

forward

a future of reliable wireless ad hoc networks of roaming devices

On a Calculus for Security Protocol Synthesis

September 28, 2004

Authorisation

Dr. Gavin Lowe
FORWARD Steering Committee Member
PP. Dr. Sadie Creese
FORWARD Steering Committee Member

Date: 30th September 2004

Authors

Michael Auty, Oxford University, mike.auty@comlab.ox.ac.uk

Gavin Lowe, Oxford University, gavin.lowe@comlab.ox.ac.uk

Executive Summary

This report forms deliverable D10 of the FORWARD project, the third deliverable in Task 1.3 of Work Package 1 Authentication and Key Management. The aim of Work Package 1 is to enable key management solutions for the Next Wave, by investigating the feasibility of novel authentication and key management solutions and by enabling the design and assessment of authentication and key management systems for the pervasive paradigm. This report provides a review of work conducted on Task 1.3 *Synthesis and Composition of Security Protocols*.

The research presented here provides significant progress towards a calculus for synthesising security protocols. Protocol descriptions are annotated with assertions that state properties that will be true when the protocol execution reaches that point. Proof rules are given that allow the assertions to be verified. A novel feature of the calculus is that the initial development of a protocol uses abstract messages that describe the intention of a message, rather than the concrete implementation; rules are given that allow these abstract messages to be suitably implemented.

A formal semantic model of protocol executions is presented. This is used to give a formal, precise meaning to protocol annotations and to abstract messages. It is also used to verify a number of annotation rules; significant progress is made towards verifying rules for implementing abstract messages. The calculus is illustrated with the development of a well know protocol; the development helps to cast light on the failure of an earlier version of that protocol.

Contents

1	Introduction	1
2	Example	3
2.1	First steps	3
2.2	Creating nonces	3
2.3	Getting the message across	4
2.4	Is there anybody out there?	5
2.5	A concrete refinement	6
2.6	Composability through independence	7
3	Protocol semantics	10
3.1	Messages	10
3.2	Abstract messages	11
3.3	Local states	12
3.3.1	Protocol templates	13
3.3.2	Bindings	14
3.3.3	Local states	14
3.3.4	Operational semantics	14
3.4	Feasible protocols	16
3.5	The intruder	18
3.6	Global states	18
3.6.1	Operational semantics	18
3.7	Protocol semantics	19
4	Annotations	21
4.1	Correctness of annotations	21
4.2	Structural annotation rules	21
4.3	Annotation macros	23
4.3.1	<i>knows</i>	23
4.3.2	<i>session</i>	24
4.3.3	<i>honest</i>	25
4.3.4	<i>Always</i>	25
4.4	<i>new x</i>	26
5	Disjoint encryption	28
5.1	A theorem about agreement	28
5.2	A theorem about secrecy	30
6	Abstract messages	36
6.1	Refinement	36
6.2	Concrete messages	37
6.2.1	Semantics	37
6.3	Conjunction	37
6.4	<i>any</i>	38
6.5	<i>canExtract</i> and <i>keepSecret</i>	38
6.5.1	Annotation rules	39
6.5.2	Refinement rules	40
6.6	<i>provesKnowledgeOf</i>	40
6.6.1	Semantics	41
6.6.2	Annotation rules	41
6.6.3	Example refinements	43

6.7	<i>bind</i>	43
6.7.1	Semantics	43
6.7.2	Proof rule	44
6.7.3	Example refinements	45
7	The Needham Schroeder Public Key Protocol	46
8	Tool support for creating concrete messages	49
8.1	A prolog representation for sets of messages	49
8.2	Encoding refinement rules	50
8.3	The verification step	51
8.4	Examples	54
9	Conclusions	56
9.1	Summary	56
9.2	Related Work	56
9.3	Future Work	56

1 Introduction

Creating security protocols is a difficult task. Numerous security protocols have been published, only later to be discovered to be flawed; for example, the Needham Schroeder Public Key Protocol was first published in 1978 [NS78], and was the subject of several subsequent analyses (e.g. [BAN89]), only to be found to be flawed in 1995 [Low95].

Various approaches to analysing protocols have been proposed. State exploration techniques (for example [Low96, Low98, MCJ97, MMS97]) build a model of the state space of a small instance of the protocol (with a bounded number of protocol runs), together with a model of the most general attacker who can interact with the protocol, and then use a tool to explore the state space, looking for insecure states. Theorem provers have been used to produce machine-assisted proofs of protocols (for example [Pau98, Coh00]). The NRL Analyzer [Mea96] combines automated theorem proving with state space analysis techniques. Protocols have been verified directly by hand using special-purpose logics such as BAN Logic [BAN89], or GNY Logic [GNY90]. The Strand Spaces approach [THG99] builds a special-purpose model of protocols; the protocols are then either proved by hand, or automatically (for example using Athena [SBP01]).

Proving the security of a protocol, with any of these methods, is non-trivial; inventing a security protocol from scratch is even harder. An easier method of protocol creation and verification could be to synthesise the protocol from its requirements. This report outlines a calculus for protocol synthesis that allows protocols to be built from small components, and provides a method of quickly proving the security of protocols composed from previously proven security protocols.

Throughout this report, we work in the Dolev-Yao Model [DY83]. We assume that the network is under the complete control of a malicious agent or intruder. The intruder can intercept all messages passing on the network, and can send fake messages, possibly claiming to come from a different agent, provided he can create those messages from those he has already seen or knew initially, by encrypting or decrypting with known keys, concatenating or splitting pairs, or hashing. However, we assume perfect cryptography, so we assume that the intruder cannot learn anything from a ciphertext if he does not know the appropriate decrypting key.

Protocols in our calculus do not take the form of a standard protocol specification, where each message specifies exactly *how* it should be implemented, built from atomic pieces of data, using concatenation, encryption and hashing. Instead protocols in our calculus use *abstract messages* which convey *what* each message is supposed to do. Abstract messages represent requirements on the corresponding concrete messages, and do not specify how these requirements are achieved. The calculus provides various abstract messages, each providing a single requirement on the concrete message; these can then be conjoined to make stronger requirements. This makes it much simpler to see what a protocol is trying to do at each step, and helps ensure that a message is not doing something unintended.

Our calculus adapts the idea of program annotations [Hoa69] from programs to security protocols. We annotate the protocol description with assertions that state properties that will be true when a protocol reaches that point. More precisely, each protocol annotation will be from the point of view of a single participant: each assertion will state properties that are guaranteed to be true whenever the participant in question reaches that point in the protocol. We write $\{pre\} e \{post\}$ to mean that if the sequence of events e is executed, starting from a state where pre holds, then it can be guaranteed that $post$ will hold in the final state.

We present proof rules that allow assertions to be verified, based on the abstract messages used in each step, assuming the assertion preceding that step holds. The calculus thus allows protocols to be synthesised and simultaneously proved correct. It also allows protocols to be composed, by matching the final assertion of one with the initial assertion of the next.

In order to explain precisely the meaning of the constructs of the calculus, and in order to ensure the proof rules are correct, we provide a semantic model. In particular, we present semantics for the abstract messages, stating formally what each abstract message means.

Lastly the calculus needs a way to translate abstract messages into a concrete form, to allow

proven protocols to be implemented. We discuss rules for the refinement of abstract messages; verifying these rules is left for future work.

To help the reader understand the various elements involved in this calculus, we give a simple worked example in Section 2, explaining briefly each of the elements. In Section 3 we outline the semantic model upon which the calculus is based. We formalise the meaning of annotations in Section 4, verify some structural annotation rules, and define some useful macros for use in annotations. In Section 5 we define a particular property enjoyed by some protocols, namely disjoint encryption: that different encrypted components within the protocol have distinct forms; we prove two theorems, concerning agreement and secrecy, which we believe will be useful in verifying message refinement rules. We study abstract messages in more detail in Section 6: we present a semantic definition for each abstract message, and rules to allow the abstract messages to be used in annotations; we discuss rules to refine the abstract messages to concrete messages. In Section 7 we look at a larger example, namely that of the Adapted Needham Schroeder Public Key Protocol [Low95]: we derive the protocol using the rules presented earlier; our development gives some insight into the failings of the original version of the protocol. In Section 8 we describe a prototype implementation of tool support for the message refinement rules. Finally, in Section 9, we sum up and discuss related work and future directions for the research.

2 Example

In order to illustrate the main features of the calculus we will use it to develop a small protocol. The protocol merely establishes a shared secret, and provides one way authentication and liveness properties. We will progress through the development of a protocol proof, explaining the various constructions as necessary.

2.1 First steps

We begin by specifying precisely what we require our protocol to do, defining both the assumptions we will make at the start, and the properties we need to hold at the end. In this particular example, the protocol will make use of a nonce challenge to provide fresh authentication of the agent b . We will assume that a and b already share the key k , that a and b are different agents, and that they are both honest (have predictable behaviour). We would like to reach a state in which agent a can be certain that agent b has a session running with the correct value for na . Since we will require that a and b share the key, and that it is not released during the protocol we can make these facts into an invariant, a predicate that must be true at each state reached in the protocol.

$$\begin{array}{l} \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\ \{ I \} \\ \dots \\ \{ I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na) \} \end{array}$$

Figure 1: Initial protocol specification

As can be seen in Figure 1 we annotate protocols in a style similar to Hoare triples [Hoa69]. Every annotated protocol is taken from the perspective of a particular honest participant: all annotations in this section are taken from the perspective of a . The annotations specify statements that are guaranteed to hold when the participant involved reaches that point in the protocol.

In this example, we assume that initially the invariant must be true; this represents the precondition of the protocol. At the end of the protocol the invariant must still hold, but extra conditions must also hold; these represent the postcondition of the protocol. The ellipses (“...”) represent the part of the protocol that we still need to develop.

This example uses two macros, the function *knows* and the predicate *session*.

- The set $\text{knows}(x)$ returns all participants who know the piece of data x at this point in the protocol. The set $\text{knows}(x)$ can change from state to state. In this particular example, the invariant states that only the agents a and b could know the key k during the protocol run.
- The predicate $\text{session}(b; x)$ states that the participant b is participating in a session of the protocol and has the value x associated with the correct variable in its local state. Here it is used at the end of the protocol to show that once complete there is an instance of agent b that has the value na bound.

2.2 Creating nonces

We will now construct the nonce, and show the facts we can deduce about the state once the nonce has been created. We add our first state modifying construction, as shown in Figure 2.

The new na event creates a new nonce within the local state. Afterwards, the state should be the same as before, except now containing a new value bound to the variable na , known only to the

$$\begin{aligned}
 & \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
 & \left\{ I \right\} \\
 & a : \text{new } na \\
 & \left\{ I \wedge \text{knows}(na) = \{a\} \right\} \\
 & \dots \\
 & \left\{ I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na) \right\}
 \end{aligned}$$

Figure 2: Protocol annotation showing nonce creation

participant that created it; this is captured by stating that $\text{knows}(na)$ is now precisely the set $\{a\}$. This is justified by the following proof rule:

$$\begin{array}{c}
 \left\{ P \right\} a : \text{new } x \left\{ P \wedge \text{knows}(x) = \{a\} \right\} \\
 \text{provided } x \text{ is not free in } P, \text{ and } P \text{ refers only to state variables.}
 \end{array}$$

We will often concatenate several events and corresponding assertions; for example, in Figure 2, the new na and resulting assertion is concatenated with the part of the protocol still to be developed. The following proof rule justifies this.

$$\frac{\begin{array}{c} \{pre\} e_1 \{mid\} \\ \{mid\} e_2 \{post\} \end{array}}{\{pre\} e_1 e_2 \{post\}}$$

We write the resulting annotation, corresponding to the consequence of this rule, as

$$\{pre\} e_1 \{mid\} e_2 \{post\}$$

2.3 Getting the message across

We have now reached a stage in the protocol development where a message needs to be sent. The send event where participant a sends message m is denoted by:

$$a : \text{send } m.$$

It should be noted that when reasoning about the security of a protocol, the intended recipient is not needed (and not given, after the arrow) since within the Dolev-Yao [DY83] model, the intruder could intercept and redirect any message sent.

Figure 3 gives the next stage in the annotation. It uses two abstract messages:

- $\text{canExtract}_k(x)$ is an abstract message that specifies that the data x can be extracted only by a participant who knows k . Here it is used to allow a participant in the set $\text{knows}(k)$, i.e. $\{a, b\}$, to extract the nonce.
- $\text{keepSecret}(k)$ is a message which does not allow k to be learnt by any participant. It is used here to maintain the invariant that $\text{knows}(k)$ does not change from $\{a, b\}$.

These two abstract messages have been conjoined, which means that the concrete message sent must have the properties of both abstract messages, in this case a message which reveals na only to a participant who knows k , but which does not reveal k under any circumstances.

