{\mathsf{uKey} :: \mathsf{msg}}\,\textbf{ss\_uk} \qquad \frac{}{\mathsf{bKey} :: \mathsf{msg}}\,\textbf{ss\_bk}$$

They are typable in any signature, and so are nodes.

$$\frac{}{\Sigma \vdash \mathsf{tag}}\,\textbf{ttp\_tag} \qquad \frac{}{\Sigma \vdash \mathsf{nat}}\,\textbf{ttp\_nat} \qquad \frac{}{\Sigma \vdash \mathsf{uKey}}\,\textbf{ttp\_uk} \qquad \frac{}{\Sigma \vdash \mathsf{bKey}}\,\textbf{ttp\_bk} \qquad \frac{}{\Sigma \vdash \mathsf{node}}\,\textbf{ttp\_bk}$$

The typing rule for blinded-key encryption is unsurprising:

$$\frac{\Sigma \vdash t : \mathsf{msg} \quad \Sigma \vdash k : \mathsf{bKey}}{\Sigma \vdash \{\!|t|\!\}_k : \mathsf{msg}}\,\textbf{mtp\_uke}$$

The following typing rules express the expected typing relations for the constructors of natural numbers and blinded and unblinded keys.

$$\frac{}{\Sigma \vdash \mathsf{z} : \mathsf{nat}}\,\textbf{mtp\_z} \qquad \frac{\Sigma \vdash d : \mathsf{nat}}{\Sigma \vdash \mathsf{s}(d) : \mathsf{nat}}\,\textbf{mtp\_s} \qquad \frac{\Sigma \vdash k'_1 : \mathsf{uKey} \quad \Sigma \vdash k'_2 : \mathsf{uKey}}{\Sigma \vdash \mathsf{f}(k'_1, k'_2) : \mathsf{bKey}}\,\textbf{mtp\_f} \qquad \frac{\Sigma \vdash k : \mathsf{bKey}}{\Sigma \vdash \mathsf{g}(k) : \mathsf{uKey}}\,\textbf{mtp\_g}$$

Multicast messages are typed similarly to normal (unicast) network messages.

$$\frac{\Sigma \vdash t : \mathsf{msg}}{\Sigma \vdash !\mathsf{N}(t)}\,\textbf{ptp\_!net}$$

The typing rules of other constructs are as in the rest of this document.

9.3.3 Data Structures

Both the group manager GM and the group members need to maintain data structures to support *OFT*. Because these data structures extend beyond the sessions of a single (sub)protocol, we rely on memory predicates to express them in a relational fashion. As will become apparent shortly, principals will also use memory predicates to process these data structures in a recursive manner.

Observe that, in Section 8, the intruder memory predicate I was used both to hold intruder data and to process it. There, memorized messages were independent from each other, yet the structure of the messages in the intruder memory predicate drove the recursive procedures to take them apart.

Group Manager

The group manager relies on the following three memory predicates to represent the *OFT* tree in a relational way.

$\mathsf{Root}_{\mathsf{GM}}(n)$	*Node n is the root of the tree*
$\mathsf{Node}_{\mathsf{GM}}(n, k, n_p)$	*Node n has key k and parent n_p*
$\mathsf{Member}_{\mathsf{GM}}(n, d, A)$	*Member A occupies node n at depth d*

The memory predicate $\mathsf{Root}_{\mathsf{GM}}(n)$ designates node n as the root of the tree. The memory predicate $\mathsf{Node}_{\mathsf{GM}}(n, k, n_p)$ expresses the relation linking every node n to its parent n_p in the tree, and also records the unblinded key k associated with n. We will maintain the invariant that this structure is indeed a tree.

The group managers maintains an instance $\mathsf{Member}_{\mathsf{GM}}(n, d, A)$ for each leaf n in the tree. It records the member A associated with it. For convenience, we also keep track of its depth d in the tree (where the depth of the root is z). We will not store members at the root of the tree (i.e., the state will never contain a predicate instance of the form $\mathsf{Member}_{\mathsf{GM}}(n, \mathsf{z}, A)$ for any n and A).

Group Members

Each member A keeps track of the current group key, of its own depth in the tree, and for each node n from its own position in the tree to the root, it maintains the unblinded key k associated with that node and the blinded key k' of its (left or right) sibling — it has no information about any other node in the tree. This is achieved through the following memory predicates:

$\mathsf{GroupKey}_A(k)$	*The current group key is k*
$\mathsf{Depth}_A(d)$	*A lives at depth d*
$\mathsf{TreeView}_A(d, k, k')$	*The node at depth d on a path to the root has unblinded key k and its sibling has blinded key k'*

Observe that members do not have access to node identifiers. Instead, they refer to nodes on a path to the root through their depth.

Just as for GM's view, no group member lives at depth z.

9.3.4 Joining the Group

When a new member joins the group, the group manager expands the tree by splitting a current leaf node: a new inner node is added whose children host the joining member and the displaced member. Both are given a new (unblinded) key and, each key on the path to the root is updated. Then, the group manager sends a unicast message to the new member informing her of the unblinded keys of all the nodes to the root and of the blinded key of their siblings, so that she can create her TreeView. It also send a multicast message to the entire group to inform existing members of changes to the keys on their path to the root. These members will then update relevant portions of their TreeView.

Although intuitively simple, the overall join protocol is fairly complicate as different principals perform distinct actions, it manipulates recursive data structures (trees and tree views), and it must handle special cases. We will first describe the actions of the group manager, then of the joining principal, then the common actions of all existing members, then the actions specific to the displaced member, and finally specific actions of the other existing members.

Group Manager

For simplicity, we assume that the group manager GM starts the joining protocol upon receiving the message $\mathsf{N}(\mathsf{join}\ A)$. In actuality, there will be some kind of vetting mechanism that involves authentication and authorization of A's request. We first concentrate on the case where the tree is not empty. GM needs to perform the following actions:

- Select a member B to displace from leaf node n;
- create new nodes n_A and n_B for A and B respectively;

- update the tree so that n is the parent of n_A and n_B;
- create new unblinded keys k_A and k_B for A and B respectively and record them;
- update the keys associated with every node on a path from n to the root of the tree.

We will model the last item below with dedicated rewrite rules. In order to do so, GM relies on another memory predicate:

$\mathsf{UpdateTree}_{\mathsf{GM}}(n, A, d, M_A, M_m)$	*Update the path from node n at depth d to the root, sending welcome message M_A to joining member A and update message M_m to existing members.*

The above sequence of actions is captured by the following anchored rule. Notice that it picks the displaced member B at random and prepares for updating the path from the n to the root.

$$\left(\begin{array}{llcll}
 & & & & \mathsf{Node}_{\mathsf{GM}}(n_A, k_A, n) \\
 & & & & \mathsf{Member}_{\mathsf{GM}}(n_A, \mathsf{s}(d), A) \\
\forall A, B : \mathsf{principal}. & & & \exists k_A : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n_B, k_B, n) \\
\forall d : \mathsf{nat}. & \mathsf{N}(\mathsf{join}\ A) & & \exists k_B : \mathsf{uKey}. & \mathsf{Member}_{\mathsf{GM}}(n_B, \mathsf{s}(d), B) \\
\forall n, n_p : \mathsf{node}. & \mathsf{Member}_{\mathsf{GM}}(n, d, B) & \to & \exists n_A : \mathsf{node}. & \mathsf{Node}_{\mathsf{GM}}(n, \mathsf{f}(\mathsf{g}(k_A), \mathsf{g}(k_B)), n_p) \\
\forall k : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) & & \exists n_B : \mathsf{node}. & \mathsf{UpdateTree}_{\mathsf{GM}}(n, A, d, \\
 & & & & \qquad\qquad (\mathsf{s}(d),\ k_A,\ \mathsf{g}(k_B)), \\
 & & & & \qquad\qquad \{\!|\mathsf{newkey}, k_B,\ \mathsf{g}(k_A)|\!\}_k)
\end{array}\right)^{\mathsf{GM}}$$

Observe that the key associated to node n is replaced with the unblinded key $\mathsf{f}(\mathsf{g}(k_A), \mathsf{g}(k_B))$ obtained from the new keys k_A and k_B associated with n_A and n_B respectively. In the $\mathsf{UpdateTree}_{\mathsf{GM}}$ predicate, the welcome message for A is initialized to the depth $\mathsf{s}(d)$ of her node, her new (unblinded) key k_A and the blinded key $\mathsf{g}(k_B)$ of her sibling. The update message to the other members of the group is initialized to just B's new key k_B together with A's blinded key $\mathsf{g}(k_A)$ and tag newkey, all encrypted with the key k that was originally associated with n, the node on which B was sitting (so that only B can decrypt this message).

Before we examine how the rest of the welcome and update messages are constructed, let us settle the special case where A is the first member of the group, and is therefore starting from an empty tree. In this case, the group manager simply create a new node n_A and key k_A for A and send her a welcome message which includes her key and an arbitrary value (here $\star$) for the blinded key of the nonexistent sibling node of n_A.

$$\left(\begin{array}{llcll}
\forall A : \mathsf{principal}. & \mathsf{N}(\mathsf{join}\ A) & & \exists k_A : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n_A, k_A, n) \\
\forall n : \mathsf{node}. & \mathsf{Root}_{\mathsf{GM}}(n) & \to & \exists n_A : \mathsf{node}. & \mathsf{Member}_{\mathsf{GM}}(n_A, \mathsf{s}(\mathsf{z}), A) \\
\forall k_{\mathsf{GM},A} : \mathsf{shK}\ \mathsf{GM}\ A. & & & & \mathsf{N}(\{\mathsf{welcome},\ k_A,\ \star\}_{k_{\mathsf{GM},A}})
\end{array}\right)^{\mathsf{GM}}$$

The memory predicate UpdateTree is used by GM for the purpose of updating the keys of the nodes on a path from the displaced member to the root of the tree, and once the root has been reached to send the welcome and update messages. The following two rules travel this path towards the root.