This step leads us from a state where $I \wedge \text{knows}(na) = \{a\}$ holds, to a state where $I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a\} \cup \text{knows}(k)$ holds: the $\text{keepSecret}(k)$ message requires that $\text{knows}(k)$ after

$$\begin{array}{l}
\text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
\{I\} \\
a : \text{new } na \\
\{I \wedge \text{knows}(na) = \{a\}\} \\
a : \text{send } \text{canExtract}_k(na) \wedge \text{keepSecret}(k) \\
\{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a\} \cup \text{knows}(k)\} \\
\{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\}\} \\
\dots \\
\{I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na)\}
\end{array}$$

Figure 3: Annotation including nonce challenge

is the same as $\text{knows}(k)$ before (thus maintaining the invariant); the $\text{canExtract}_k(na)$ message expands the set of people who could know na by possibly adding those who could know k ($\text{knows}(k)$). We give rules to formally justify this step later.

The assertion immediately after the send event can be simplified to $I \wedge a \subseteq \text{knows}(na) \subseteq \{a, b\}$. This simplification is justified by the following rule:

$$\frac{\{pre\}e\{post\} \quad post \Rightarrow post'}{\{pre\}e\{post'\}}$$

We will tend to write the resulting annotation as $\{pre\}e\{post\}\{post'\}$.

For completeness we also present the complimentary rule:

$$\frac{\{pre\}e\{post\} \quad pre' \Rightarrow pre}{\{pre'\}e\{post\}}$$

We will tend to write the resulting annotation as $\{pre'\}\{pre\}e\{post\}$.

2.4 Is there anybody out there?

The protocol so far has established that b could know the fresh nonce na , but since a has not received anything from the outside world, he cannot yet deduce that b does in fact know na . For that he will have to receive a message which in some way shows him that someone knows na ; from that and the other conditions that hold, we can deduce that in fact it can only be b that knows na .

Figure 4 shows a 's receipt of an abstract message. The receive event, of participant a receiving message m is denoted by:

$$a : \text{receive } m$$

The supposed sender is not specified, for similar reasons to why the intended recipient is not specified in send events: the recipient cannot be sure of the identity of the sender.

This message centres around $\text{provesKnowledgeOfNR}(na)$, which informs the receiver a that somebody knows the value of na ; further, that participant is not a himself: this extra clause ensures that messages are not reflected (the "NR" stands for "not reflected"); this will allow us to deduce that this message will prove that someone *else* knows na . This gives us the $\exists b' \bullet \text{session}(b'; na) \wedge b' \neq a$ clause of the assertion.

$$\begin{array}{l}
 \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
 \{I\} \\
 a : \text{new } na \\
 \{I \wedge \text{knows}(na) = \{a\}\} \\
 a : \text{send } \text{canExtract}_k(na) \wedge \text{keepSecret}(k) \\
 \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a\} \cup \text{knows}(k)\} \\
 \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\}\} \\
 a : \text{receive } \text{provesKnowledgeOfNR}(na) \wedge \text{keepSecret}(k) \wedge \text{keepSecret}(na) \\
 \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\} \wedge \exists b' \bullet \text{session}(b'; na) \wedge b' \neq a\} \\
 \{I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na)\}
 \end{array}$$

Figure 4: Completed protocol annotations

The message also uses $\text{keepSecret}(k)$, which will ensure that the invariant is maintained, and $\text{keepSecret}(na)$ which stops na being learnt by others.

We can now deduce that since someone does exist who has na in their state, and that person is not a , and since the only people who know na (and thus could have na in their state) are a and b (since $\text{knows}(na) \subseteq \{a, b\}$), that the person who has na in their state must be b , i.e. $\text{session}(b; na)$. Finally, this tells us that $b \in \text{knows}(na)$ so $\text{knows}(na) = \{a, b\}$. This establishes the required postcondition.

2.5 A concrete refinement

It should be noted that the abstract messages do not specify how their requirements should be met, merely what properties they must achieve; for example $\text{canExtract}_k(x)$ does not specify that encryption must be used. We write $m \sqsubseteq m'$ if message m' meets the requirements of m ; typically, m will be an abstract message, and m' a concrete implementation.

There will often be several concrete implementations for each abstract message. In fact there may be several messages which fulfil this requirement only within the context of a particular protocol (perhaps relying on the particular concrete implementations of earlier messages); however, we will tend to use implementations that fulfil the requirements in *all* protocols.

For this example, we can refine the first abstract message as follows. Firstly, note that

$$\text{canExtract}_k(na) \sqsubseteq \{na\}_k,$$

since encryption with a symmetric key k means that only a possessor of k can extract the contents. Further

$$\text{keepSecret}(k) \sqsubseteq \{na\}_k,$$

since in our model, which assumes perfect cryptography, k cannot be learnt from $\{na\}_k$. Hence

$$\text{canExtract}_k(na) \wedge \text{keepSecret}(k) \sqsubseteq \{na\}_k.$$

This last step is justified by the following proof rule concerning the refinement of the conjunction of abstract messages:

$$\frac{
 \begin{array}{l}
 m_1 \sqsubseteq m \\
 m_2 \sqsubseteq m
 \end{array}
 }{
 m_1 \wedge m_2 \sqsubseteq m
 }$$

Similarly the second abstract message can be refined by hashing na with the identity of the sender, and implementing a check in the concrete protocol not to accept the message if the identifier included in the hash is not as expected. We have

$$\begin{aligned} \text{provesKnowledgeOfNR}(na) &\sqsubseteq h(na, b), \\ \text{keepSecret}(k) &\sqsubseteq h(na, b), \\ \text{keepSecret}(na) &\sqsubseteq h(na, b). \end{aligned}$$

and hence

$$\left(\begin{array}{l} \text{provesKnowledgeOfNR}(na) \wedge \\ \text{keepSecret}(k) \wedge \text{keepSecret}(na) \end{array} \right) \sqsubseteq h(na, b).$$

This gives us the verified concrete protocols given in Figure 5.

$$\begin{array}{l} \text{Invariant } I \cong \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\ \{I\} \\ a : \text{new } na \\ \{I \wedge \text{knows}(na) = \{a\}\} \\ a : \text{send}\{na\}_k \\ \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a\} \cup \text{knows}(k)\} \\ \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\}\} \\ a : h(na, b) \\ \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\} \wedge \exists b' \bullet \text{session}(b'; na) \wedge b' \neq a\} \\ \{I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na)\} \end{array}$$

Figure 5: Concrete implementation 1

However this abstract message could equally be refined by encrypting the identity of the sender, using the nonce as the key; such a message will not be accepted if the identity is not as expected.

$$\begin{aligned} \text{provesKnowledgeOfNR}(na) &\sqsubseteq \{b\}_{na} \\ \text{keepSecret}(k) &\sqsubseteq \{b\}_{na} \\ \text{keepSecret}(na) &\sqsubseteq \{b\}_{na} \end{aligned}$$

Because of the perfect cryptography assumptions, this message will not reveal na ; k does not appear and so cannot be revealed. Hence

$$\left(\begin{array}{l} \text{provesKnowledgeOfNR}(na) \wedge \\ \text{keepSecret}(k) \wedge \text{keepSecret}(na) \end{array} \right) \sqsubseteq \{b\}_{na}.$$

This gives us the concrete protocol, given in Figure 6.

2.6 Composability through independence

The logic makes intuitive reasoning about protocols much simpler to formalise. However, to produce a valid proof of the security of a protocol we need to think about unexpected protocol interactions, mostly notably the interactions between two separate protocols, or between different parts of a protocol.

$$\begin{array}{l}
 \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
 \{I\} \\
 a : \text{new } na \\
 \{I \wedge \text{knows}(na) = \{a\}\} \\
 a : \text{send}\{na\}_k \\
 \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a\} \cup \text{knows}(k)\} \\
 \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\}\} \\
 a : \text{receive}\{b\}_{na} \\
 \{I \wedge \{a\} \subseteq \text{knows}(na) \subseteq \{a, b\} \wedge \exists b' \bullet \text{session}(b'; na) \wedge b' \neq a\} \\
 \{I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na)\}
 \end{array}$$

Figure 6: Concrete implementation 2

As an example of such interactions, consider the following two protocols. Protocol α encrypts a secret with the recipient's public key, for secrecy:

Message $\alpha.1$ $a \rightarrow b : \{\text{secret}\}_{pk(b)}$.

Protocol β uses a nonce challenge: a encrypts na with b 's public key, and b returns it as plaintext:

Message $\beta.1$ $a \rightarrow b : \{na\}_{pk(b)}$

Message $\beta.2$ $b \rightarrow a : na$.

There is an obvious interaction between these protocols:

Message $\alpha.1$ $a \rightarrow I_b : \{\text{secret}\}_{pk(b)}$

Message $\beta.1$ $I_a \rightarrow b : \{\text{secret}\}_{pk(b)}$

Message $\beta.2$ $b \rightarrow I_a : \text{secret}$.

The intruder replays the message from protocol α into protocol β , to trick b into decrypting the secret and revealing it.

One possible way to protect against this would be to require a completely disjoint key space (which means that no two keys can be shared between protocols); however this would require that every protocol have a different key, which would quickly become infeasible.

Another method of ensuring that two protocols cannot interact in unforeseen ways is to use disjoint encryption. Disjoint encryption between two protocols holds if every encrypted component of one protocol has a different form than every encrypted component from the other protocol. This means that no message involving encrypted components from one protocol can be forged and passed off as being from the other protocol, since it will be identified and rejected by the recipient.

In [GTF00], Guttman and Thayer prove a protocol independence result based on disjoint encryption: they show that if two protocols are secure in isolation, and they use disjoint encryption, then they are secure when combined.

Disjoint encryption is easy to achieve, by including a unique protocol identifier within each encryption, thus differentiating encrypted components of a similar form between two protocols (since they will have different identifiers).

We will assume no interactions between the protocol being developed and others; this is a reasonable assumption because of the above result.

Similarly, some protocols suffer from interactions between different messages within the protocol: an encrypted component from one place in the protocol can be passed off in another place in the

protocol. However, suppose that the separate messages of the protocol satisfy the disjoint encryption property; then such replays are not possible: each message can effectively be considered a sub-protocol, so the messages are independent by Guttman and Thayer's result. Many of our message refinement rules will depend upon there being no interactions from other parts of the protocol: they will have a side condition that the protocol as a whole satisfies the disjoint encryption property. During the initial part of a protocol development, while dealing with abstract messages, we will assume no interactions. In the final step of converting abstract messages into concrete form, we will ensure that there are no interactions by enforcing disjoint encryption.

These assumptions do not ensure that two message components are from the same run of a protocol, merely that they are from the same protocol and from the same step within that protocol. It is the responsibility of the protocol designer to ensure that if two message components from the same position in the same protocol need to be distinguished further (for instance between runs with two different parties) that all parties involved can tell the difference by the data received in the messages.

3 Protocol semantics

In this section we build a semantic model of protocol executions; in later sections we build on this to give a semantics to annotations, give a semantics to abstract messages, and prove annotation and refinement rules.

We begin, in Section 3.1 by defining the types of messages and message templates. In Section 3.2 we define abstract messages. We define the local states of agents in Section 3.3, and give an operational semantics. In Section 3.4 we consider what it means for a protocol to be feasible for a particular agent. We describe the model of the intruder in Section 3.5. We combine these in Section 3.6 to give the model of a global state, and lift the operational semantics for individual agents to the global level. Finally we define the semantics of a protocol in Section 3.7.

3.1 Messages

We begin by defining the type of actual messages. It is important to distinguish between message templates and actual messages: the former contain free variables, and are used in the definition of a protocol; the latter have all the variables instantiated with values, and are what are actually sent across the network.

We assume disjoint types Var of variables and Val of atomic values. We use variables for two purposes within our model: to represent fields within a protocol definition, or as program variables storing values in agents' states. We use the convention of representing variables by small letters and values by capitals. We also assume the existence of a special value $\perp \notin Val$, representing an undefined value, and define $Val^\perp \triangleq Val \cup \{\perp\}$.

We assume two inverse functions:

$$\begin{aligned} _^{-1var} &: Var \leftrightarrow Var, \\ _^{-1val} &: Val \leftrightarrow Val. \end{aligned}$$

between variables and values. If M is a value then messages encrypted with M can be decrypted with M^{-1val} , and vice versa. If x is a variable then it is intended that x^{-1var} holds the corresponding decrypting key. We assume that $(x^{-1var})^{-1var} = x$, and similarly for values. We will drop the val and var subscripts where that will not cause confusion.

We define actual messages and message templates by the grammars:

$$\begin{aligned} Msg &::= Val \mid (Msg, Msg) \mid \{Msg\}_{Val}, \\ Template &::= Var \mid Val \mid (Template, Template) \mid \{Template\}_{Var}. \end{aligned}$$

Messages and templates are built up from atomic values by pairing and encryption. We omit parentheses where appropriate. Note that an actual message can be obtained from a template by substituting or instantiating all the free variables with values. We use the convention of representing templates by small letters and actual messages by capitals.