The first rule is concerned with an inner node n at depth $\mathsf{s}(d)$ on this path. It recomputes the keys of the parent node n_p on the basis of the key k of n (recomputed at the previous iteration) and the key k_s of its sibling n_s. The welcome message to the joining member A is extended with the blinded key $\mathsf{g}(k_s)$ of n_s, while the update message to existing members is extended with the blinded key $\mathsf{g}(k)$ of n, tagged and encrypted with k_s (remember that n is the sibling of n_s). This makes n's new blinded key available to all the members in n_s's subtree.

$$\left(\begin{array}{llcl}
\forall A : \mathsf{principal}. & & & \mathsf{UpdateTree}_{\mathsf{GM}}(n_p, A, d, \\
\forall k_{\mathsf{GM},A} : \mathsf{shK}\ \mathsf{GM}\ A. & \mathsf{UpdateTree}_{\mathsf{GM}}(n, A, \mathsf{s}(d), M_A, M_m) & & \qquad\qquad (M_A,\ \mathsf{g}(k_s)), \\
\forall d : \mathsf{nat}. & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) & & \qquad\qquad (\{\!|\mathsf{rekey}, \mathsf{g}(k)|\!\}_{k_s},\ M_m)) \\
\forall n, n_s, n_p, n_{pp} : \mathsf{node}. & \mathsf{Node}_{\mathsf{GM}}(n_p, k_p, n_{pp}) & \to & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\
\forall k, k_p, k_s : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) & & \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) \\
\forall M_A, M_m : \mathsf{msg}. & & & \mathsf{Node}_{\mathsf{GM}}(n_p, \mathsf{f}(\mathsf{g}(k), \mathsf{g}(k_s)), n_{pp})
\end{array}\right)^{\mathsf{GM}}$$

When the root (the node n at depth z) has been reached, GM sends the welcome message M_A to A by encrypting it and an acknowledgment tag with the key $k_{\mathsf{GM},A}$ shared between them. It also broadcasts the updated message M_m to all existing members

$$\begin{pmatrix} \begin{array}{l} \forall A : \mathsf{principal}. \\ \forall k_{\mathsf{GM},A} : \mathsf{shK\ GM}\ A. \\ \forall n : \mathsf{node}. \\ \forall M_A, M_m : \mathsf{msg}. \end{array} & \mathsf{UpdateTree}_{\mathsf{GM}}(n, A, \mathsf{z}, M_A, M_m) & \rightarrow & \begin{array}{l} \mathsf{N}(\{\mathsf{welcome},\ M_A\}_{k_{\mathsf{GM},A}}) \\ !\mathsf{N}(M_m) \end{array} \end{pmatrix}^{\mathsf{GM}}$$

Recall that the memory predicate $\mathsf{UpdateTree}(n, A, d, M_A, M_m)$ obeys the invariant that d is the depth of node n.

If the node n_A where A is installed is at depth d, GM sends her a single unicast message $\{\mathsf{welcome},\ M_A\}_{k_{\mathsf{GM},A}}$ of size proportional to d, and it sends one broadcast message rekey, M_m, also of size proportional to d.

Joining Member

The actions of the joining member A consist in sending a join request, receiving the welcome message, and processing it. This is realized by the following two-rule role (the memory predicate $\mathsf{ReverseJoin}_A$ is discussed below):

$$\begin{pmatrix} \exists L : \mathsf{principal} \times \mathsf{tag}. & & & \\ & \cdot & \rightarrow & \begin{array}{l} \mathsf{N}(\mathsf{join}\ A) \\ L(A) \end{array} \\ \begin{array}{l} \forall d : \mathsf{nat}. \\ \forall M : \mathsf{msg}. \end{array} & \begin{array}{l} \mathsf{N}(\{\mathsf{welcome},\ M_A\}_{k_{\mathsf{GM},A}}) \\ L(A) \end{array} & \rightarrow & \mathsf{ReverseJoin}_A(M_A) \end{pmatrix}^{\forall A}$$

As mentioned earlier, we do not model the vetting of A's request. Consequently, we do not consider a denial subprotocol.

Upon receiving the welcome message M_A, the joining member A relies on the following two memory predicates to process it.

$\mathsf{ReverseJoin}_A(M)$	*Reassociate message M*
$\mathsf{CreatePath}_A(d, M)$	*Create A's tree view from depth d with message M*

The nesting of the concatenation operators in the welcome message M_A is the opposite of what A needs. Therefore she first reverses it using a subprocedure driven by ReverseJoin.

$$\big(\forall X, Y, Z : \mathsf{msg}. \quad \mathsf{ReverseJoin}_A((X,\ Y),\ Z) \quad \rightarrow \quad \mathsf{ReverseJoin}_A(X,\ (Y,\ Z)))\big)^{\forall A}$$

Once the welcome message has been fully reassociated, its leftmost component has the form $(\mathsf{s}(d),\ k_A,\ k'_s)$ where $\mathsf{s}(d)$ is the depth of A's new node, k_A is its unblinded key, and k'_s is the blinded key of its sibling. She can therefore create a new TreeView element for her node and set her depth. She also defines the memory predicate $\mathsf{createPath}_A(d, M)$ which will allow creating TreeView elements for the rest of the path to the root, where d is the depth of the next node to process.

$$\begin{pmatrix} \begin{array}{l} \forall d : \mathsf{nat}. \\ \forall k_A : \mathsf{uKey}. \\ \forall k'_s : \mathsf{bKey}. \end{array} & \mathsf{ReverseJoin}_A((\mathsf{s}(d),\ k_A,\ k'_s),\ M) & \rightarrow & \begin{array}{l} \mathsf{TreeView}_A(\mathsf{s}(d), k_A, k'_s) \\ \mathsf{Depth}_A(\mathsf{s}(d)) \\ \mathsf{CreatePath}_A(d, M) \end{array} \end{pmatrix}^{\forall A}$$

While processing the memory predicate $\mathsf{CreatePath}_A(d, M)$, the joining principal A maintains the invariant that she has memorized the unblinded keys at depths below d in TreeView predicates. The blinded keys of the siblings of the remaining nodes to the root will be found in M.

The first rule for CreatePath processes an inner node n, whose depth is therefore $\mathsf{s}(d)$ for some d. It has just stored the key k of n's child on a path to A in a TreeView predicate, and it finds the blinded key k'_s of n's sibling at the head of what is left of the reversed welcome message. It then simply creates a TreeView for n on the basis of k'_s and k and calls itself recursively on the remainder of the message.

$$\begin{pmatrix} \forall d : \mathsf{nat}. & & & \\ \forall k_c : \mathsf{uKey}. & \mathsf{CreatePath}_A(\mathsf{s}(d), (k'_s,\ M)) & \to & \mathsf{CreatePath}_A(d, M) \\ \forall k'_s, k'_c : \mathsf{bKey}. & \mathsf{TreeView}_A(\mathsf{s}(\mathsf{s}(d)), k_c, k'_c) & & \mathsf{TreeView}_A(\mathsf{s}(d), \mathsf{f}(\mathsf{g}(k_c), k'_c), k'_s) \\ \forall M : \mathsf{msg}. & & & \mathsf{TreeView}_A(\mathsf{s}(\mathsf{s}(d)), k_c, k'_c) \end{pmatrix}^{\forall A}$$

The recursion ends upon reaching the root: the depth in CreatePath is z and there is nothing left of the welcome message. Then, A uses the same procedure to construct the group key.

$$\begin{pmatrix} \forall k_c : \mathsf{uKey}. & \mathsf{CreatePath}_A(\mathsf{z}, \cdot) & \to & \mathsf{GroupKey}_A(\mathsf{f}(\mathsf{g}(k_c), k'_c)) \\ \forall k'_c : \mathsf{bKey}. & \mathsf{TreeView}_A(\mathsf{s}(\mathsf{z}), k_c, k'_c) & & \mathsf{TreeView}_A(\mathsf{s}(\mathsf{z}), k_c, k'_c) \end{pmatrix}^{\forall A}$$

Altogether, A has sent one message (the join request) and received one unicast message from GM of length proportional to d, where d is her depth in the tree. Processing this response involves $2d$ rule applications.

Existing Members — part I

The existing members of the group process the update message M_m broadcast by the group manager by copying it in a private UpdateTreeView memory predicate. This is achieved by the following rule:

$$\begin{pmatrix} \forall M_m : \mathsf{msg}. & !\mathsf{N}(M_m) & \to & \mathsf{UpdateTreeView}_C(M_m) \\ & & & !\mathsf{N}(M_m) \end{pmatrix}^{\forall C}$$

Observe that we implement multicast by reading the message $!\mathsf{N}(M_m)$ and then retransmitting it. As we said, each existing member C makes use of the auxiliary memory predicate

$\mathsf{UpdateTreeView}_C(M)$ *Update the tree view on the basis of message M*

Upon saving the update message, the displaced member B and other existing members C process this message differently.