We define a submessage relation \preceq for use later; first over messages:

$$\begin{aligned} X \preceq Y &\Leftarrow X = Y, \\ X \preceq (M_1, M_2) &\Leftarrow X \preceq M_1 \vee X \preceq M_2, \\ X \preceq \{M\}_K &\Leftarrow X \preceq M \vee X \preceq K \vee X \preceq K^{-1}. \end{aligned}$$

And then over templates:

$$\begin{aligned} m \preceq m' &\Leftarrow m = m', \\ m \preceq (m_1, m_2) &\Leftarrow m \preceq m_1 \vee m \preceq m_2, \\ m \preceq \{m'\}_k &\Leftarrow m \preceq m' \vee m \preceq k \vee m \preceq k^{-1}. \end{aligned}$$

Note in particular that the *decrypting* key is a submessage of an encryption. We extend the submessage relation (over messages) to take a set of messages on the right:

$$X \preceq B \Leftrightarrow \exists M \in B \bullet X \preceq M.$$

It will also be useful to talk about direct submessages: those submessages that can be obtained without performing any decryption:

$$\begin{aligned} m \ll m' &\Leftarrow m = m', \\ m \ll (m_1, m_2) &\Leftarrow m \ll m_1 \vee m \ll m_2. \end{aligned}$$

We will make the *strong typing assumption*: i.e. that each honest agent will only accept a value received if it is of the expected type. See [HLS03] for an implementation of this assumption.

We assume a type *TypeName* of names of atomic types (e.g. *Nonce*, *PublicKey*, *AgentIdentity*, ...). We then define a type of types of messages by

$$Type ::= TypeName \mid (Type, Type) \mid \{Type\}_{Type}.$$

For example, $\{(Nonce, AgentIdentity)\}_{PublicKey}$ represents the type of nonces and agent identities encrypted with public keys.

We assume a function

$$type_{var} :: Var \rightarrow Type$$

giving the intended types of all variables in the system. Note that this means that if the definitions of the protocols for two nodes make use of the same variable name, then they must both give the same type to that variable. We also assume a function

$$type_{val} :: Val \rightarrow Type.$$

That gives the types of atomic values. We lift the functions to message templates and messages

$$\begin{aligned} type_{template} &:: Template \rightarrow Type, \\ type_{msg} &:: Msg \rightarrow Type \end{aligned}$$

in the obvious way. We'll drop the subscripts from the $type_*$ functions where that will not cause confusion.

3.2 Abstract messages

In this section we briefly describe the ideas behind abstract messages, and how they are modelled formally. We postpone some of the details to Section 6.

The idea behind abstract messages is that most protocol designers know what they are trying to achieve, but have to write concrete message templates which may have other meanings, or which do not entirely capture the intended meaning. The abstract messages provide the designer with a means to express what the message *should* do, not how to implement it. The concrete implementation of the abstract message can be determined later, and in fact there may be several possible concrete implementations of the same abstract message, as seen in Section 2.5.

We consider abstract messages defined by the grammar

$$\begin{aligned} AbsMsg ::= & Template \mid AbsMsg \wedge AbsMsg \mid canExtract_{Var}(Var) \mid \\ & keepSecret(Var) \mid provesKnowledgeOfNR(Var) \mid \dots \end{aligned}$$

We have left the grammar open, as we will introduce more abstract messages in Section 6, and we suspect that the study of further example developments will suggest yet more useful abstract

messages. Note in particular that a concrete message template is considered to be an abstract message.

We define the semantics of an abstract message to be the set of all the concrete message templates that meet the desired property. The semantics may be dependent upon the protocol: for instance, in one protocol a message may prove knowledge of a value x — and so be an implementation of $provesKnowledgeOfNR(x)$ — by revealing a different value y that was previously encrypted with x ; however, this won't be the case in all protocols. For this reason we use a semantic function that takes the abstract message and the particular protocol in question, and returns the semantics (set of possible concrete message templates) for that abstract message. We write $\llbracket m \rrbracket_{\Pi}$ for the semantics of abstract message m in protocol Π :

$$\llbracket - \rrbracket_{\Pi} : AbsMsg \times Protocol \rightarrow \mathbb{P} Template.$$

In Section 6 we will give the semantics of each form of abstract message, together with rules for using those abstract messages in annotations, and refining them to concrete messages. However, it is worth giving the semantics of a concrete message template here: it is simply the singleton set containing that concrete template.

$$\llbracket m \rrbracket_{\Pi} = \{m\}, \quad \text{for } m \in Template.$$

Recall also that we write $am \sqsubseteq am'$ if abstract message am can be implemented by am' . We define a protocol-dependent notion of refinement by

$$am \sqsubseteq_{\Pi} am' \Leftrightarrow \llbracket am \rrbracket_{\Pi} \supseteq \llbracket am' \rrbracket_{\Pi},$$

and define a protocol-independent refinement by

$$am \sqsubseteq am' \Leftrightarrow \forall \Pi \bullet am \sqsubseteq_{\Pi} am',$$

where the quantification is taken over all protocols Π that satisfy the disjoint encryption property.

Note that there are two degrees of freedom within an abstract message: the choice (made during the design of the protocol) of concrete message template with which to implement it; and the choice (made at run-time) of values to instantiate the free variables.

It is worth considering the implications of the fact that the refinement relation is parameterised by the protocol Π . There are two scenarios to consider:

- The final protocol is known, and a rational construction or verification is being performed. In this case, each refinement step can be verified against the protocol in question.
- The final protocol is not known, but is being developed. Some of the refinement rules we give later will include conditions on the protocol Π , such as the disjoint encryption property. If such a refinement rule is used, the conditions need to be checked against the part of the protocol developed so far, and borne in mind for the remainder of the development, or checked at the end.

3.3 Local states

Our global state will comprise a number of honest agents, or *nodes*, together with an intruder, which communicate together. In this section we describe how we model the local states of honest agents. The model includes the program, defining how the agent acts, and the binding of variables to values. We give an operational semantics showing how the local state evolves as events are performed.

3.3.1 Protocol templates

Part of the state of an honest agent will be a definition of the sequence of events that it should perform. As with messages, we distinguish between templates for events (using abstract messages), and the actual events themselves (described below in Section 3.3.4).

We consider four types of *event templates* performed by protocol participants:

send The event template $\text{send } m$ represents the sending of a message described by the abstract message m ;

receive The event template $\text{receive } m$ represents the receipt of a message described by the abstract message m ;

new The event template $\text{new } x$ represents the fresh generation of a value to be stored in the variable x ;

newpair The event template $\text{new}(x, y)$ represents the fresh generation of a asymmetric key pair to be stored in the variables x and y ; we specify that x and y should be inverses in this case: $y = x^{-1var}$.

Note that we generate both members of a key pair together. To enforce this, we will ban the use of the construct $\text{new } x$ for x an asymmetric key (i.e. where $x \in \text{dom } _^{-1} \wedge x^{-1} \neq x$).

Formally, event templates are defined by the grammar

$$\text{EventTemplate} ::= \text{send } \text{AbsMsg} \mid \text{receive } \text{AbsMsg} \mid \\ \text{new } \text{Var} \mid \text{newpair}(\text{Var}, \text{Var}).$$

We lift the refinement relation from abstract messages to event templates in the obvious way:

$$\begin{aligned} \text{send } m \sqsubseteq_{\Pi} \text{send } m' &\Leftrightarrow m \sqsubseteq_{\Pi} m', \\ \text{receive } m \sqsubseteq_{\Pi} \text{receive } m' &\Leftrightarrow m \sqsubseteq_{\Pi} m', \\ \text{new } x \sqsubseteq_{\Pi} \text{new } x, \\ \text{newpair}(x, y) \sqsubseteq_{\Pi} \text{newpair}(x, y). \end{aligned}$$

A *program* for an honest agent is then the sequence of event templates to be followed:

$$\text{Prog} \hat{=} \text{EventTemplate}^*.$$

We write prog for a typical element of Prog .

If a program contains only concrete messages, i.e. every send or receive event template is of a concrete message $M \in \text{Template}$, then we say that it is a *concrete program*.

We lift the notion of refinement from event templates to programs point-wise:¹

$$\begin{aligned} \text{prog} \sqsubseteq_{\Pi} \text{prog}' &\Leftrightarrow \text{length } \text{prog} = \text{length } \text{prog}' \wedge \\ &\forall i \in 1 \dots \text{length } \text{prog} \bullet \text{prog}(i) \sqsubseteq_{\Pi} \text{prog}'(i). \end{aligned}$$

We write $\text{vars}(m)$ for the set of variables appearing in message template m :

$$\begin{aligned} \text{vars}(x) &= \{x\}, & \text{for } x \in \text{Var}, \\ \text{vars}(X) &= \{\}, & \text{for } X \in \text{Val}, \\ \text{vars}(m_1, m_2) &= \text{vars}(m_1) \cup \text{vars}(m_2), \\ \text{vars}(\{m\}_k) &= \text{vars}(m) \cup \text{vars}(k). \end{aligned}$$

¹ $\text{prog}(i)$ represents the i th element of the sequence prog .

We write $newVars(m)$ for the new variables that could be bound as the result of receiving message m ; note that the encryption key cannot be so bound:

$$\begin{aligned} newVars(x) &= \{x\}, & \text{for } x \in Var, \\ newVars(X) &= \{\}, & \text{for } X \in Val, \\ newVars(m_1, m_2) &= newVars(m_1) \cup newVars(m_2), \\ newVars(\{m\}_k) &= newVars(m). \end{aligned}$$

We lift this to event templates:

$$\begin{aligned} newVars(new\ x) &= \{x\}, \\ newVars(newpair(x, y)) &= \{x, y\}, \\ newVars(send\ m) &= \{\}, \\ newVars(receive\ m) &= newVars(m). \end{aligned}$$

We lift these to programs in the obvious way.

3.3.2 Bindings

Part of the local state of each honest agent will record the values of variables. We model this by a partial mapping, or *binding*:

$$Binding \hat{=} Var \leftrightarrow Val.$$

We will write ρ for a typical binding.

Note that the binding is a *partial* function. We will occasionally want to deal with a situation where a public key pk is in the domain of the binding ρ but the corresponding secret key sk isn't, and where we want to talk about the value of sk — or more accurately, we want to talk about the inverse value of the value of pk ; in such cases, we will abuse notation and just write $\rho(sk)$; more generally, if $k \notin \text{dom } \rho$, $k^{-1} \in \text{dom } \rho$, then we write $\rho(k)$ as shorthand for $(\rho(k^{-1}_{var}))^{-1}_{val}$. Further, we will sometimes write $\rho(x) = \perp$ as shorthand for $x \notin \text{dom } \rho$ and if x^{-1} is defined that $x^{-1} \notin \text{dom } \rho$, i.e. to indicate that x is not bound in ρ .

The operational semantics we give, below, will ensure that variables in the bindings are well-typed, in the sense that if $\rho(x) = X$ then $type_{var}(x) = type_{val}(X)$.

If ρ is a binding and m a message template, all of whose variables are bound in ρ , then we write $m[\rho]$ for the corresponding actual message, where each variable x is replaced by $\rho(x)$. Similarly, if P is a predicate, we write $P[\rho]$ for the result of the corresponding substitution.

3.3.3 Local states

We will represent the local state of an agent by a triple $(prog, \rho, id) : Prog \times Binding \times Var$, where $prog$ is the remaining sequence of event templates it needs to perform, ρ is a binding, and $id \in \text{dom } \rho$ is a distinguished variable that represents the local agent's identity. Given a local state s , we will write " $s.prog$ ", " $s.\rho$ " and $s.id$ to refer to the three components. We use the convention that the selection operator "." binds tighter than all other operators, including function application, so for example $s.\rho(x) = (s.\rho)(x)$. Note that $s.id$ is the *variable* that represents the agent's identity, not the value of that identity, which is stored in $s.\rho(s.id)$.

3.3.4 Operational semantics

We now give operational semantics for local states. We consider four types of *events* performed by protocol participants, analogous to event templates:

send The event $\text{send } M$ represents the local agent sending actual message M ;

receive The event $\text{receive } M$ represents the local agent receiving actual message M ;

new The event $\text{new } X$ represents the local agent freshly generating the value X ;

newpair The event $\text{newpair}(X, Y)$ represents the local agent freshly generating the asymmetric key pair (X, Y) .