Just like the welcome message for the joining member, the update message is associated in the wrong way. The following rule reassociates it.

$$\big(\forall X, Y, Z : \mathsf{msg}. \quad \mathsf{UpdateTreeView}_C((X,\ Y),\ Z) \quad \to \quad \mathsf{UpdateTreeView}_C(X,\ (Y,\ Z)))\big)^{\forall C}$$

Displaced Member

The group member B that has been displaced will find, as the first component of the update message, a term $\{\!|\mathsf{newkey}, k_{new}, k'_{new}|\!\}_k$ encrypted with the key k of the node at the current depth d of B. This informs him that k_{new} is his new key (at depth $\mathsf{s}(d)$) and k'_{new} is the blinded key of its sibling (the joining member). On the basis of this information, he will create a new TreeView for his new deeper node, and trigger a recalculation of the key of all the nodes on his path to the root. This is done by defining the memory predicate $\mathsf{Rekey}_B(d)$, whose processing rules are given below under "Existing Members — part II". The rest of the update message is discarded since B has all the information he needs to update the key chain.

$$\begin{pmatrix} \forall d : \mathsf{nat}. & \mathsf{UpdateTreeView}_B(\{\!|\mathsf{newkey}, k_{new},\ k'_{new}|\!\}_k, M) & & \mathsf{TreeView}_B(\mathsf{s}(d), k_{new}, k'_{new}) \\ \forall k : \mathsf{uKey}. & \mathsf{TreeView}_B(d, k, k'_s) & \to & \mathsf{TreeView}_B(d, k, k'_s) \\ \forall k'\!:\!\mathsf{bKey}. & \mathsf{Depth}_B(d) & & \mathsf{Depth}_B(\mathsf{s}(d)) \\ \forall M : \mathsf{msg}. & & & \mathsf{Rekey}_B(d) \end{pmatrix}^{\forall B}$$

Altogether, B receives one (multicast) message of length proportional to the depth of his node in the tree. The key update performed through Rekey has cost proportional to d.

Other Existing Members

For all other existing group members, the update message will contain an updated blinded key for the siblings of an initial segment of their path to the root. In order to act on it, they first need to discard prefixes that are irrelevant to them. This is achieved by the following simple rule:

$$\big(\forall X, M : \mathsf{msg}.\quad \mathsf{UpdateTreeView}_C(X,\ M) \quad\rightarrow\quad \mathsf{UpdateTreeView}_C(M)\big)^{\forall C}$$

When such a member C sees the first $\mathsf{UpdateTreeView}$ message whose contents has the form $(\{\!|k'_s|\!\}_k,\ X)$ for a known key k (i.e., one that is found in a $\mathsf{TreeView}_C$ predicate), he knows that it contains a new blinded key k'_s for the sibling of the corresponding node. Then, C updates this $\mathsf{TreeView}$ element and prepares for a rekey at depth d. The rest of the update message, M, is discarded.

$$\begin{pmatrix} \forall d : \mathsf{nat}. & & & \\ \forall k : \mathsf{uKey}. & \mathsf{UpdateTreeView}_C(\{\!|\mathsf{rekey}, k'_s,|\!\}_k\ M) & \rightarrow & \mathsf{TreeView}_C(\mathsf{s}(d), k, k'_s) \\ \forall k'_s, k'_{old} : \mathsf{bKey}. & \mathsf{TreeView}_C(\mathsf{s}(d), k, k'_{old}) & & \mathsf{Rekey}_C(d) \\ \forall M : \mathsf{msg}. & & & \end{pmatrix}^{\forall C}$$

Note that, as written, a member may discard relevant message fragments using the rule shown under "Existing Members — part I" before applying this rule. This would produce an incorrect tree view. For the sake of simplicity, we leave this point of non-determinism open in our specification, as a correct description would be significantly more complex.

Just like the displaced principal, other existing members receive one multicast message of length proportional to the depth d of the insertion, and need to do an amount of work that is proportional to d. However, this work is variously split between discarding message components and rekeying.

Existing Members — part II

Rekeying is performed by all existing group members. It is triggered by inserting a memory predicate of the form

$$\mathsf{Rekey}_C(d) \quad \textit{Rekey the path to the root from depth } d$$

in the state. It is implemented by the next two rules. The outcome will be to update all $\mathsf{TreeView}_C$ elements at depth d and smaller with the value of their current key. The new group key is obtained upon reaching the root of the tree.

Recall that no $\mathsf{TreeView}$ is defined for depth z. The next rule operate under the invariant that, when processing $\mathsf{Rekey}_C(\mathsf{s}(d))$, all $\mathsf{TreeView}$ elements of C are accurate at depths greater than $\mathsf{s}(d)$ but not at depths $\mathsf{s}(d)$ and lower. In particular, $\mathsf{TreeView}_C$ exists and is accurate for the node n' at depth $\mathsf{s}(\mathsf{s}(d))$, but not for the node n at depth $\mathsf{s}(d)$. The rule recomputes the key of n by combining the correct key k of n' with the (also correct) blinded key k' of n's sibling — this new key has value $\mathsf{f}(\mathsf{g}(k), k')$. It then calls itself recursively on depth d.

$$\begin{pmatrix} \forall d : \mathsf{nat}. & \mathsf{Rekey}_C(\mathsf{s}(d)) & & \mathsf{Rekey}_C(d) \\ \forall k, k_{old} : \mathsf{uKey}. & \mathsf{TreeView}_C(\mathsf{s}(\mathsf{s}(d)), k, k') & \rightarrow & \mathsf{TreeView}_C(\mathsf{s}(\mathsf{s}(d)), k, k') \\ \forall k', k'_p : \mathsf{bKey}. & \mathsf{TreeView}_C(\mathsf{s}(d), k_{old}, k'_p) & & \mathsf{TreeView}_C(\mathsf{s}(d), \mathsf{f}(\mathsf{g}(k), k'), k'_p) \end{pmatrix}^{\forall C}$$

Once the root is reached, the group key is updated on the basis of the keys of the element just below it.

$$\begin{pmatrix} \forall d : \mathsf{nat}. & \mathsf{Rekey}_C(\mathsf{z}) & & \\ \forall k, k_{old} : \mathsf{uKey}. & \mathsf{GroupKey}_C(k_{old}) & \rightarrow & \mathsf{GroupKey}_C(\mathsf{f}(\mathsf{g}(k), k')) \\ \forall k' : \mathsf{bKey}. & \mathsf{TreeView}_C(\mathsf{s}(\mathsf{z}), k, k') & & \mathsf{TreeView}_C(\mathsf{s}(\mathsf{z}), k, k') \end{pmatrix}^{\forall C}$$

9.3.5 Eviction

When evicting a group member member A, the corresponding leaf node n_A is removed and the tree is compacted so that its sibling n_s becomes its parent. It is also necessary to change the blinded key of n_s so that A has no usable information about the tree she has been evicted from. To do so, the group manager gives a new unblinded key to some arbitrary member B in the subtree starting at n_s. It then informs all members in this subtree that they must update their view of the tree (since their path to the root has been shortened by one node) and perform the rekeying operation. The other group members need to just do the rekeying.

Group Manager

From the group manager's perspective, the eviction protocol resembles the joining protocol, but is slightly more involved as the tree below the evicted member's sibling must be handled differently from the other remaining members. Altogether, the operation will make use of the following private memory predicates:

$\mathsf{Evict}_{\mathsf{GM}}(A)$	*Evict member A*
$\mathsf{PickMember}_{\mathsf{GM}}(n, \bar{d}, d)$	*Pick a member to rekey underneath node n*
$\mathsf{UpdateTree2}_{\mathsf{GM}}(n, \bar{d}, d, M)$	*Update the path from node n at depth d to the root sending update message M to members*
$\mathsf{UpdateTree3}_{\mathsf{GM}}(n, M)$	*Update the path from node n to the root sending update message M to members*
$\mathsf{ShiftUp}_{\mathsf{GM}}(n)$	*Decrement the depth of leaves under n by one*

Assume that the decision of evicting member A has resulted in the addition of the memory state predicate $\mathsf{Evict}_{\mathsf{GM}}(A)$. Carrying out this decision consists of several operations: sending an eviction message to A, removing her node and compacting the tree, updating the keys in the tree, and notifying the remaining members through multicast. The following rule immediately does the first three operations, and prepares for the remaining two (which require visiting the tree). The memory predicate $\mathsf{PickMember}_{\mathsf{GM}}(n_s, \mathsf{s}(d), \mathsf{s}(\mathsf{z}))$ prepares to pick a member under the former sibling node n_s. The value $\mathsf{s}(\mathsf{z})$ will be used to change mode of operation on the way back, while $\mathsf{s}(d)$ serves the purpose of informing members in this subtree of which node to eliminate in their view.

$$\left(\begin{array}{lcl} \begin{array}{l} \forall A : \mathsf{principal}. \\ \forall k_{\mathsf{GM},A} : \mathsf{shK\ GM}\ A. \\ \forall n, n_s, n_p, n_{ps}, n_{pp} : \mathsf{node}. \\ \forall k, k_s, k_p, k_{ps} : \mathsf{uKey}. \end{array} & \begin{array}{l} \mathsf{Evict}_{\mathsf{GM}}(A) \\ \mathsf{Member}_{\mathsf{GM}}(n, \mathsf{s}(d), A) \\ \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\ \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) \\ \mathsf{Node}_{\mathsf{GM}}(n_p, k_p, n_{pp}) \end{array} \rightarrow & \begin{array}{l} \mathsf{N}(\{\mathsf{evict}\}_{k_{\mathsf{GM},A}}) \\ \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_{pp}) \\ \mathsf{PickMember}_{\mathsf{GM}}(n_s, \mathsf{s}(d), \mathsf{s}(\mathsf{z})) \end{array} \end{array}\right)^{\mathsf{GM}}$$

GM follows an arbitrary path to a leaf starting at the sibling of the evicted node. Whenever it follows a branch, it uses the memory predicate ShiftUp to also start a parallel procedure to shift every depth value in the subtree that was not followed.

$$\left(\begin{array}{lcl} \begin{array}{l} \forall n, n_s, n_p : \mathsf{node}. \\ \forall k, k_s : \mathsf{uKey}. \\ \forall d, \bar{d} : \mathsf{nat}. \\ \forall M : \mathsf{msg}. \end{array} & \begin{array}{l} \mathsf{PickMember}_{\mathsf{GM}}(n_p, \bar{d}, d') \\ \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\ \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) \end{array} \rightarrow & \begin{array}{l} \mathsf{PickMember}_{\mathsf{GM}}(n, \bar{d}, \mathsf{s}(d')) \\ \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\ \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) \\ \mathsf{ShiftUp}_{\mathsf{GM}}(n_s) \end{array} \end{array}\right)^{\mathsf{GM}}$$

When a leaf n corresponding to member B is reached, GM creates a new key k, updates B's information, notifies him of the change, and prepares for a rekey using procedure UpdateTree2.

$$\left(\begin{array}{lcl} \begin{array}{l} \forall B : \mathsf{principal}. \\ \forall k_{\mathsf{GM},B} : \mathsf{shK\ GM}\ B. \\ \forall d, \bar{d} : \mathsf{nat}. \\ \forall n, n_p : \mathsf{node}. \\ \forall k_{old} : \mathsf{uKey}. \\ \forall M : \mathsf{msg}. \end{array} & \begin{array}{l} \mathsf{PickMember}_{\mathsf{GM}}(n, \bar{d}, \mathsf{s}(d')) \\ \mathsf{Member}_{\mathsf{GM}}(n, \mathsf{s}(d), B) \\ \mathsf{Node}_{\mathsf{GM}}(n, k_{old}, n_p) \end{array} \rightarrow\ \exists k : \mathsf{uKey}. & \begin{array}{l} \mathsf{Member}_{\mathsf{GM}}(n, d, B) \\ \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\ \mathsf{UpdateTree2}_{\mathsf{GM}}(n_p, d', \bar{d}, \\ \qquad\qquad \{\mathsf{newkey}', \bar{d}, k\}_{k_{\mathsf{GM},B}}) \end{array} \end{array}\right)^{\mathsf{GM}}$$

The procedure UpdateTree2 is similar to UpdateTree in the joining protocol, except that no joining member nor welcome message are involved. It grows the update message to the remaining members with information so that the nodes under this tree can shorten the view of their path to the root of the tree and perform a rekey.

$$\begin{pmatrix} \forall d, \bar{d} : \mathsf{nat}. & \mathsf{UpdateTree2}_{\mathsf{GM}}(n, \bar{d}, \mathsf{s}(d'), M) & & \mathsf{UpdateTree2}_{\mathsf{GM}}(n_p, \bar{d}, d', (M, \{\!|\mathsf{shift}, \bar{d}, \mathsf{g}(k)|\!\}_{k_s})) \\ \forall n, n_s, n_p, n_{pp} : \mathsf{node}. & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) & \to & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\ \forall k, k_p, k_s : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n_p, k_p, n_{pp}) & & \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) \\ \forall M : \mathsf{msg}. & \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) & & \mathsf{Node}_{\mathsf{GM}}(n_p, \mathsf{f}(\mathsf{g}(k), \mathsf{g}(k_s)), n_{pp}) \end{pmatrix}^{\mathsf{GM}}$$

When the (parent of the) evicted node is reached, it shifts to procedure $\mathsf{UpdateTree3}_{\mathsf{GM}}$ which behave similarly, except that the update message will be extended only with a rekey instruction.

$$\begin{pmatrix} \forall n : \mathsf{node}. & & & \\ \forall \bar{d} : \mathsf{nat}. & \mathsf{UpdateTree2}_{\mathsf{GM}}(n, \mathsf{z}, \bar{d}, M) & \to & \mathsf{UpdateTree3}_{\mathsf{GM}}(n, M) \\ \forall M : \mathsf{msg}. & & & \end{pmatrix}^{\mathsf{GM}}$$

$$\begin{pmatrix} \forall d : \mathsf{nat}. & \mathsf{UpdateTree3}_{\mathsf{GM}}(n, M) & & \mathsf{UpdateTree3}_{\mathsf{GM}}(n_p, (M, \{\!|\mathsf{rekey}, \mathsf{g}(k)|\!\}_{k_s})) \\ \forall n, n_s, n_p, n_{pp} : \mathsf{node}. & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) & \to & \mathsf{Node}_{\mathsf{GM}}(n, k, n_p) \\ \forall k, k_p, k_s : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n_p, k_p, n_{pp}) & & \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) \\ \forall M : \mathsf{msg}. & \mathsf{Node}_{\mathsf{GM}}(n_s, k_s, n_p) & & \mathsf{Node}_{\mathsf{GM}}(n_p, \mathsf{f}(\mathsf{g}(k), \mathsf{g}(k_s)), n_{pp}) \end{pmatrix}^{\mathsf{GM}}$$

When reaching the root, GM broadcasts the update message.

$$\begin{pmatrix} \forall n : \mathsf{node}. & \mathsf{UpdateTree3}_{\mathsf{GM}}(n, M) & \to & !\mathsf{N}(M) \\ \forall M : \mathsf{msg}. & \mathsf{Root}_{\mathsf{GM}}(n) & & \mathsf{Root}_{\mathsf{GM}}(n) \end{pmatrix}^{\mathsf{GM}}$$

The procedure triggered with memory predicate $\mathsf{ShiftUp}_{\mathsf{GM}}$ simply traverses a subtree all the way down to its leaves, where it decrements the depth of every member by one.

$$\begin{pmatrix} & & & \mathsf{ShiftUp}_{\mathsf{GM}}(n_l) \\ \forall n, n_l, n_r : \mathsf{node}. & \mathsf{ShiftUp}_{\mathsf{GM}}(n) & & \mathsf{ShiftUp}_{\mathsf{GM}}(n_r) \\ \forall k_l, k_r : \mathsf{uKey}. & \mathsf{Node}_{\mathsf{GM}}(n_l, k_l, n) & \to & \mathsf{Node}_{\mathsf{GM}}(n_l, k_l, n) \\ & \mathsf{Node}_{\mathsf{GM}}(n_r, k_r, n) & & \mathsf{Node}_{\mathsf{GM}}(n_r, k_r, n) \end{pmatrix}^{\mathsf{GM}}$$

$$\begin{pmatrix} \forall n : \mathsf{node}. & \mathsf{ShiftUp}_{\mathsf{GM}}(n) & & \\ \forall d : \mathsf{nat}. & & \to & \mathsf{Member}_{\mathsf{GM}}(n, d, A) \\ \forall A : \mathsf{principal}. & \mathsf{Member}_{\mathsf{GM}}(n, \mathsf{s}(d), A) & & \end{pmatrix}^{\mathsf{GM}}$$

Evicted Member

Upon receiving an eviction message, the evicted member A removes her depth predicate and sets the memory predicate $\mathsf{DestroyPath}_A(d)$ which she will use to dispose of all the path elements associated with this group.

$$\begin{pmatrix} \forall k_{\mathsf{GM},A} : \mathsf{shK}\ \mathsf{GM}\ A. & \mathsf{N}(\{\mathsf{evict}\}_{k_{\mathsf{GM},A}}) & \to & \mathsf{DestroyPath}_A(d) \\ \forall d : \mathsf{nat}. & \mathsf{Depth}_A(d) & & \end{pmatrix}^{\forall A}$$

The auxiliary predicate is read as follows.

$\mathsf{DestroyPath}_A(d)$ *Dispose of data structures at depth d and below*

A gets rid of her TreeView elements one at a time, and eliminates the group key when she reaches the root.

$$\begin{pmatrix} \forall d : \mathsf{nat}. & \mathsf{DestroyPath}_A(\mathsf{s}(d)) & & \\ \forall k : \mathsf{uKey}. & & \to & \mathsf{DestroyPath}_A(d) \\ \forall k' : \mathsf{bKey}. & \mathsf{TreeView}_A(\mathsf{s}(d), k, k') & & \end{pmatrix}^{\forall A}$$

$$\begin{pmatrix} \forall k : \mathsf{uKey}. & \mathsf{DestroyPath}_A(\mathsf{z}) & \to & \cdot \\ & \mathsf{GroupKey}_A(k) & & \end{pmatrix}^{\forall A}$$

From A's perspective, eviction involved receiving one message and applying d rules where d was her depth in the tree.

All Members — part I

Just like in the joining protocol, existing members receive the multicast update message in their $\mathsf{UpdateTreeView}$ memory predicate and reassociate it.