Formally we define events according to the grammar:

$$\text{Event} ::= \text{send } Msg \mid \text{receive } Msg \mid \text{new } Val \mid \text{newpair}(Val, Val).$$

We write $s \xrightarrow{E} s'$ to mean that from local state s , the event E can be performed to reach local state s' . The \longrightarrow relation is defined as follows:

- If the next event template in the program is of the form $\text{new } x$, then the agent can perform the event $\text{new } X$ for a value X of the same type as x ; the binding is updated to bind x to X :

$$\begin{aligned} & (\langle \text{new } x \rangle \wedge \text{prog}, \rho, id) \xrightarrow{\text{new } X} (\text{prog}, \rho \oplus \{x \mapsto X\}, id), \\ & \text{provided } \text{type}_{val}(X) = \text{type}_{var}(x), x \notin \text{dom } _^{-1} \vee x^{-1} = x. \end{aligned}$$

We will ensure later that the value X generated is fresh.

- The semantics of newpair is very similar, except two values, which must be inverses, are involved:

$$\begin{aligned} & (\langle \text{newpair}(x, y) \rangle \wedge \text{prog}, \rho, id) \xrightarrow{\text{newpair}(X, Y)} \\ & (\text{prog}, \rho \oplus \{x \mapsto X, y \mapsto Y\}, id), \\ & \text{provided } \text{type}_{val}(X) = \text{type}_{var}(x), \text{type}_{val}(Y) = \text{type}_{var}(y), \\ & X^{-1} = Y. \end{aligned}$$

- If the next event template in the program is of the form $\text{send } am$ (where am is an abstract message), then the agent can perform the event $\text{send } m[\rho]$, provided m is a concrete template corresponding to am (i.e. in the semantics of am):

$$\begin{aligned} & (\langle \text{send } am \rangle \wedge \text{prog}, \rho, id) \xrightarrow{\text{send } m[\rho]} (\text{prog}, \rho, id), \\ & \text{provided } m \in \llbracket am \rrbracket_{\Pi}. \end{aligned}$$

Note that there are two steps in the instantiation of am : the choice (at design time) of concrete template; and the substitution (at run time) of each free variable with the appropriate value from the binding. In the case that am is already a concrete template, the semantics simplifies to

$$\begin{aligned} & (\langle \text{send } m \rangle \wedge \text{prog}, \rho, id) \xrightarrow{\text{send } m[\rho]} (\text{prog}, \rho, id), \\ & \text{for } m \in \text{Template}. \end{aligned}$$

- If the next event template in the program is of the form $\text{receive } am$, then the agent can perform the event $\text{receive } m[\rho']$ provided m is a concrete template corresponding to am , and update its binding to ρ' for a suitable binding ρ' . More precisely, the new binding must: (1) extend ρ by giving new values to the variables received in m ; (2) respect the types of variables; (3) respect inverses:

$$\begin{aligned} & (\langle \text{receive } am \rangle \wedge \text{prog}, \rho, id) \xrightarrow{\text{receive } m[\rho']} (es, \rho', id), \\ & \text{provided } m \in \llbracket am \rrbracket_{\Pi}, \\ & \rho' \supseteq \rho, \text{dom } \rho' = \text{dom } \rho \cup \text{newVars}(m), \\ & \forall x \in \text{dom } \rho' \bullet \text{type}_{var}(x) = \text{type}_{val}(\rho(x)), \\ & \forall x, y \in \text{dom } \rho' \bullet x^{-1} = y \Rightarrow \rho'(x)^{-1} = \rho'(y)^{-1}. \end{aligned}$$

Note, in particular, that if a variable has had a value bound to it already, and a message using that variable is received, then only the previous value will be accepted: this means that the value received must be checked against the value stored.

Note that the value of the identity variable id is preserved by the semantics.

We adopt standard shorthands concerning the transition relation; for example, we write $s \longrightarrow s'$ for $\exists E \in \text{Event} \bullet s \xrightarrow{E} s'$.

The following lemma captures some properties of the operational semantics.

Lemma 1

1. If $(\langle e \rangle \frown prog, \rho, id) \xrightarrow{E} (prog, \rho', id')$ then $\rho \subseteq \rho' \wedge \text{dom } \rho' = \text{dom } \rho \cup \text{newVars}(e)$.
2. If $(prog \frown prog', \rho, id) \longrightarrow^* (prog', \rho', id')$ then $\rho \subseteq \rho' \wedge \text{dom } \rho' = \text{dom } \rho \cup \text{newVars}(prog)$.

Proof: (sketch)

1. This follows from a straightforward case analysis.
2. This follows from the previous case by a straightforward induction. □

We say that a binding is well-typed if the type of every variable agrees with the type of the value stored in it, and variables that represent inverses of one another store values that are inverses of one another:

$$\begin{aligned} \text{wellTyped}(\rho) \hat{=} & \forall x \in \text{dom } \rho \bullet \text{type}_{\text{var}}(x) = \text{type}_{\text{val}}(\rho(x)) \\ & \wedge \\ & \forall x, y \in \text{dom } \rho \bullet x^{-1} = y \Rightarrow \rho(x)^{-1} = \rho(y). \end{aligned}$$

The property of being well-typed is preserved by the operational semantics:

Lemma 2

$$\text{wellTyped}(\rho) \wedge (prog, \rho, id) \xrightarrow{E} (prog', \rho', id) \Rightarrow \text{wellTyped}(\rho').$$

3.4 Feasible protocols

Recall that a concrete program contains no abstract messages. In this section, we consider the circumstances under which a concrete program is feasible, in the sense that every variable is bound before it is used. We will use the initial binding to store the initial knowledge of the agent in question, i.e. the initial binding will contain those values that it needs to run the protocol, bound to suitable variables. We make this precise below.

We define a predicate canUnpack such that $\text{canUnpack}(xs, ms)$ means that an agent who has appropriate values for the set of variables xs can unpack the set of templates ms so as to obtain all the variables within it. canUnpack is defined to be the smallest predicate such that:

$$\begin{aligned} & \text{canUnpack}(xs, \{\}), \\ & \text{canUnpack}(xs, \{v\} \cup ms) \Leftarrow \text{canUnpack}(xs \cup \{v\}, ms), \\ & \quad \text{for } v \in \text{Var}, \\ & \text{canUnpack}(xs, \{(m_1, m_2)\} \cup ms) \Leftarrow \text{canUnpack}(xs, \{m_1, m_2\} \cup ms), \\ & \text{canUnpack}(xs, \{\{m\}_k\} \cup ms) \Leftarrow k^{-1} \in xs \wedge \\ & \quad \text{canUnpack}(xs, \{m\} \cup ms). \end{aligned}$$

Definition 1 We define LocalState to be the set of all triples $(prog, \rho, id) : \text{Prog} \times \text{Binding} \times \text{Var}$ such that:

1. $prog$ uses no abstract messages.
2. The variable id , representing the agent's identity, is bound:

$$id \in \text{dom } \rho.$$

3. Whenever the agent is supposed to send a message described by template m , the agent is able to produce the message from his initial knowledge ($\text{dom } \rho$) and the variables bound subsequently ($\text{newVars}(prog')$ below):

$$\forall prog' \wedge \langle \text{send } m \rangle \leq prog \bullet \text{vars}(m) \subseteq \text{dom } \rho \cup \text{newVars}(prog').$$

4. Whenever the agent is supposed to receive a message described by template m , the agent is able to unpack the message from his initial knowledge and the variables bound subsequently:

$$\forall prog' \wedge \langle \text{receive } m \rangle \leq prog \bullet \\ \text{canUnpack}(\text{dom } \rho \cup \text{newVars}(prog'), \{m\}).$$

5. Whenever the agent is supposed to generate a new value for a variable, that variable it not already bound:

$$\forall prog' \wedge \langle \text{new } x \rangle \leq prog \bullet x \notin \text{dom } \rho \cup \text{newVars}(prog') \wedge \\ \forall prog' \wedge \langle \text{newpair}(x, y) \rangle \leq prog \bullet x, y \notin \text{dom } \rho \cup \text{newVars}(prog').$$

We say that a protocol is *feasible* if it is a member of *LocalState*. The goal of a protocol development will always be to end up with a feasible protocol.

The following lemma shows that being an element of *LocalState* is preserved by the operational semantics.

Lemma 3 If $(prog \wedge prog', \rho, id) \in \text{LocalState}$ and $(prog \wedge prog', \rho, id) \longrightarrow^* (prog', \rho', id)$, then $(prog', \rho', id) \in \text{LocalState}$.

Proof: (sketch) We need to check each of the conditions from Definition 1 in turn.

Condition 1 is trivially maintained. Condition 2 is trivial because id does not change, and the binding can only increase (Lemma 1).

For condition 3, suppose $prog = prog_1 \wedge prog_2 \wedge \langle \text{send } m \rangle \wedge prog_3$ and

$$(prog, \rho, id) \longrightarrow^* (prog_2 \wedge \langle \text{send } m \rangle \wedge prog_3, \rho', id).$$

Then

$$\begin{aligned} & \text{vars}(m) \\ \subseteq & \left\langle \begin{array}{l} \text{the initial state is in } \text{LocalState} \\ \text{dom } \rho \cup \text{newVars}(prog_1 \wedge prog_2) \end{array} \right\rangle \\ = & (\text{dom } \rho \cup \text{newVars}(prog_1)) \cup \text{newVars}(prog_2) \\ = & \left\langle \text{Lemma 1} \right\rangle \\ & \text{dom } \rho' \cup \text{newVars}(prog_2), \end{aligned}$$

as required.

Conditions 4 and 5 are very similar. □

3.5 The intruder

We now discuss how we model the intruder. We simply record the set of messages that the intruder knew initially or has seen subsequently. We capture this formally when we discuss global states, below.

We will need to capture the way the intruder can produce new messages from messages he already knows. We write $B \vdash M$ is the message M can be obtained from the set of messages B by the intruder. The relation \vdash is defined by the following five rules.

member $M \in B \Rightarrow B \vdash M$;

pair $B \vdash M_1 \wedge B \vdash M_2 \Rightarrow B \vdash (M_1, M_2)$;

split $B \vdash (M_1, M_2) \Rightarrow B \vdash M_1 \wedge B \vdash M_2$;

encrypt $B \vdash M \wedge B \vdash K \Rightarrow B \vdash \{M\}_K$;

decrypt $B \vdash \{M\}_K \wedge B \vdash K^{-1} \Rightarrow B \vdash M$.

Below we write “*intruder*” for the identity of the intruder².

3.6 Global states

A *global state* is a collection of local states of honest agents, together with the state of the intruder. We model this by a function σ with domain $0..n$ for some n : $\sigma(0)$ will represent the state of the intruder; $\sigma(1), \dots, \sigma(n)$ will represent the states of the honest agents. Formally:

$$\begin{aligned} \text{GlobalState} \cong \{ \sigma : \mathbb{N} \leftrightarrow ((\text{Prog} \times \text{Binding} \times \text{Var}) \cup \mathbb{P} \text{Message}) \mid \\ \exists n : \mathbb{N} \bullet \text{dom } \sigma = 0..n \wedge \sigma(0) \in \mathbb{P} \text{Message} \wedge \\ \forall i \in 1..n \bullet \sigma(i) \in \text{Prog} \times \text{Binding} \times \text{Var} \}. \end{aligned}$$

Note that several different nodes may have the same values for their identity variables, representing that a particular honest agent may be running several different programs (or roles) simultaneously.

Below we will write σ_0 for the initial global state, and n for the number of honest nodes.

We assume that the intruder’s identity *intruder* is distinct from the identities of all the other nodes:

$$\forall i \in 1..n \bullet \sigma_0(i).id \neq \text{intruder}.$$

In most annotations, we will assume that the programs of different nodes are consistent in the sense that they use the same variable name for variables that are intended to be equal. For example, if an agent has a send event $\text{send } m$ that is intended to be received in the event $\text{receive } m'$, then m and m' will be defined using the same variables, so will in fact be syntactically equal. Further, if two nodes have the same identity variables, they will be running the same program: $\sigma_0(i).id = \sigma_0(j).id \Rightarrow \sigma_0(i).prog = \sigma_0(j).prog$.

3.6.1 Operational semantics

We now give operational semantics for global states. We write $\sigma \xrightarrow{i:E} \sigma'$ to represent that from global state σ , node i can perform the event E causing the global state to evolve to σ' . The operational semantics is defined by the four rules below. We arrange for all communications to go via the intruder, rather than having honest agents synchronise directly; so a send event by an honest agent simply causes the corresponding message to be added to the intruder’s knowledge; and a receive event can happen provided the intruder can produce the corresponding message.

² It is straightforward to extend the model so as to give the intruder multiple identities, or equivalently to allow several intruders with different identities to work together.