Rekeyed Member

The rekeyed member B replaces his key k_{old} with the new key k from the GM. It invokes the procedure $\mathsf{Rekey'}_C(d,\bar{d})$ to rekey the segment of its path to the root from depth d to the depth $\bar{d}$ of the evicted node.

$$\left(\begin{array}{lll} \forall k_{\mathsf{GM},B} : \mathsf{shK\ GM}\ B. & & \\ \forall d,\bar{d} : \mathsf{nat}. & \mathsf{UpdateTreeView}_B(\{\mathsf{newkey'},\bar{d},k\}_{k_{\mathsf{GM},B}},M) & \to & \mathsf{Rekey'}_B(d,\bar{d}) \\ \forall k,k_{old} : \mathsf{uKey}. & \mathsf{TreeView}_B(\mathsf{s}(\mathsf{s}(d)),k_{old},k') & & \mathsf{TreeView}_B(\mathsf{s}(\mathsf{s}(d)),k,k') \\ \forall k' : \mathsf{bKey}. & & \end{array}\right)^{\forall B}$$

Members in Upshifted Subtree

The members in the subtree that starts with the sibling of the evicted node (the *upshifted tree*) need to carry out two operations: rekey their path to the root and reduce their depth by one. This achieved using the following two memory predicates:

$\mathsf{Rekey'}_C(d,\bar{d})$	*From depth d, rekey and shift up depth $\bar{d}$ and then just rekey*
$\mathsf{ShiftUp}_C(d)$	*Decrement the depth of the path from d and below*

When asked to process the message of the form $\{\!|\mathsf{shift},\bar{d},k'|\!\}_k$ for the deepest key k in its tree view, member C replaces the old blinded key at that depth with k' and then triggers the procedures $\mathsf{Rekey'}$ to update the blinded keys of all the nodes to its path to the root.

$$\left(\begin{array}{llll} \forall d,\bar{d} : \mathsf{nat}. & \mathsf{UpdateTreeView}_C(\{\!|\mathsf{shift},\bar{d},k'|\!\}_k,M) & & \mathsf{TreeView}_C(d,k,k') \\ \forall k : \mathsf{uKey}. & & \to & \\ \forall k',k'_{old} : \mathsf{bKey}. & \mathsf{TreeView}_C(\mathsf{s}(d),k,k'_{old}) & & \mathsf{Rekey'}_C(d,\bar{d}) \end{array}\right)^{\forall C}$$

The auxiliary predicate $\mathsf{Rekey'}_C$ updates the keys of C's tree view upwards toward the root, without changing their depth. It stops at the depth where the evicted member lived.

$$\left(\begin{array}{llll} \forall d,\bar{d} : \mathsf{nat}. & \mathsf{Rekey'}_C(\mathsf{s}(d),\bar{d}) & & \mathsf{Rekey'}_C(d,\bar{d}) \\ \forall k,k_{old} : \mathsf{uKey}. & \mathsf{TreeView}_C(\mathsf{s}(\mathsf{s}(d)),k,k') & \to & \mathsf{TreeView}_C(\mathsf{s}(cs(d)),k,k') \\ \forall k',k'_p : \mathsf{bKey}. & \mathsf{TreeView}_C(\mathsf{s}(d),k_{old},k'_p) & & \mathsf{TreeView}_C(\mathsf{s}(d),\mathsf{f}(\mathsf{g}(k),k'),k'_p) \end{array}\right)^{\forall C}$$

There, it removes the $\mathsf{TreeView}$ element at that depth, invokes a regular rekeying of the nodes from there up using Rekey and triggers the increment of every depth below this element using the auxiliary predicate $\mathsf{ShiftUp}$.

$$\left(\begin{array}{llll} \forall d : \mathsf{nat}. & \mathsf{Rekey'}_C(\mathsf{s}(d),\mathsf{s}(d)) & & \mathsf{Rekey}_C(d) \\ \forall k,k_{old} : \mathsf{uKey}. & & \to & \\ \forall k' : \mathsf{bKey}. & \mathsf{TreeView}_C(\mathsf{s}(d),k,k') & & \mathsf{ShiftUp}_C(\mathsf{s}(d)) \end{array}\right)^{\forall C}$$

As it moves downwards, the predicate $\mathsf{ShiftUp}_C(d)$ decrements the depth of every $\mathsf{TreeView}$ element it finds, as well as the depth of member C when it reaches it.

$$\left(\begin{array}{llll} \forall d : \mathsf{nat}. & \mathsf{ShiftUp}_A(d) & & \mathsf{ShiftUp}_A(\mathsf{s}(d)) \\ \forall k : \mathsf{uKey}. & & \to & \\ \forall k' : \mathsf{bKey}. & \mathsf{TreeView}_A(\mathsf{s}(d),k,k') & & \mathsf{TreeView}_A(d,k,k') \end{array}\right)^{\forall A}$$

$$\left(\forall d : \mathsf{nat}. \quad \begin{array}{l} \mathsf{ShiftUp}_A(\mathsf{s}(d)) \\ \mathsf{Depth}_A(\mathsf{s}(d)) \end{array} \quad \to \quad \mathsf{Depth}_A(d)\right)^{\forall A}$$

All Members — part II

At this point, all members perform the rekey operation just as in the joining subprotocol. Members in the upshifted tree continue from where Rekey$'$ left off, while other members do it from scratch.

10 Conclusions

This report collects, rationalizes and completes a body of previously published work on the *MSR 2.0* cryptographic protocol specification language. In the past, we had published aspects of this research, but never in a single place and not always using a consistent syntax or the same set of features, which made it difficult for colleagues to get a full picture of *MSR 2.0*. In this regard, the present report is meant to act as a reference. It also contains several definitions and numerous proofs that were never published.

A variant of *MSR 2.0*, documented in [24], extends the scope of this language beyond cryptographic protocols. The main difference consist in the ability to declare symbols (rather than using hardwired constructs specialized to security applications), in the presence of guards (state predicates that must be present for a rule to fire but that are not consumed), and the absence of data access validation (a static check specialized to cryptographic protocol specification). This variant has been implemented using the Maude rewriting tool [39].

Acknowledgments

We are thankful to Cathy Meadows for the many discussion during the early stages of this work, and to the reviewers of the several papers that presented aspects of *MSR 2.0*.

References

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

[2] Martín Abadi and Philip Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proceedings of the 1st IFIP International Conference on Theoretical Computer Science*, pages 3–22. Springer-Verlage LNCS 1872, 2000.

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

[4] Michael Backes, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Cryptographically Sound Security Proofs for Basic and Public-Key Kerberos. *International Journal of Information Security — IJIS*, 10(2):107–134, 2011.

[5] Michael Backes, Brigitte Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *Proceedings of CCS'03*, pages 220–230, 2003.

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

[7] Stefano Bistarelli, Iliano Cervesato, Gabriele Lenzini, and Fabio Martinelli. Relating Process Algebras and Multiset Rewriting for Immediate Decryption Protocols. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Second International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security — MMM'03*, pages 86–99, St. Petersburg, Russia, 20–24 September 2003. Springer-Verlag LNAI 2776.

[8] Stefano Bistarelli, Iliano Cervesato, Gabriele Lenzini, and Fabio Martinelli. Relating Process Algebras and Multiset Rewriting for Security Protocol Analysis. In R. Gorrieri, editor, *Third Workshop on Issues in the Theory of Security — WITS'03*, pages 21–31, Warsaw, Poland, 5–6 April 2003.

[9] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006.

[10] Stephen 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.

[11] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. A Formal Analysis of Some Properties of Kerberos 5 Using MSR. In *Fifteenth Computer Security Foundations Workshop — CSFW-15*, pages 175–190, Cape Breton, NS, Canada, 24–26 June 2002. IEEE Computer Society Press.

[12] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. Verifying Confidentiality and Authentication in Kerberos 5. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security - Theories and Systems — ISSS 2003*, pages 1–24, Tokyo, Japan, 4–6 November 2003. Springer-Verlag LNCS 3233.

[13] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. A Formal Analysis of Some Properties of Kerberos 5 Using MSR. Technical Report MS-CIS-04-04, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, April 2004.

[14] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Formal Analysis of Kerberos 5. *Theoretical Computer Science*, 367(1-2):57–87, November 2006.

[15] Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange). In *Proceedings of the third Theory of Cryptography Conference (TCC'06)*, 2006.

[16] Iliano Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. In A. Seda, editor, *First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology — MFCSIT'00*, pages 1–43, Cork, Ireland, 19–21 July 2000. Elsevier ENTCS 40.

[17] Iliano Cervesato. A Specification Language for Crypto-Protocols 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*, pages 1–22, Paphos, Cyprus, 1 December 2001.

[18] Iliano Cervesato. The Dolev-Yao Intruder is the Most Powerful Attacker. In J. Halpern, editor, *16th Annual Symposium on Logic in Computer Science — LICS'01*, Boston, MA, 16–19 June 2001. IEEE Computer Society Press.

[19] Iliano Cervesato. Typed MSR: Syntax and Examples, First International Workshop on Mathematical Methods. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Models and Architectures for Computer Networks Security — MMM'01*, pages 159–177, St. Petersburg, Russia, 21–23 May 2001. Springer-Verlag LNCS 2052.

[20] Iliano Cervesato. Data Access Specification and the Most Powerful Symbolic Attacker in MSR. In M. Okada, B. Pierce, Andre Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security: Theories and Systems — ISSS 2002*, pages 384–416, Tokyo, Japan, 8–10 November 2002. Springer-Verlag LNCS 2609. Revised Papers of the 2002 Mext-NSF-JSPS International Symposium.

[21] Iliano Cervesato. The Wolf Within. In J. Guttman, editor, *Second Workshop on Issues in the Theory of Security — WITS'02*, Portland, OR, 14–15 January 2002.

[22] Iliano Cervesato. The Logical Meeting Point of Multiset Rewriting and Process Algebra: Progress Report. Technical Memo CHACS-5540-153, Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC, September 2004.

[23] Iliano Cervesato. Towards a Notion of Quantitative Security Analysis. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, editors, *Quality of Protection: Security Measurements and Metrics — QoP'05*, pages 131–144. Springer-Verlag Advances in Information Security 23, 2006.

[24] Iliano Cervesato. MSR 2.0: Language Definition and Programming Environment. Technical Report CMU-CS-11-141, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2011.

[25] Iliano Cervesato, Michael Backes, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Cryptographically Sound Security Proofs for Basic and Public-Key Kerberos. In D. Gollmann and A. Sabelfeld, editors, *11th European Symposium On Research In Computer Security — ESORICS'06*, pages 362–383, Hamburg, Germany, 18–20 September 2006. IEEE Computer Society Press.

[26] Iliano Cervesato, Stefano Bistarelli, Gabriele Lenzini, and Fabio Martinelli. Relating Multiset Rewriting and Process Algebras for Security Protocol Analysis. *Journal of Computer Security*, 13(1):3–47, February 2005.

[27] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. In *12th Computer Security Foundations Workshop — CSFW-12*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[28] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating Strands and Multiset Rewriting for Security Protocol Analysis. In *13th Computer Security Foundations Workshop — CSFW-13*, pages 35–51, Cambridge, UK, 3–5 July 2000. IEEE Computer Society Press.

[29] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Comparison between Strand Spaces and Multiset Rewriting for Security Protocol Analysis. In M. Okada, B. Pierce, Andre Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security: Theories and Systems — ISSS 2002*, pages 356–383, Tokyo, Japan, 8–10 November 2002. Springer-Verlag LNCS 2609. Revised Papers of the 2002 Mext-NSF-JSPS International Symposium.

[30] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Comparison between Strand Spaces and Multiset Rewriting for Security Protocol Analysis. *Journal of Computer Security*, 13(2):265–316, April 2005.

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

[32] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Specifying Kerberos 5 Cross-Realm Authentication. In Catherine Meadows and Jan Jrjens, editors, *Fifth Workshop on Issues in the Theory of Security — WITS'05*, pages 12–26, Long Beach, CA, 10–11 January 2005. ACM Digital Library.

[33] Iliano Cervesato, Aaron D. Jaggard, Joe-Kai Tsay, Andre Scedrov, and Christopher Walstad. Breaking and Fixing Public-Key Kerberos. In Mitsu Okada and Ichiro Satoh, editors, *Eleventh Annual Asian Computing Science Conference — ASIAN'06*, pages 167–181, Tokyo, Japan, 6–8 December 2006. Springer-Verlag LNCS 4435. Post-conference proceedings.

[34] Iliano Cervesato, Aaron D. Jaggard, Joe-Kai Tsay, Andre Scedrov, and Christopher Walstad. Breaking and Fixing Public-Key Kerberos. In Dieter Gollmann and Jan Jrjens, editors, *Sixth Workshop on Issues in the Theory of Security — WITS'06*, pages 55–70, Vienna, Austria, 25–26 March 2006.

[35] Iliano Cervesato, Aaron D. Jaggard, Joe-Kai Tsay, Andre Scedrov, and Christopher Walstad. Breaking and Fixing Public-Key Kerberos. *Information & Computation*, 206(2-4):402–424, April 2008.

[36] Iliano Cervesato and Andre Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. In Ruy de Queiroz, editor, *Thirteenth Workshop on Logic, Language, Information and Computation — WoLLIC'06*, pages 145–176, Stanford, CA, 18–21 July 2006. Elsevier ENTCS 165.

[37] Iliano Cervesato and Andre Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 207(10):1044–1077, October 2009.

[38] Iliano Cervesato and Mark-Oliver Stehr. Representing the MSR Cryptoprotocol Specification Language in an Extension of Rewriting Logic with Dependent Types. In Narciso Martí-Oliet, editor, *Fifth International Workshop on Rewriting Logic and its Applications — WRLA'04*, pages 183–207, Barcelona, Spain, 27–28 March 2004. Elsevier ENTCS 117.

[39] Iliano Cervesato and Mark-Oliver Stehr. Representing the MSR Cryptoprotocol Specification Language in an Extension of Rewriting Logic with Dependent Types. *Higher-Order and Symbolic Computation*, 20(1/2):3–35, June 2007.

[40] Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In *Proceedings of ESOP-14*, pages 157–171, 2005.

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

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

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

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

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

[46] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12:247–311, 2004.

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

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

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

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

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

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

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

[54] Laurence 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.

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

[56] Brigitte Pientka. An insider's look at LF type reconstruction:Everything you (n)ever wanted to know. Submitted for publication, 2010.

[57] Vitaly Shmatikov and Ulrich 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.

[58] Dawn 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.

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

A Collected Rules

This appendix collects the grammatical productions, judgments and rules used throughout the report for the ease of the reader. More specifically, the syntax of *MSR* is summarized in Section A.1, and the rules for type-checking, data access specification, and execution are given in Sections A.2, A.3 and A.4, respectively. For the convenience of the reader, we give the number of the page where each notion is first introduced.

A.1 Syntax

Atomic messages:	a	$::=$	A	*(Principal)*	[p. 2]
		$\mid$	k	*(Key)*	
		$\mid$	n	*(Nonce)*	
		$\mid$	m	*(Raw datum)*	
Parametric messages:	t	$::=$	a	*(Atomic messages)*	[p. 2]
		$\mid$	x	*(Variables)*	[p. 6]
		$\mid$	$t_1\ t_2$	*(Concatenation)*	
		$\mid$	$\{t\}_k$	*(Symmetric-key encryption)*	
		$\mid$	$\{\!\{t\}\!\}_k$	*(Asymmetric-key encryption)*	
Message tuples:	$\vec{t}$	$::=$	$\cdot$	*(Empty tuple)*	[p. 8]
		$\mid$	$t,\ \vec{t}$	*(Tuple extension)*	
States:	S	$::=$	$\cdot$	*(Empty state)*	[p. 11]
		$\mid$	$S,\ \mathsf{N}(t)$	*(Extension with a network predicate)*	
		$\mid$	$S,\ \mathsf{L}_l(\vec{t})$	*(Extension with a role state predicate)*	
		$\mid$	$S,\ \mathsf{M}_{\mathsf{A}}(\vec{t})$	*(Extension with a memory predicate)*	

Types:	τ	$::=$	$\mathsf{principal}$	*(Principals)*	[p. 3]
		$\mid$	nonce	*(Nonces)*	
		$\mid$	$\mathsf{shK}\ A\ B$	*(Shared keys)*	
		$\mid$	$\mathsf{pubK}\ A$	*(Public keys)*	
		$\mid$	$\mathsf{privK}\ k$	*(Private keys)*	
		$\mid$	msg	*(Messages)*	
Tuple types:	$\vec{\tau}$	$::=$	$\cdot$	*(Empty tuple)*	[p. 8]
		$\mid$	$\tau^{(x)} \times \vec{\tau}$	*(Tuple type extension)*	

Predicate sequences:	lhs	$::=$	$\cdot$	*(Empty predicate sequence)*	[p. 13]
		$\mid$	$lhs,\ \mathsf{N}(t)$	*(Extension with a network predicate)*	
		$\mid$	$lhs,\ L(\vec{e})$	*(Extension with a role state predicate)*	
		$\mid$	$lhs,\ \mathsf{M}_A(\vec{t})$	*(Extension with a memory predicate)*	
Right-Hand sides:	rhs	$::=$	lhs	*(Sequence of message predicates)*	[p. 13]
		$\mid$	$\exists x : \tau.\ rhs$	*(Fresh data generation)*	
Rule:	r	$::=$	$lhs \to rhs$	*(Rule core)*	[p. 13]
		$\mid$	$\forall x : \tau.\ r$	*(Parameter closure)*	
Rule collections:	ρ	$::=$	$\cdot$	*(Empty role)*	[p. 14]
		$\mid$	$\exists L : \vec{\tau}.\ \rho$	*(Role state predicate parameter declaration)*	
		$\mid$	$r,\ \rho$	*(Extension with a rule)*	
Protocol theories:	$\mathcal{P}$	$::=$	$\cdot$	*(Empty protocol theory)*	[p. 15]
		$\mid$	$\mathcal{P},\ \rho^{\forall A}$	*(Extension with a generic role)*	
		$\mid$	$\mathcal{P},\ \rho^{A}$	*(Extension with an anchored role)*	

Active role sets:	R	$::=$	$\cdot$	*(No active role)*	[p. 16]
		$\mid$	$R,\ \rho^{\mathsf{A}}$	*(Extension with an instantiated role)*	

Signatures:	Σ	$::=$	$\cdot$	*(Empty signature)*	[p. 4]
		$\mid$	$\Sigma,\ a : \tau$	*(Atomic message declaration)*	
		$\mid$	$\Sigma,\ \mathsf{L}_l : \vec{\tau}$	*(Local state predicate declaration)*	[p. 10]
		$\mid$	$\Sigma,\ \mathsf{M}\_ : \vec{\tau}$	*(Memory predicate declaration)*	[p. 10]
Typing contexts:	Γ	$::=$	Σ	*(Plain signature)*	[p. 6]
		$\mid$	$\Gamma, x : \tau$	*(Extension with a variable declaration)*	
		$\mid$	$\Gamma, L : \vec{\tau}$	*(Extension with a role state predicate declaration)*	[p. 12]
Knowledge contexts:	Δ	$::=$	$\cdot$	*(Empty knowledge context)*	[p. 20]
		$\mid$	Δ, a	*(Extension with atomic knowledge)*	
		$\mid$	Δ, x	*(Extension with parametric knowledge)*	
		$\mid$	Δ, t	*(Extension with ground terms)*	[p. 27]

Snapshot: $C \quad ::= \quad [S]_{\Sigma}^{R}$ [p. 36]