We consider first new X events. We need to specify that the value X that results from this event really is a new value; this is captured by the following predicate:

$$\begin{aligned} isNew(X)(\sigma) \triangleq & \forall M \in \sigma(0) \bullet X \not\leq M \\ & \wedge \\ & \forall i > 0; y \in \text{dom } \sigma(i). \rho \bullet X \not\leq \sigma(i). \rho(y). \end{aligned}$$

The event $i : \text{new } X$ can occur if: (1) the node i can do the corresponding new X event; (2) no other node changes its state; and (3) the value X is new:

$$\frac{\begin{array}{l} \sigma(i) \xrightarrow{\text{new } X} \sigma'(i) \\ \forall j \in 0..n \bullet j \neq i \Rightarrow \sigma(j) = \sigma'(j) \\ isNew(X)(\sigma) \end{array}}{\sigma \xrightarrow{i:\text{new } X} \sigma'} \quad [i > 0]$$

The semantics of newpair events is very similar:

$$\frac{\begin{array}{l} \sigma(i) \xrightarrow{\text{newpair}(X,Y)} \sigma'(i) \\ \forall j \in 0..n \bullet j \neq i \Rightarrow \sigma(j) = \sigma'(j) \\ isNew(X)(\sigma) \wedge isNew(Y)(\sigma) \end{array}}{\sigma \xrightarrow{i:\text{newpair}(X,Y)} \sigma'} \quad [i > 0]$$

The event $i : \text{send } M$ can occur if: (1) the node i can do the corresponding send M event; (2) M is added to the intruder's knowledge; and (3) no other node changes its state:

$$\frac{\begin{array}{l} \sigma(i) \xrightarrow{\text{send } M} \sigma'(i) \\ \sigma'(0) = \sigma(0) \cup \{M\} \\ \forall j \in 1..n \bullet j \neq i \Rightarrow \sigma(j) = \sigma'(j) \end{array}}{\sigma \xrightarrow{i:\text{send } M} \sigma'}$$

The event $i : \text{receive } M$ can occur if: (1) the node i can do the corresponding receive M event; (2) the intruder is able to produce the message M to send it (possibly faked) to i ; and (3) no other node changes its state:

$$\frac{\begin{array}{l} \sigma(i) \xrightarrow{\text{receive } M} \sigma'(i) \\ \sigma(0) \vdash M \\ \forall j \in 0..n \bullet j \neq i \Rightarrow \sigma(j) = \sigma'(j) \end{array}}{\sigma \xrightarrow{i:\text{receive } M} \sigma'}$$

3.7 Protocol semantics

A *system trace* is an alternating sequence of the form

$$\langle \sigma_0, i_1:E_1, \sigma_1, i_2:E_2, \sigma_2, \dots, \sigma_n \rangle,$$

where each σ_i is a global state, each i_j is a node index, and each E_j is an event, such that

$$\sigma_0 \xrightarrow{i_1:E_1} \sigma_1 \xrightarrow{i_2:E_2} \sigma_2 \dots \sigma_n.$$

This trace represents a protocol run in which the initial state is σ_0 , then event $i_1 : E_1$ occurs and the state evolves into σ_1 , and so on.

The semantics of a protocol Π , written $traces(\Pi)$, is then the set of all traces that can be observed of it.

We write

$$\sigma \langle i_1 : E_1, \dots, i_n : E_n \rangle \Longrightarrow \sigma'$$

to indicate

$$\sigma \xrightarrow{i_1 : E_1} \dots \xrightarrow{i_n : E_n} \sigma'.$$

We write $\sigma \Longrightarrow \sigma'$ for $\exists tr \bullet \sigma \xrightarrow{tr} \sigma'$. We define $States(\Pi)$ to be all the reachable states, i.e. states σ such that $\sigma_0 \Longrightarrow \sigma$.

If $tr = \langle i_1 : E_1, i_2 : E_2, \dots, i_n : E_n \rangle$ is a sequence of events, then we write $tr \upharpoonright i$ for the restriction of tr to the events performed by node i :

$$\begin{aligned} \langle \rangle \upharpoonright i &= \langle \rangle, \\ (\langle j : E \rangle \wedge tr) \upharpoonright i &= \langle E \rangle \wedge (tr \upharpoonright i) \text{ if } j = i, \\ &tr \upharpoonright i \text{ otherwise.} \end{aligned}$$

4 Annotations

In this section we consider annotations in more detail. In Section 4.1 we formally define the meaning of an annotation. In Section 4.2 we give some structural annotation rules. In Section 4.3 we give formal definitions of the specification macros we have used, together with a few annotation rules using them. Finally in Section 4.4 we give an annotation rule for the new x construct.

4.1 Correctness of annotations

Consider an assertion P that is intended to hold for a node $i > 0$ in some state σ . The free variables within P refer to the values within i 's binding $(\sigma(i), \rho)$ and so need to be substituted with those values; the resulting predicate is then interpreted with respect to σ : $P[\sigma(i), \rho](\sigma)$. We abbreviate this to $P(\sigma)[i]$, pronounced “ P in σ for i ”:

$$P(\sigma)[i] \hat{=} P[\sigma(i), \rho](\sigma).$$

Note that we need to be careful with the substitution, for not every free occurrence of a variable x within P refers to i 's value for x : some may refer to a different node's value for x . In such cases, we define the substitution to “do the right thing”; we make this more precise when we discuss relevant macros, below, specifically the *session* macro.

Suppose $\sigma_0(i).prog = es_0 \hat{\wedge} es_1$; then to say that i can be sure that predicate P holds after es_0 means that for every state σ where i has remaining program es_1 , it must be the case that P holds in σ for i :

$$\forall \sigma \in States(\Pi) \mid \sigma(i).prog = es_1 \bullet P(\sigma)[i].$$

We now formally define the annotation $a : \{pre\} es \{post\}$. Roughly speaking, we want to say that the annotation is correct if $post$ holds just after es is performed, assuming pre always holds just before es . Recall, however, that the annotation may use abstract messages within es , whereas the actual system will use concrete messages; we therefore consider all executions resulting from event templates that are refinements of es . More precisely, if $\sigma(i).id = a$ and $\sigma_0(i).prog = es_0 \hat{\wedge} es' \hat{\wedge} es_1$ where $es' \sqsupseteq es$, then the annotation is correct if $post$ always holds after $es_0 \hat{\wedge} es'$, assuming pre always holds after es_0 .

$$\begin{aligned} a : \{pre\} es \{post\} &\hat{=} \\ &\forall i \in 1..n \mid \sigma_0(i).id = a \bullet \\ &\forall es_0, es_1, es' \mid \sigma_0(i).prog = es_0 \hat{\wedge} es' \hat{\wedge} es_1 \wedge es' \sqsupseteq es \bullet \\ &\quad (\forall \sigma \in States(\Pi) \mid \sigma(i).prog = es' \hat{\wedge} es_1 \bullet pre(\sigma)[i]) \\ &\quad \Rightarrow \\ &\quad (\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i]). \end{aligned}$$

4.2 Structural annotation rules

We now prove some of the structural annotation rules that we used earlier. Within these rules, we blur the distinction between single events and sequences of events.

Annotation rule 1 (Strengthen precondition)

$$\frac{a : \{pre\} e \{post\} \quad pre' \Rightarrow pre}{a : \{pre'\} e \{post\}}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge e' \sqsupseteq e \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet pre'(\sigma)[i]. \end{aligned}$$

Then by the second hypothesis,

$$\forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet pre(\sigma)[i].$$

Then by the first hypothesis

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i],$$

and so $a : \{pre'\}e\{post\}$ as required. \square

Annotation rule 2 (Weaken postcondition)

$$\frac{a : \{pre\}e\{post\} \quad post \Rightarrow post'}{a : \{pre\}e\{post'\}}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge e' \sqsupseteq e \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet pre(\sigma)[i]. \end{aligned}$$

Then by the first hypothesis,

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i].$$

Hence, by the second hypothesis,

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post'(\sigma')[i].$$

And so $a : \{pre\}e\{post'\}$, as required. \square

Annotation rule 3 (Sequential composition)

$$\frac{a : \{pre\}e_1\{mid\} \quad a : \{mid\}e_2\{post\}}{a : \{pre\}e_1e_2\{post\}}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e'_1 \hat{\wedge} e'_2 \hat{\wedge} es_1 \wedge e'_1 \sqsupseteq e_1 \wedge e'_2 \sqsupseteq e_2 \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e'_1 \hat{\wedge} e'_2 \hat{\wedge} es_1 \bullet pre(\sigma)[i]. \end{aligned}$$

Then by the first hypothesis,

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = e'_2 \hat{\wedge} es_1 \bullet mid(\sigma')[i].$$

Hence by the second hypothesis,

$$\forall \sigma'' \in States(\Pi) \mid \sigma''(i).prog = es_1 \bullet post(\sigma'')[i].$$

Hence $a : \{pre\}e_1e_2\{post\}$ as required. \square

We also give a rule concerning conjunctions of postconditions; this rule allows us to verify conjuncts of a postcondition separately.

Annotation rule 4 (Conjunction of postconditions)

$$\frac{a : \{pre\}e\{post_1\} \quad a : \{pre\}e\{post_2\}}{a : \{pre\}e\{post_1 \wedge post_2\}}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge e' \sqsupseteq e \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet pre(\sigma)[i]. \end{aligned}$$

Then by the first hypothesis

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post_1(\sigma')[i],$$

and by the second hypothesis,

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post_2(\sigma')[i].$$

Hence

$$\forall \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet (post_1 \wedge post_2)(\sigma')[i],$$

and so $a : \{pre\}e\{post_1 \wedge post_2\}$. □

4.3 Annotation macros

In this section we give semantics to the annotation macros that we introduced informally earlier.

4.3.1 *knows*

The macro $knows(x)$ returns the set of participants who know the value of x . Recall that assertions are interpreted with respect to a particular state, say state σ , and a particular node, say node i ; therefore the value of x in question is $\sigma(i).\rho(x)$. This value is obtained via the substitution:

$$knows(x)(\sigma)[i] = knows(x)[\sigma(i).\rho](\sigma) = knows(\sigma(i).\rho(x))(\sigma).$$

We therefore define the meaning of $knows$ with respect to a *value* X (as opposed to a variable). If $x \notin \text{dom } \sigma(i).\rho$, $x \in \text{dom } \sigma(i).\rho$ then recall that $\sigma(i).\rho(x)$ is shorthand for $(\sigma(i).\rho(x^{-1}))^{-1}$, so in this case $knows(x)$ represents the agents who know the decrypting key corresponding to encryptions using the value of x^{-1} .

For later convenience, we start by defining a function $knows_{id}(X)$ that gives the set of node identifiers where X is known. X is known by honest node i if $\sigma(i).\rho(y) = X$ for some y ; X is known by the intruder if $\sigma(0) \vdash X$.

$$\begin{aligned} knows_{id}(X)(\sigma) \hat{=} \{i \mid i \in 1..n \wedge \exists y \bullet \sigma(i).\rho(y) = X\} \\ \cup \\ \text{(if } \sigma(0) \vdash X \text{ then } \{0\} \text{ else } \{\}). \end{aligned}$$

We now define the function $knows(X)$ which returns the corresponding set of agent identities:

$$\begin{aligned} knows(X)(\sigma) \hat{=} \\ \{\text{if } i > 0 \text{ then } \sigma(i).\rho(\sigma(i).id) \text{ else } intruder \mid i \in knows_{id}(X)(\sigma)\}. \end{aligned}$$

Equivalently, $knows(X)$ could have been defined by

$$knows(X)(\sigma) \hat{=} \frac{\{\sigma(i).\rho(\sigma(i).id) \mid i \in 1..n \wedge \exists y \bullet \sigma(i).\rho(y) = X\}}{\cup} \\ (\text{if } \sigma(0) \vdash X \text{ then } \{intruder\} \text{ else } \{\}).$$

Note that the value of $knows(X)$ cannot, in general, be relied upon to stay the same from one state to another, even if the agent currently being considered does not perform any events: messages sent elsewhere may cause new agents to learn X . However, the value of $knows(X)$ can only increase as an execution progresses.

The following annotation rule shows that $knows(x)$ does not change as a result of a receive event, provided the recipient already knows the value.