A.2 Typing Rules

$\tau :: \tau'$	*τ is a subsort of τ'*	[p. 3]
$\Sigma \vdash t : \tau$	*Term t has type τ in signature Σ*	[pp. 4, 6]
$\Sigma \vdash \tau$	*τ is a valid type in Σ*	[p. 5]
$\vdash \Sigma$	*Σ is a valid signatures*	[pp. 5, 11]
$\models^{c} \Gamma$	*Γ is a valid typing context*	[pp. 7 12]
$\Sigma \vdash \vec{t} : \vec{\tau}$	*Term tuple $\vec{t}$ has type $\vec{\tau}$ in signature Σ*	[p. 9]
$\Gamma \vdash \vec{\tau}$	*$\vec{\tau}$ is a valid tuple type in typing context Γ*	[p. 9]
$\Sigma \vdash P$	*P is a valid message predicate in signature Σ*	[p. 10]
$\Sigma \vdash S$	*S is a valid state in signature Σ*	[p. 11]
$\Gamma \models^{r} rhs$	*rhs is a valid rule consequent in typing context Γ*	[p. 13]
$\Gamma \vdash r$	*r is a valid rule in typing context Γ*	[p. 14]
$\Gamma \vdash \rho$	*ρ is a valid rule collection in typing context Γ*	[p. 15]
$\Sigma \vdash \mathcal{P}$	*$\mathcal{P}$ is a valid protocol theory in signature Σ*	[p. 15]
$\Sigma \vdash R$	*R is a valid active role set in signature Σ*	[p. 16]

$\tau :: \tau'$ *τ is a subsort of τ'* [p. 3]

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

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

$\Sigma \vdash t : \tau \qquad \Gamma \vdash t : \tau$ *Term t has type τ in signature Σ (viz. context Γ)* [pp. 4, 6]

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

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

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

A COLLECTED RULES

$\Sigma \vdash \tau \qquad \Gamma \vdash \tau$ *τ is a valid type in signature Σ (viz. context Γ)* [pp. 5, 6]

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

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

$\vdash \Sigma$ *Σ is a valid signatures* [pp. 5, 11]

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

$\models^{c} \Gamma$ *Γ is a valid typing context* [pp. 7, 12]

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

$\Sigma \vdash \vec{t} : \vec{\tau} \qquad \Gamma \vdash \vec{t} : \vec{\tau}$ *Term tuple $\vec{t}$ has type $\vec{\tau}$ in signature Σ (viz. context Γ)* [pp. 9, 6]

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

$\Gamma \vdash \vec{\tau}$ *$\vec{\tau}$ is a valid tuple type in typing context Γ* [p. 9]

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

$\Sigma \vdash P \qquad \Gamma \vdash P$ *P is a valid message predicate in signature Σ (viz. context Γ)* [pp. 10, 6]

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

$\Sigma \vdash S \qquad \Gamma \vdash \mathit{lhs}$ *S (viz. lhs) is a valid state (viz. predicate sequence) in signature Σ (viz. context Γ)* [pp. 11, 6]

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

$\Gamma \models^{r} \mathit{rhs}$ *rhs is a valid rule consequent in typing context Γ* [p. 13]

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

$\Gamma \vdash r$ *r is a valid rule in typing context Γ* [p. 14]

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

$\Gamma \vdash \rho$ — *ρ is a valid rule collection in typing context Γ* — [p. 15]

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

$\Sigma \vdash \mathcal{P}$ — *$\mathcal{P}$ is a valid protocol theory in signature Σ* — [p. 15]

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

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

$\Sigma \vdash R$ — *R is a valid active role set in signature Σ* — [p. 16]

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

A.3 Data Access Specification Rules

$\Gamma; \Delta \Vdash^{s}_{A} k \gg \Delta'$	*Given knowledge Δ, principal A can decipher a message encrypted with shared key k in context Γ*	[p. 23]
$\Gamma; \Delta \Vdash^{a}_{A} k \gg \Delta'$	*Given knowledge Δ, principal A can decipher a message encrypted with public key k in context Γ*	[p. 24]
$\Gamma; \Delta \Vdash_{A} \vec{t} \gg \Delta'$	*Given knowledge Δ and terms $\vec{t}$, principal A can knows Δ' in context Γ*	[p. 22]
$\Delta > \vec{e} > \Delta'$	*Merging context knowledge Δ and elementary term tuple $\vec{e}$ yields Δ'*	[p. 22]
$\Gamma; \Delta \Vdash_{A} lhs > \vec{t} \gg \Delta'$	*Given knowledge Δ, predicate sequence lhs and terms $\vec{t}$, principal A can knows Δ' in context Γ*	[p. 21]
$\Gamma \looparrowright_{A} e$	*Principal A can access atomic information e in context Γ*	[p. 26]
$\Gamma; \Delta \looparrowright_{A} t$	*Given knowledge Δ, principal A can construct term t*	[p. 26]
$\Gamma; \Delta \looparrowright_{A} \vec{t}$	*Given knowledge Δ, principal A can construct term tuple $\vec{t}$*	[p. 26]
$\Gamma; \Delta \looparrowright_{A} lhs$	*Predicate sequence lhs is constructible from knowledge Δ for principal A*	[p. 25]
$\Gamma; \Delta \Vdash_{A} rhs$	*Right-hand side rhs implements valid data access specification for principal A in context Γ given knowledge Δ*	[p. 25]
$\Gamma \Vdash_{A} r$	*Rule r implements valid data access specification for principal A in context Γ*	[p. 19]
$\Gamma \Vdash_{A} \rho$	*Rule sequence ρ implements valid data access specification for principal A in context Γ*	[p. 19]
$\Sigma \Vdash \mathcal{P}$	*Protocol theory $\mathcal{P}$ implements valid data access specification in signature Σ*	[p. 19]
$\Sigma \Vdash R$	*Active role set R implements valid data access specification in signature Σ*	[p. 29]

A COLLECTED RULES

$\Gamma; \Delta \Vdash^s_A k \gg \Delta'$ *Given knowledge Δ, principal A can decipher a message encrypted with shared key k in context Γ* [p. 23]

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

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

$\Gamma; \Delta \Vdash^a_A k \gg \Delta'$ *Given knowledge Δ, principal A can decipher a message encrypted with public key k in context Γ* [p. 24]

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

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

$\Gamma; \Delta \Vdash_A \vec{t} \gg \Delta'$ *Given knowledge Δ and terms $\vec{t}$, principal A can knows Δ' in context Γ* [pp. 22, 27]

$$\frac{}{\Gamma; \Delta \Vdash_A \cdot \gg \Delta}\ \textbf{tac\_dot} \qquad \frac{\Gamma; (\Delta, e) \Vdash_A \vec{t} \gg \Delta'}{\Gamma; (\Delta, e) \Vdash_A e, \vec{t} \gg \Delta'}\ \textbf{tac\_kn} \qquad \frac{(\Gamma, e : \tau, \Gamma'); (\Delta, e) \Vdash_A \vec{t} \gg \Delta'}{(\Gamma, e : \tau, \Gamma'); \Delta \Vdash_A e, \vec{t} \gg \Delta'}\ \textbf{tac\_ukn}$$

$$\frac{\Gamma; \Delta \Vdash_A t_1, t_2, \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A (t_1\ t_2), \vec{t} \gg \Delta'}\ \textbf{tac\_cnc}$$

$$\frac{\Gamma; \Delta \Vdash^s_A k \gg \Delta' \quad \Gamma; \Delta' \Vdash_A t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \{t\}_k, \vec{t} \gg \Delta''}\ \textbf{tac\_ske} \qquad \frac{\Gamma; \Delta \Vdash^a_A k \gg \Delta' \quad \Gamma; \Delta' \Vdash_A t, \vec{t} \gg \Delta''}{\Gamma; \Delta \Vdash_A \{\!\{t\}\!\}_k, \vec{t} \gg \Delta''}\ \textbf{tac\_pke}$$

$$\Rightarrow \qquad \frac{\Gamma; (\Delta, t) \Vdash_{\mathsf{A}} \vec{t} \gg \Delta'}{\Gamma; (\Delta, t) \Vdash_{\mathsf{A}} t, \vec{t} \gg \Delta'}\ \textbf{tac\_kn*} \qquad \Leftarrow$$

$\Delta > \vec{e} > \Delta' \qquad \Delta > \vec{t} > \Delta'$ *Merging context knowledge Δ and (elementary) term tuple $\vec{e}$ (viz. $\vec{t}$) yields Δ'* [pp. 22, 27]

$$\frac{}{\Delta > \cdot > \Delta}\ \textbf{mac\_dot} \qquad \frac{\Delta > \vec{e} > \Delta'}{\Delta > e, \vec{e} > (\Delta', e)}\ \textbf{mac\_ukn} \qquad \frac{\Delta > \vec{e} > \Delta'}{(\Delta, e) > e, \vec{e} > (\Delta', e)}\ \textbf{mac\_kn}$$

$$\Rightarrow \qquad \frac{\Delta > \vec{t} > \Delta'}{\Delta > t, \vec{t} > (\Delta', t)}\ \textbf{mac\_ukn*} \qquad \frac{\Delta > \vec{t} > \Delta'}{(\Delta, t) > t, \vec{t} > (\Delta', t)}\ \textbf{mac\_kn*} \qquad \Leftarrow$$