Annotation rule 5 For every message m , and for every set s of variables representing identities such that $a \in s$:

$$a : \{knows(x) = s\} \text{ receive } m \{knows(x) = s\}.$$

Proof: Suppose

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge e' \sqsupseteq \text{receive } m \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet (knows(x) = s)(\sigma)[i].$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the event corresponding to e' . Let

$$X = \sigma(i).\rho(x), \quad A = \sigma(i).\rho(a), \quad S = \{\sigma(i).\rho(b) \mid b \in s\}.$$

Let $\hat{\sigma} = \sigma' \oplus \{i \mapsto \sigma(i)\}$. Then $\hat{\sigma} \in States(\Pi)$, reachable by the same trace that led to σ' but without the event corresponding to e' . Also, $\hat{\sigma}(i).prog = \sigma(i).prog = e' \hat{\wedge} es_1$, so by the assumption corresponding to the precondition, $(knows(x) = s)(\hat{\sigma})[i]$, i.e.:

$$knows(X)(\hat{\sigma}) = S.$$

For every $j \neq i$, $j \in knows_{id}(X)(\sigma') \Leftrightarrow j \in knows_{id}(X)(\hat{\sigma})$, because $\sigma'(j) = \hat{\sigma}(j)$, and so for every $B \neq A$, $B \in knows(X)(\sigma') \Leftrightarrow B \in knows(X)(\hat{\sigma})$. And $A \in knows(X)(\sigma')$ and $A \in knows(X)(\hat{\sigma})$ by the precondition and the assumption that $a \in s$. Hence $knows(X)(\sigma') = knows(X)(\hat{\sigma}) = S$, i.e. $(knows(x) = s)(\sigma')$, as required. \square

4.3.2 session

If B is an honest agent then the notation

$$session(b \rightsquigarrow B; x_1 \rightsquigarrow X_1, \dots, x_k \rightsquigarrow X_k)(\sigma)$$

means that for some node i , the variable representing the agent's identity is b , that b is bound to B , and each x_j is bound to X_j . If B is dishonest then the notation means that B knows each of the X_j : a dishonest agent is not forced to bind values to variables in any predictable way.

$$session(b \rightsquigarrow B; x_1 \rightsquigarrow X_1, \dots, x_k \rightsquigarrow X_k)(\sigma) \hat{=} \\ \exists i > 0 \bullet \sigma(i).id = b \wedge \sigma(i).\rho(b) = B \wedge \forall j \in 1..k \bullet \sigma(i).\rho(x_j) = X_k \\ \vee \\ B = intruder \wedge \forall j \in 1..k \bullet \sigma(0) \vdash X_j.$$

Recall that an assertion P is interpreted with respect to a particular node, say node i , via the substitution $P(\sigma)[i] = P[\sigma(i).\rho](\sigma)$. In the case of the *session* macro, we define this substitution

to be performed only on variables on the right hand side of \rightsquigarrow symbols, not those on the left hand side. For example,

$$\begin{aligned} \text{session}(b \rightsquigarrow c; x \rightsquigarrow y)(\sigma)[i] = \\ \text{session}(b \rightsquigarrow \sigma(i).\rho(c); x \rightsquigarrow \sigma(i).\rho(y))(\sigma), \end{aligned}$$

i.e. the other node's b variable is bound to the value of node i 's c variable, and the other node's x variable is bound to the value of node i 's y variable.

Often the value of a variable, x say, in a local agent's state, say B 's state, will match the value of the variable of the same name in the current scope; if the current annotation is from the point of view of agent A , then this means that A 's value of x is the same as B 's value of x . In such cases we simplify the binding " $x \rightsquigarrow x$ " to just " x ", representing that from A 's point of view, B has x bound to the correct variable. We adopt the same convention with the identity variable. For example,

$$\begin{aligned} \text{session}(b; x)(\sigma)[i] &\equiv \text{session}(b \rightsquigarrow b; x \rightsquigarrow x)(\sigma)[i] \\ &\equiv \text{session}(b \rightsquigarrow \sigma(i).\rho(b); x \rightsquigarrow \sigma(i).\rho(x))(\sigma). \end{aligned}$$

The following lemma relates the *session* and *knows* macros.

Lemma 4

$$\text{session}(a \rightsquigarrow b; x \rightsquigarrow y)(\sigma)[i] \Rightarrow \sigma(i).\rho(b) \in \text{knows}(\sigma(i).\rho(y))(\sigma).$$

4.3.3 honest

The predicate *honest*(X) asserts that the set of participants in X are honest in the sense that they do not deviate from the protocol definition:

$$\text{honest}(X) \hat{=} \text{intruder} \notin X.$$

Note that if *honest*(X) holds, then it will hold throughout an execution as an invariant.

4.3.4 Always

If P is a predicate, then $\Box P$ (pronounced "always P ") represents that P holds in this and all subsequent states:

$$\Box P(\sigma) \hat{=} \forall \sigma', tr \mid \sigma \xrightarrow{tr} \sigma' \bullet P(\sigma').$$

Note that $\Box P \Rightarrow P$, because $\sigma \xrightarrow{\langle \rangle} \sigma$. Note also that

$$\begin{aligned} &(\Box P)(\sigma)[i] \\ &\equiv (\Box(P[\sigma(i).\rho]))(\sigma) \\ &\equiv \forall \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P[\sigma(i).\rho](\sigma') \\ &\equiv \left\langle \begin{array}{l} P \text{ can refer only to variables bound in } \sigma(i).\rho, \\ \text{and } \sigma'(i).\rho \text{ agrees on those variables (Lemma 1)} \end{array} \right\rangle \\ &\quad \forall \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P[\sigma'(i).\rho](\sigma') \\ &\equiv \forall \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P(\sigma')[i]. \end{aligned}$$

The following rule allows assertions of the form $\Box P$ to be carried forward through annotations.

Annotation rule 6 For all events e and predicates P

$$a : \{ \Box P \} e \{ \Box P \}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge e' \sqsupseteq e \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet (\Box P)(\sigma)[i]. \end{aligned}$$

Suppose σ' is such that $\sigma'(i).prog = es_1$ and suppose $\sigma' \Longrightarrow \sigma''$. Let σ be the state immediately before the event corresponding to e' ; then $\sigma(i).prog = e' \hat{\wedge} es_1$ and $\sigma \Longrightarrow \sigma'$. But then $\sigma \Longrightarrow \sigma''$ and so $P(\sigma'')[i]$ because $(\Box P)(\sigma)[i]$. Hence $(\Box P)(\sigma')[i]$. \square

4.4 new x

In this section we verify the proof rule for new x given earlier.

It is useful to define a macro i holds X to specify that node i has X as a submessage of one of the messages it holds:

$$\begin{aligned} (i \text{ holds } X)(\sigma) \hat{=} i = 0 \wedge \exists M \in \sigma(i) \bullet X \preceq M \\ \vee \\ i > 0 \wedge \exists y \bullet X \preceq \sigma(i).\rho(y). \end{aligned}$$

Note that

$$\begin{aligned} isNew(X)(\sigma) \Leftrightarrow \forall i \bullet \neg (i \text{ holds } X)(\sigma), \\ i \in knows_{id}(X)(\sigma) \Rightarrow (i \text{ holds } X)(\sigma). \end{aligned}$$

The following lemma shows how i can acquire X :

Lemma 5 If

$$\sigma \xrightarrow{j:E} \sigma' \wedge \neg (i \text{ holds } X)(\sigma) \wedge (i \text{ holds } X)(\sigma')$$

then

$$\begin{aligned} (E = \text{new } X \vee \exists Y \bullet E = \text{newpair}(X, Y) \vee E = \text{newpair}(Y, X)) \wedge i = j \\ \vee \\ \exists M \bullet E = \text{send } M \wedge i = 0 \wedge X \preceq M \\ \vee \\ \exists M \bullet E = \text{receive } M \wedge i = j \wedge X \preceq M. \end{aligned}$$

Proof: Direct from the operational semantics. \square

Annotation rule 7 (New) If pre refers only to state variables, and x is not free in pre then

$$a : \{pre\} \text{ new } x \{knows(x) = \{a\} \wedge pre\}$$

Note that the restrictions on pre are necessary to prevent preconditions such as $\# \rho = 3$, which would not be preserved by the creation of a new variable within ρ .

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge e' \sqsupseteq \text{new } x \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \hat{\wedge} es_1 \bullet pre(\sigma)[i]. \end{aligned}$$

Then $e' = \text{new } x$. Suppose σ' is such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the new x event. Then $pre(\sigma)[i]$ and $\sigma'(i).\rho = \sigma'(i) \oplus \{x \mapsto X\}$ for some fresh X . Let σ''

be the global state immediately after the transition, which might not be the same as σ' at nodes other than i ; then:

$$\begin{aligned} \sigma \xrightarrow{i:\text{new } X} \sigma'' \longrightarrow^* \sigma' \wedge \sigma''(i) = \sigma'(i) \wedge \\ \text{isNew}(X)(\sigma) \wedge \forall j \neq i \bullet \sigma(j) = \sigma''(j). \end{aligned}$$

We consider the two conjuncts of the postcondition separately. For the first conjunct, we need to show

$$(\text{knows}(x) = \{a\})(\sigma')[i] \equiv \text{knows}(X)(\sigma') = \{A\},$$

where $A = \sigma(i).\rho(a)$. Clearly $(\text{knows}(X) = \{A\})(\sigma'')$ because X is fresh:

$$\begin{aligned} \text{isNew}(X)(\sigma) &\equiv \forall j \bullet \neg (j \text{ holds } X)(\sigma) \\ &\Rightarrow \forall j \neq i \bullet \neg (j \text{ holds } X)(\sigma''). \end{aligned}$$

But node i sends no messages between σ'' and σ' , and so

$$\forall j \neq i \bullet \neg (j \text{ holds } X)(\sigma')$$

from Lemma 5 and a simple case analysis. Hence $\text{knows}_{id}(X)(\sigma') = \{i\}$ and so $\text{knows}(X)(\sigma') = \{A\}$.

For the second conjunct, we need to show $\text{pre}(\sigma')[i]$. Let $\hat{\sigma} = \sigma' \oplus \{i \mapsto \sigma(i)\}$. Then it is clear that $\hat{\sigma} \in \text{States}(\Pi)$, reachable via the same trace that reached σ' except excluding the $i : \text{new } X$ event. Further, $\hat{\sigma}(i).\text{prog} = \text{new } x \wedge \text{es}_1$. Hence by the hypothesis of the rule, $\text{pre}(\hat{\sigma})[i]$. But

$$\begin{aligned} &\text{pre}(\hat{\sigma})[i] \\ &\equiv \langle \text{definition} \rangle \\ &\quad \text{pre}[\hat{\sigma}(i).\rho](\hat{\sigma}) \\ &\equiv \langle x \text{ not free in } \text{pre} \rangle \\ &\quad \text{pre}[\sigma'(i).\rho](\hat{\sigma}) \\ &\equiv \langle x \text{ not free in } \text{pre}, \text{ and } \text{pre} \text{ refers only to state variables} \rangle \\ &\quad \text{pre}[\sigma'(i).\rho](\sigma') \\ &\equiv \langle \text{definition} \rangle \\ &\quad \text{pre}(\sigma')[i]. \end{aligned}$$

□

We believe a similar rule holds for `newpair`; verifying it is left as future work.

5 Disjoint encryption

In this section we define the disjoint encryption property, and then prove two theorems that follow from it.

We start by extending the submessage relation to $Template \leftrightarrow EventTemplate$ in the obvious way:

$$\begin{aligned} m \preceq \text{send } m' &\Leftrightarrow m \preceq m', \\ m \preceq \text{receive } m' &\Leftrightarrow m \preceq m. \end{aligned}$$

We now capture the disjoint encryption assumption.

Definition 2 (Disjoint encryption) Suppose in the initial state σ_0 , the j_1 th message of the program at node i_1 and the j_2 th message of the program at node i_2 both contain encrypted submessages that have the same type:

$$\begin{aligned} \{m_1\}_{k_1} \preceq \sigma_0(i_1).prog(j_1) \wedge \{m_2\}_{k_2} \preceq \sigma_0(i_2).prog(j_2) \wedge \\ type(\{m_1\}_{k_1}) = type(\{m_2\}_{k_2}). \end{aligned}$$

Then either the two nodes are in fact running the same program and these two messages are in fact the same message of that program, or one is a send and the other a receive of the same template:

$$\begin{aligned} \sigma_0(i_1).prog = \sigma_0(i_2).prog \wedge j_1 = j_2 \\ \vee \\ \exists m \in Template \bullet \sigma_0(i_1).prog(j_1) = \text{send } m \wedge \sigma_0(i_2).prog(j_2) = \text{receive } m \\ \vee \\ \sigma_0(i_1).prog(j_1) = \text{receive } m \wedge \sigma_0(i_2).prog(j_2) = \text{send } m. \end{aligned}$$

In other words, the same encrypted component can appear only in corresponding events.

5.1 A theorem about agreement

In this section we prove a result that shows that, under certain circumstances, all occurrences of a value X in honest agents' states are bound to the same variable x :

We define X to be bound only to x in σ if all occurrences of X in honest agents' states are bound to x :

$$\begin{aligned} (X \text{ boundOnlyTo } x)(\sigma) \hat{=} \\ \forall i \in 1..n; y \in \text{dom } \sigma(i).\rho \mid X \preceq \sigma(i).\rho(y) \bullet \\ y = x \wedge \sigma(i).\rho(y) = X. \end{aligned}$$

We will need the following lemma which says that if the intruder can deduce a message containing $\{M\}_K$, then either he knows both M and K (so can perform the encryption), or he knows a message containing $\{M\}_K$:

Lemma 6 If $B \vdash M' \wedge \{M\}_K \preceq M'$ then $B \vdash M \wedge B \vdash K$ or $\{M\}_K \preceq B$.

Proof: Straightforward rule induction. □

We now prove the result alluded to above.

Theorem 1 Suppose:

1. The protocol satisfies the disjoint encryption property.
2. The intruder did not initially hold any message containing X :

$$X \not\preceq \sigma_0(0).$$

3. Any honest agent who held X initially had it bound to x :

$$(X \text{ boundOnlyTo } x)(\sigma_0).$$

4. Trace tr ends in state σ_1 , where the intruder does not know X :

$$\text{last } tr = \sigma_1 \wedge \sigma_1(0) \not\vdash X.$$

5. If X is generated in a new or newpair event, then it is generated to instantiate x :

$$\begin{aligned} & \forall tr' \wedge \langle \sigma, i : \text{new } X, \sigma' \rangle \leq tr \bullet \sigma(i).prog = \langle \text{new } x \rangle \wedge \sigma'(i).prog \\ & \wedge \\ & \forall tr' \wedge \langle \sigma, i : \text{newpair}(X, Y), \sigma' \rangle \leq tr \bullet \\ & \quad \exists y \bullet \sigma(i).prog = \langle \text{newpair}(x, y) \rangle \wedge \sigma'(i).prog \\ & \wedge \\ & \forall tr' \wedge \langle \sigma, i : \text{newpair}(Y, X), \sigma' \rangle \leq tr \bullet \\ & \quad \exists y \bullet \sigma(i).prog = \langle \text{newpair}(y, x) \rangle \wedge \sigma'(i).prog. \end{aligned}$$

Then any honest agent who holds X does so with it bound to the variable x :

$$(X \text{ boundOnlyTo } x)(\sigma_1). \tag{1}$$

Note that this theorem really concerns two quite different scenarios:

- If an honest agent does hold X initially (necessarily bound to x), then no new or newpair events for X can occur (because of the freshness condition on such events), and so assumption 5 holds vacuously.
- If no honest agent holds X initially, then it must be introduced (if at all) by a new x , newpair(x, y) or newpair(y, x) event. The theorem then gives a result about *all* values X that could be introduced for x . Note that in this case, assumptions 2, 3 and 5 are automatically satisfied.

Proof: Suppose, for a contradiction, that the result does not hold. Consider the shortest counterexample trace tr . By assumption 3, tr is not the trivial trace $\langle \sigma_0 \rangle$. So consider the last event of tr , and perform a case analysis:

- Case $i : \text{new } Y$. new events change bindings only for the node and variable in question. Hence the only way that equation (1) can be falsified by this event is if it is a new X event for a variable $y \neq x$. But this contradicts assumption 5.
- Case $i : \text{newpair}(Y, Z)$. This is very similar to the previous case.
- Case $i : \text{send } M$. send events do not change any bindings, so cannot falsify equation (1).
- Case $i : \text{receive } M$. The intruder does not know X in the final state, and so X must appear encrypted in M . Consider the smallest encrypted component of M containing the occurrence of X that gets misbound, say $\{M_1\}_K$, instantiating template $\{m_1\}_k$. Now $\sigma_1(0) \not\vdash M_1$ because $\sigma_1(0) \not\vdash X$. Hence by Lemma 6, $\{M_1\}_K \preceq \sigma_1(0)$. Now consider the earliest point in the trace at which $\{M_1\}_K \preceq \sigma(0)$. This was not true in the initial state by assumption 2. Hence it must have come about as the result of an event $j : \text{send } M'$ with $\{M_1\}_K \preceq M'$, say with $\{M_1\}_K$ instantiating template $\{m'_1\}_{k'}$. By the presumed minimality of the counterexample tr , node j has X bound only to x in this state, so X instantiates only x in this event. Also, $\text{type}(\{m_1\}_k) = \text{type}(\{m'_1\}_{k'})$, so by the disjoint encryption assumption, $\{m\}_k = \{m'\}_{k'}$. Hence X must instantiate the same variables of $\{m\}_k$ in the receive M event as it does of $\{m'\}_{k'}$ in the send M' event, namely just x . This gives a contradiction.

□

5.2 A theorem about secrecy

Suppose we have some set of values Xs that are always sent encrypted by some key K such that $K^{-1} \in Xs$; then if the intruder does not know any element of Xs initially, then he cannot undo any of the encryptions to learn a new value of Xs . We make this precise below.

Define the set of values Xs to be self encrypted in message M if every occurrence of $x \in Xs$ is encrypted by some key K such that $K^{-1} \in Xs$:

$$\begin{aligned} Xs \text{ selfEncln } Y &\Leftrightarrow Y \notin Xs, \quad \text{for } Y \in Val, \\ Xs \text{ selfEncln } (M_1, M_2) &\Leftrightarrow Xs \text{ selfEncln } M_1 \wedge Xs \text{ selfEncln } M_2, \\ Xs \text{ selfEncln } \{M\}_K &\Leftrightarrow K^{-1} \in Xs \vee Xs \text{ selfEncln } M. \end{aligned}$$

Note that if Y doesn't hold and element of Xs , then $Xs \text{ selfEncln } Y$:

Lemma 7

$$(\forall X \in Xs \bullet X \not\leq Y) \Rightarrow Xs \text{ selfEncln } Y.$$

Proof: Straightforward induction over the structure of Y . □

If every message that the intruder knows has Xs self encrypted, then the same is true of any message that the intruder can produce:

Lemma 8

$$(\forall M_1 \in B \bullet Xs \text{ selfEncln } M_1) \wedge B \vdash M \Rightarrow Xs \text{ selfEncln } M.$$

Proof: This is a fairly straightforward rule induction. We give just the interesting case, that of **decrypt**.

Suppose $B \vdash \{M\}_K \wedge B \vdash K^{-1}$, and suppose for a contradiction that $\neg Xs \text{ selfEncln } M$. Then by the inductive hypothesis, $Xs \text{ selfEncln } \{M\}_K$, and so $K^{-1} \in Xs$ by the definition of *selfEncln*. But the inductive hypothesis applied to K^{-1} gives $Xs \text{ selfEncln } K^{-1}$, and so $K^{-1} \notin Xs$, by the definition of *selfEncln*, giving a contradiction. □

Lemma 9

$$(\forall M_1 \in B \bullet Xs \text{ selfEncln } M_1) \Rightarrow \forall X \in Xs \bullet B \not\vdash X.$$

Proof: This follows immediately from the previous lemma, noting that for $X \in Xs$, $\neg (Xs \text{ selfEncln } X)$. □

Below, we will adapt this lemma to variables. We start by extending the *selfEncln* relation to message templates:

$$\begin{aligned} xs \text{ selfEncln } y &\Leftrightarrow y \notin xs, \quad \text{for } y \in Var, \\ xs \text{ selfEncln } (m_1, m_2) &\Leftrightarrow xs \text{ selfEncln } m_1 \wedge xs \text{ selfEncln } m_2, \\ xs \text{ selfEncln } \{m\}_k &\Leftrightarrow k^{-1} \in xs \vee xs \text{ selfEncln } m. \end{aligned}$$

One might think that in order for a protocol to keep the values of some set of variables secret xs , it is enough for xs to be self encrypted in every message of the protocol. However, that is not true. Consider the following protocol, where *pkb* represents b 's public key:

Message 1. $a \rightarrow b : \{x\}_{pkb}$.

The set of variables $\{x, skb\}$, where *skb* is the corresponding secret key, is self encrypted in every message of this protocol. However, b cannot deduce $knows(x) \subseteq \{a, b\}$, because the message is not

authenticated in any way. As another example, consider the following protocol (the repetition of x in message 2 is simply to ensure that the disjoint encryption property is satisfied):

Message 1. $a \rightarrow b : a, \{x\}_{pkb}$

Message 2. $b \rightarrow a : \{x, x\}_{pka}$.

Here the set of variables $\{x, ska, skb\}$ is self encrypted in every message. However, a cannot deduce $knows(x) \subseteq \{a, b\}$, because an attacker could change the value of a in message 1 to his own identity, so as to learn x from the resulting message 2.

To overcome these difficulties, we need certain fields in the protocol to be authenticated. The following relation captures the fact that y is authenticated by some key in xs :

$$\begin{aligned} y \text{ authedBy } xs \text{ in } (m_1, m_2) &\Leftrightarrow y \text{ authedBy } xs \text{ in } m_1 \vee \\ &\quad y \text{ authedBy } xs \text{ in } m_2, \\ y \text{ authedBy } xs \text{ in } \{m\}_k &\Leftrightarrow k \in xs \wedge y \ll m \vee y \text{ authedBy } xs \text{ in } m. \end{aligned}$$

Suppose, then, that we have a protocol where a set of variables xs is self encrypted in every message, and that when a value is newly received for some $x \in xs$, that value is authenticated by xs . Which nodes, then, could learn node i 's values for xs ? The answer is captured by the relation below. We write $i \equiv_{xs} j$ if, for some $x \in xs$, initially nodes i and j agree on the value of x , or one of them is dishonest and initially holds the other's value for x ; more precisely, \equiv_{xs} is the smallest equivalence relation such that

$$\begin{aligned} i \equiv_{xs} j &\Leftarrow \exists x \in xs \bullet \sigma_0(i).\rho(x) = \sigma_0(j).\rho(x) \vee \\ &\quad i = 0 \wedge \sigma_0(0) \vdash \sigma_0(j).\rho(x). \end{aligned}$$

We write $[i]_{xs}$ for the equivalence class of i under xs . We will show that, under certain circumstances, $[i]_{xs}$ is (at most) all the nodes that could learn i 's values for xs .

Theorem 2 Let Π be a feasible protocol. Let xs be a set of variables of Π , and define \overline{xs} to be xs closed under inverses:

$$\overline{xs} \triangleq xs \cup \{x^{-1} \mid x \in xs \cap \text{dom}(-^{-1})\}.$$

Let $i \in 1 \dots n$ such that $0 \not\equiv_{xs} i$. Suppose:

1. The protocol satisfies the disjoint encryption property.
2. Any honest agent who holds k 's value for $x \in xs$ and $k \in [i]_{xs}$ initially has it bound only to x , and any honest agent who holds the inverse of k 's value for x has it bound only to x^{-1} :

$$\begin{aligned} \forall x \in xs; k \in [i]_{xs} \bullet \\ (\sigma_0(k).\rho(x) \text{ boundOnlyTo } x)(\sigma_0) \wedge \\ x \in \text{dom}(-^{-1}) \Rightarrow ((\sigma_0(k).\rho(x))^{-1} \text{ boundOnlyTo } x^{-1})(\sigma_0). \end{aligned}$$

3. The intruder does not initially hold any message containing k 's value for $x \in xs$, for any $k \in [i]_{xs}$:

$$\forall x \in xs; k \in [i]_{xs} \bullet \sigma_0(k).\rho(x) \not\leq \sigma_0(0).$$

4. The variables xs are all self encrypted in all messages of the protocol:

$$\begin{aligned} \forall k \in [i]_{xs}; j \in 1 \dots \text{length}(\sigma_0(k).\text{prog}) \bullet \\ xs \text{ selfEncln } \sigma_0(k).\text{prog}(j). \end{aligned}$$

5. The variables in \overline{xs} are all authenticated when received:

$$\begin{aligned} \forall x \in \overline{xs}; k \in [i]_{xs}; j \in 1..length(\sigma_0(k).prog); m \text{ in } Template \mid \\ \sigma_0(k).prog(j) = \text{receive } m \wedge x \text{ newIn } (\sigma_0(k), j) \bullet \\ x \text{ authedBy } xs \text{ in } m. \end{aligned}$$

Where $x \text{ newIn } (s, j)$ means that the variable x is first bound in the local state s by being received in message j :

$$\begin{aligned} x \text{ newIn } ((prog, \rho, id), j) \Leftrightarrow x \notin \text{dom } \rho \wedge x \preceq prog(j) \wedge \\ \forall l < j \bullet x \not\preceq prog(l). \end{aligned}$$

Then for every $\sigma \in States(\Pi)$:

1. The set of values for xs held by members of $[i]_{xs}$ are self encrypted in the intruder's knowledge:

$$\forall M \in \sigma(0) \bullet \{\sigma(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\} \text{ selfEncln } M.$$

2. If $k \in [i]_{xs}$ then no agents other than $[i]_{xs}$ learn k 's values for xs :

$$\forall k \in [i]_{xs}; x \in xs \cap \text{dom } \sigma(k).\rho \bullet \text{knows}_{id}(\sigma(k).\rho(x))(\sigma) \subseteq [i]_{xs}.$$

3. For every $k \in [i]_{xs}$ and $x \in \overline{xs} \cap \text{dom } \sigma(k).\rho$, either (a) k 's value for x matches j 's initial value for x , i.e. $\sigma(k).\rho(x) = \sigma_0(j).\rho(x)$, for some $j \in [i]_{xs}$, or (b) it was generated via a $j : \text{new } X$ or $j : \text{newpair}(X, Y)$ event, instantiating x , for some $j \in [i]_{xs}$.