$\Gamma; \Delta \Vdash_A lhs > \vec{t} \gg \Delta'$ *Given knowledge Δ, predicate sequence lhs and terms $\vec{t}$, principal A can knows Δ' in context Γ* [pp. 21, 27]

$$\frac{\Delta > (A, \vec{e}) > \Delta' \quad \Gamma; \Delta' \Vdash_A lhs > \vec{t}' \gg \Delta''}{\Gamma; \Delta \Vdash_A (L(A, \vec{e}), lhs) > \vec{t}' \gg \Delta''}\ \textbf{lac\_rsp} \qquad \frac{\Gamma; \Delta \Vdash_A lhs > (t, \vec{t}') \gg \Delta''}{\Gamma; \Delta \Vdash_A (\mathsf{N}(t), lhs) > \vec{t}' \gg \Delta''}\ \textbf{lac\_net}$$

$$\frac{\Gamma; \Delta \Vdash_A lhs > (\vec{t}, \vec{t}') \gg \Delta'}{\Gamma; \Delta \Vdash_A (\mathsf{M}_A(\vec{t}), lhs) > \vec{t}' \gg \Delta'}\ \textbf{lac\_mem} \qquad \frac{\Gamma; \Delta \Vdash_A \vec{t} \gg \Delta'}{\Gamma; \Delta \Vdash_A \cdot > \vec{t} \gg \Delta'}\ \textbf{lac\_dot}$$

$$\Rightarrow \qquad \frac{\Delta > (\mathsf{A}, \vec{t}) > \Delta' \quad \Gamma; \Delta' \Vdash_{\mathsf{A}} lhs > \vec{t}' \gg \Delta''}{\Gamma; \Delta \Vdash_{\mathsf{A}} (L(\mathsf{A}, \vec{t}), lhs) > \vec{t}' \gg \Delta''}\ \textbf{lac\_rsp*} \qquad \Leftarrow$$

A COLLECTED RULES

$\Gamma \looparrowright_A e$ *Principal A can access atomic information e in context Γ* [p. 26]

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

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

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

$\Gamma; \Delta \looparrowright_A t$ *Given knowledge Δ, principal A can construct term t* [pp. 26, 27]

$$\frac{}{\Gamma; (\Delta, e) \looparrowright_A e}\ \textbf{cac\_kn} \qquad \frac{\Gamma \looparrowright_A e}{\Gamma; (\Delta, e) \looparrowright_A e}\ \textbf{cac\_ukn}$$

$$\frac{\Gamma; \Delta \looparrowright_A t_1 \quad \Gamma; \Delta \looparrowright_A t_2}{\Gamma; \Delta \looparrowright_A t_1\, t_2}\ \textbf{cac\_cnc} \qquad \frac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A k}{\Gamma; \Delta \looparrowright_A \{t\}_k}\ \textbf{cac\_ske} \qquad \frac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A k}{\Gamma; \Delta \looparrowright_A \{\!\{t\}\!\}_k}\ \textbf{cac\_pke}$$

$$\Rightarrow \qquad \frac{}{\Gamma; (\Delta, t) \looparrowright_A t}\ \textbf{cac\_gen}* \qquad \Leftarrow$$

$\Gamma; \Delta \looparrowright_A \vec{t}$ *Given knowledge Δ, principal A can construct term tuple $\vec{t}$* [p. 26]

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

$\Gamma; \Delta \looparrowright_A \mathit{lhs}$ *Predicate sequence lhs is constructible from knowledge Δ for principal A* [p. 25]

$$\frac{}{\Gamma; \Delta \looparrowright_A \cdot}\ \textbf{rac\_dot}$$

$$\frac{\Gamma; \Delta \looparrowright_A t \quad \Gamma; \Delta \looparrowright_A \mathit{lhs}}{\Gamma; \Delta \looparrowright_A \mathsf{N}(t), \mathit{lhs}}\ \textbf{rac\_net} \qquad \frac{\Gamma; \Delta \looparrowright_A \vec{t} \quad \Gamma; \Delta \looparrowright_A \mathit{lhs}}{\Gamma; \Delta \looparrowright_A \mathsf{M}_A(\vec{t}), \mathit{lhs}}\ \textbf{rac\_mem} \qquad \frac{\Gamma; \Delta \looparrowright_A (A, \vec{e}) \quad \Gamma; \Delta \looparrowright_A \mathit{lhs}}{\Gamma; \Delta \looparrowright_A L(A, \vec{e}), \mathit{lhs}}\ \textbf{rac\_rsp}$$

$\Gamma; \Delta \Vdash_A \mathit{rhs}$ *Right-hand side rhs implements valid data access specification for principal A in context Γ given knowledge Δ* [p. 25]

$$\frac{(\Gamma, x : \mathsf{nonce}); (\Delta, x) \Vdash_A \mathit{rhs}}{\Gamma; \Delta \Vdash_A \exists x : \mathsf{nonce}.\, \mathit{rhs}}\ \textbf{rac\_nnc} \qquad \frac{(\Gamma, x : \mathsf{msg}); (\Delta, x) \Vdash_A \mathit{rhs}}{\Gamma; \Delta \Vdash_A \exists x : \mathsf{msg}.\, \mathit{rhs}}\ \textbf{rac\_msg} \qquad \frac{\Gamma; \Delta \looparrowright_A \mathit{lhs}}{\Gamma; \Delta \Vdash_A \mathit{lhs}}\ \textbf{rac\_mn}$$

$\Gamma \Vdash_A r$ *Rule r implements valid data access specification for principal A in context Γ* [p. 19]

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

$\Gamma \Vdash_A \rho$ *Rule sequence ρ implements valid data access specification for principal A in context Γ* [p. 19]

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

$\Sigma \Vdash \mathcal{P}$ *Protocol theory $\mathcal{P}$ implements valid data access specification in signature Σ* [p. 19]

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

$\Sigma \Vdash R$ *Active role set R implements valid data access specification in signature Σ* [p. 29]

$$\frac{}{\Sigma \Vdash \cdot}\ \textbf{aac\_dot} \qquad \frac{\Sigma \Vdash R \quad \Sigma \Vdash_{\textsf{A}} \rho}{\Sigma \Vdash R, \rho^{\textsf{A}}}\ \textbf{aac\_ext}$$

A.4 Execution Rules

$(rhs)_\Sigma \gg (lhs)_{\Sigma'}$	*Right-hand side instantiation*	[p. 37]
$r \rhd [S]_\Sigma \gg [S']_{\Sigma'}$	*Rule application*	[p. 37]
$\mathcal{P} \rhd C \longrightarrow C'$	*One-step sequential firing*	[p. 36]
$\mathcal{P} \rhd C \longrightarrow^* C'$	*Multi-step sequential firing*	[p. 37]
$\mathcal{P} \rhd C \Longrightarrow C'$	*One-step parallel firing*	[p. 38]
$\mathcal{P} \rhd C \Longrightarrow^* C'$	*Multi-step parallel firing*	[p. 38]

$(rhs)_\Sigma \gg (lhs)_{\Sigma'}$ *Right-hand side instantiation* [p. 37]

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

$r \rhd [S]_\Sigma \gg [S']_{\Sigma'}$ *Rule application* [p. 37]

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

$\mathcal{P} \rhd C \longrightarrow C'$ *One-step sequential firing* [p. 36]

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

$$\frac{}{\mathcal{P} \rhd [S]^{R, (\exists L : \vec{\tau}.\, \rho)^{\textsf{A}}}_{\Sigma} \longrightarrow [S]^{R, ([\textsf{L}_l/L]\rho)^{\textsf{A}}}_{(\Sigma, \textsf{L}_l : \vec{\tau})}}\ \textbf{sex\_rsp} \qquad \frac{r \rhd [S]_\Sigma \gg [S']_{\Sigma'}}{\mathcal{P} \rhd [S]^{R, (r, \rho)^{\textsf{A}}}_{\Sigma} \longrightarrow [S']^{R, (\rho)^{\textsf{A}}}_{\Sigma'}}\ \textbf{sex\_rule}$$

$$\frac{}{\mathcal{P} \rhd [S]^{R, (r, \rho)^{\textsf{A}}}_{\Sigma} \longrightarrow [S]^{R, (\rho)^{\textsf{A}}}_{\Sigma}}\ \textbf{sex\_skp} \qquad \frac{}{\mathcal{P} \rhd [S]^{R, (\cdot)^{\textsf{A}}}_{\Sigma} \longrightarrow [S]^{R}_{\Sigma}}\ \textbf{sex\_dot}$$

$\mathcal{P} \rhd C \longrightarrow^* C'$ *Multi-step sequential firing* [p. 37]

$$\frac{}{\mathcal{P} \rhd C \longrightarrow^* C}\ \textbf{sex\_it0} \qquad \frac{\mathcal{P} \rhd C \longrightarrow C' \quad \mathcal{P} \rhd C' \longrightarrow^* C''}{\mathcal{P} \rhd C \longrightarrow^* C''}\ \textbf{sex\_itn}$$

$\mathcal{P} \triangleright C \implies C'$ *One-step parallel firing* [p. 38]

$$\frac{}{\mathcal{P} \triangleright C \implies C}\ \textbf{pex\_id}$$

$$\frac{\mathcal{P} \triangleright [S_1]^{R_1}_{\Sigma} \longrightarrow [S_1']^{R_1'}_{(\Sigma,\Sigma_1')} \quad \mathcal{P} \triangleright [S_2]^{R_2}_{\Sigma} \implies [S_2']^{R_2'}_{(\Sigma,\Sigma_2')}}{\mathcal{P} \triangleright [S_1, S_2]^{(R_1,R_2)}_{\Sigma} \implies [S_1', S_2']^{(R_1',R_2')}_{(\Sigma,\Sigma_1',\Sigma_2')}}\ \textbf{pex\_par}$$

$\mathcal{P} \triangleright C \implies^* C'$ *Multi-step parallel firing* [p. 38]

$$\frac{}{\mathcal{P} \triangleright C \implies^* C}\ \textbf{pex\_it0}$$

$$\frac{\mathcal{P} \triangleright C \implies C' \quad \mathcal{P} \triangleright C' \implies^* C''}{\mathcal{P} \triangleright C \implies^* C''}\ \textbf{pex\_itn}$$