Proof: First, the condition is satisfied in the initial state σ_0 : we prove each condition in turn:

1. Assumption 3 tells us that no element of $\{\sigma_0(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\}$ is a submessage of any message of $\sigma_0(0)$, so

$$\{\sigma_0(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\} \text{ selfEncln } M$$

for every $M \in \sigma_0(0)$, by Lemma 7.

2. Let $k \in [i]_{xs}$ and $x \in xs \cap \text{dom } \sigma_0(i).\rho$. $0 \neq_{xs} i$ so $0 \neq_{xs} k$ and so $0 \notin \text{knows}_{id}(\sigma_0(k).\rho(x))(\sigma_0)$. And for $j > 0$, if $j \in \text{knows}_{id}(\sigma_0(k).\rho(x))(\sigma_0)$ then by assumption 2, $\sigma_0(j).\rho(x) = \sigma_0(k).\rho(x)$ and so $j \in [k]_{xs} = [i]_{xs}$. Hence $\text{knows}_{id}(\sigma_0(k).\rho(x))(\sigma_0) \subseteq [i]_{xs}$.

3. The result (specifically option (a)) holds trivially.

Suppose, for a contradiction, that the result does not hold, and consider the shortest trace tr leading to a counterexample state. Let tr be of the form $tr' \wedge \langle \sigma, j : E, \sigma' \rangle$.

Note that for $x \in xs$ and $k \in [i]_{xs}$, the conditions of Theorem 1 hold for $X = \sigma(k).\rho(x)$ in state σ :

1. The disjoint encryption property holds by assumption 1 of the current theorem.
2. The intruder did not initially know X by part 3 of the inductive hypothesis and: in case (a), assumption 3 of the current theorem; and in case (b), the fact that new values are freshly generated.
3. If any agent held X initially, then by part 3 of the inductive hypothesis (necessarily part (a)), $X = \sigma_0(j).\rho(x)$ for some $j \in [i]_{xs}$; so by assumption 2 of the current theorem (applied to j), X is initially bound only to x .
4. The intruder does not know X in σ by part 2 of the inductive hypothesis.
5. If X is generated via a new or newpair event, it does so to instantiate x by part 3 (b) of the inductive hypothesis.

Hence by Theorem 1:

$$\forall k \in [i]_{xs}; x \in xs \cap \text{dom } \sigma(k).\rho \bullet (\sigma(k).\rho(x) \text{ boundOnlyTo } x)(\sigma). \quad (2)$$

We now perform a case analysis over the last event E :

- Case j : new Y , with Y instantiating y . We prove the three parts of the result separately:

1. Note that

$$\begin{aligned} \{\sigma'(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\} = \\ \begin{cases} \{\sigma(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\} \cup \{Y\} & \text{if } y \in xs \wedge j \in [i]_{xs} \\ \{\sigma(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\} & \text{otherwise} \end{cases} \end{aligned}$$

and $\sigma'(0) = \sigma(0)$, and $Y \not\leq \sigma'(0)$. Hence $\{\sigma'(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\}$ *selfEncln* M for every $M \in \sigma'(0)$, by the inductive hypothesis.

2. Let $k \in [i]_{xs}$ and $x \in \text{dom } \sigma(k).\rho$. If $x \neq y$ or $k \neq j$, then

$$\text{knows}_{id}(\sigma'(k).\rho(x))(\sigma') = \text{knows}_{id}(\sigma(k).\rho(x))(\sigma) \subseteq [i]_{xs}$$

by the inductive hypothesis. And if $x = y$ and $k = j$

$$\text{knows}_{id}(\sigma'(k).\rho(x))(\sigma') = \{k\} \subseteq [i]_{xs},$$

because Y is fresh.

3. Let $k \in [i]_{xs}$ and $x \in \text{dom } \sigma(k).\rho$. If $x = y$ and $k = j$, part (b) clearly holds. In all other cases, the result holds by the inductive hypothesis.

- Case j : newpair(X, Y). This is very similar to the previous case.

- Case j : send M .

1. Note that no bindings change as a result of the event, so $\{\sigma'(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\} = \{\sigma(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\}$.

If $j \notin [i]_{xs}$, then by part 2 of the inductive hypothesis, j knows no value from $\{\sigma(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\}$, and so M contains no such value, and hence $\{\sigma'(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\}$ *selfEncln* M , by Lemma 7. The result then holds by the inductive hypothesis.

If $j \in [i]_{xs}$, then, from equation (2), if M contains $\sigma(k).\rho(x)$, it does so bound to x (because it was bound only to x in $\sigma(j).\rho$). Then by assumption 4, x is encrypted in the corresponding template by y^{-1} for some $y \in xs$; and so in M , $\sigma(k).\rho(x)$ is encrypted by $\sigma(j).\rho(y^{-1})$, which equals $(\sigma(j).\rho(y))^{-1}$ by Lemma 2 and the convention concerning inverses mentioned at the start of the theorem. Hence $\{\sigma'(k).\rho(x) \mid k \in [i]_{xs}, x \in xs\}$ *selfEncln* M . The result then holds by the inductive hypothesis.

2. Let $k \in [i]_{xs}$ and $x \in xs \cap \text{dom } \sigma(k).\rho$. By the previous part and Lemma 8, $0 \notin \text{knows}_{id}(\sigma'(k).\rho(x))(\sigma')$. Further, no binding of an honest agent changes as a result of this event, so $\text{knows}_{id}(\sigma'(k).\rho(x))(\sigma') = \text{knows}_{id}(\sigma(k).\rho(x))(\sigma) \subseteq [i]_{xs}$, by the inductive hypothesis.

3. No bindings change as a result of this event, so the result holds by the inductive hypothesis.

- Case j : receive M . Consider first the case $j \in [i]_{xs}$.

Suppose this event binds $x \in \overline{xs}$ to some value X in j 's state. By assumption 5, X is encrypted by some key Y_1^{-1} bound to $y_1^{-1} \in xs$. From the definition of a feasible protocol, j is able to decrypt this component to learn X ; therefore he either already has Y_1 bound to y_1 in his state $\sigma(j).\rho$, or learns this value from the message. In the latter case, because $y_1 \in \overline{xs}$, we can apply the same argument to this variable as we just applied to x . Continuing in this way, we obtain a sequence of variables $y_0 = x, y_1, \dots, y_m$ and values $Y_0 = X, Y_1, \dots, Y_m$ such that: y_m is bound to Y_m in j 's previous state $\sigma(j).\rho$; each y_l is encrypted by y_{l+1}^{-1} in the message,

for $l = 0 \dots m - 1$; y_l is bound to Y_l in the message; and x and each of the y_l^{-1} for $l = 1 \dots m$ are members of xs .

By part 2 of the inductive hypothesis, only agents in $[i]_{xs}$ know Y_m^{-1} . Hence the component encrypted by Y_m^{-1} and containing Y_{m-1} must have been created by some honest agent $k \in [i]_{xs}$. By the disjoint encryption assumption (assumption 1), k must have had Y_{m-1} bound to y_{m-1} when he sent this message, i.e.

$$\exists k \in [i]_{xs} \bullet Y_{m-1} = \sigma(k). \rho(y_{m-1}).$$

Then by part 3 of the inductive hypothesis, either: (a) Y_{m-1} matches the value initially held by some member of $[i]_{xs}$ for y_{m-1} , in which case Y_{m-1}^{-1} was initially held by some member of $[i]_{xs}$ for y_{m-1}^{-1} , by assumption 2; or (b) Y_{m-1} was generated by a new or newpair event for y_{m-1} , in which case Y_{m-1}^{-1} was generated in the same event for y_{m-1}^{-1} . In either case, by part 2 of the inductive hypothesis, only agents in $[i]_{xs}$ know Y_{m-1}^{-1} . We can continue in this way to show that each of the Y_l equals $\sigma(k). \rho(y_l)$ for some $k \in [i]_{xs}$, for $l = 0 \dots m - 1$. In particular, the value X received for x equals $\sigma(k). \rho(x)$ for some $k \in [i]_{xs}$.

We can now prove the three parts of the result in turn.

1. By the above argument, for every $x \in xs$ that is bound as a result of the event, the value of $\sigma'(j). \rho(x) = \sigma(k). \rho(x)$ for some $k \in [i]_{xs}$. Hence

$$\{\sigma'(k). \rho(x) \mid k \in [i]_{xs}, x \in xs\} = \{\sigma(k). \rho(x) \mid k \in [i]_{xs}, x \in xs\}.$$

Further, the intruder's knowledge doesn't change as a result of this event: $\sigma'(0) = \sigma(0)$. Hence the result holds by the inductive hypothesis.

2. First consider $x \in xs$ that is bound within j 's state by this event. Then by the above argument it is bound to $\sigma(k). \rho(x)$ for some $k \in [i]_{xs}$. Hence

$$\begin{aligned} & \text{knows}_{id}(\sigma'(j). \rho(x))(\sigma') \\ &= \text{knows}_{id}(\sigma(k). \rho(x))(\sigma') \\ &= \langle \text{only } j \text{ learns the value from this event} \rangle \\ & \quad \text{knows}_{id}(\sigma(k). \rho(x))(\sigma) \cup \{j\} \\ &\subseteq \langle \text{inductive hypothesis; } j \in [i]_{xs} \rangle \\ & \quad [i]_{xs}. \end{aligned}$$

And for $l \neq j$, $l \in [i]_{xs}$:

$$\begin{aligned} & \text{knows}_{id}(\sigma'(l). \rho(x))(\sigma') \\ &= \langle l\text{'s binding does not change as a result of this event} \rangle \\ & \quad \text{knows}_{id}(\sigma(l). \rho(x))(\sigma') \\ &\subseteq \langle \text{at most } j \text{ learns } \sigma(l). \rho(x) \text{ as a result of this event} \rangle \\ & \quad \text{knows}_{id}(\sigma(l). \rho(x))(\sigma) \cup \{j\} \\ &\subseteq \langle \text{inductive hypothesis; } j \in [i]_{xs} \rangle \\ & \quad [i]_{xs}. \end{aligned}$$

Finally, for $x \in xs$ not bound as a result of this event, and for $l \in [i]_{xs}$:

$$\begin{aligned} & \text{knows}_{id}(\sigma'(l). \rho(x))(\sigma') \\ &= \langle \text{no binding for } x \text{ changes as a result of this event} \rangle \\ & \quad \text{knows}_{id}(\sigma(l). \rho(x))(\sigma) \\ &\subseteq \langle \text{inductive hypothesis} \rangle \\ & \quad [i]_{xs}. \end{aligned}$$

3. By the above, any variable $x \in xs$ bound for the first time as a result of this event is bound to $\sigma(k).\rho(x)$ for some $k \in [i]_{xs}$. The result then follows from the inductive hypothesis.

Finally, consider the case $j \notin [i]_{xs}$. Parts 1 and 3 of the result hold immediately, because no relevant part of the state changes as a result of the event. For part 2, suppose for a contradiction that some variable y in j 's state gets bound by this event to $\sigma(k).\rho(x)$ for some $k \in [i]_{xs}$. From part 1 of the inductive hypothesis, the value of $\sigma(k).\rho(x)$ is encrypted by other elements of $\{\sigma(l).\rho(z)^{-1} \mid l \in [i]_{xs}, z \in xs\}$ in the intruder's knowledge, and hence in the message M received. Without loss of generality, suppose that $\sigma(k).\rho(x)$ is the first such value obtained; then it must have been encrypted in M by some $\sigma(l).\rho(z)^{-1}$ such that $\sigma(l).\rho(z)$ is already held by j in state σ . But this contradicts part 2 of the inductive hypothesis.

□

6 Abstract messages

In this section we consider abstract messages in more detail. We consider each of the atomic abstract messages from Section 2, as well as some additional abstract messages that we believe will prove useful; we also consider concrete messages as abstract messages, and the conjunction of abstract messages. For each of the constructs, we give a formal semantics, proof rules governing how it can be used in annotations, and example refinements with concrete messages.

6.1 Refinement

Recall that the semantics of an abstract message am in protocol Π is the set of message templates that could be used to implement am , written $\llbracket am \rrbracket_{\Pi}$.

Recall also that we write $am \sqsubseteq am'$ if abstract message am can be implemented by am' . We define a protocol-dependent notion of refinement by

$$am \sqsubseteq_{\Pi} am' \Leftrightarrow \llbracket am \rrbracket_{\Pi} \supseteq \llbracket am' \rrbracket_{\Pi}.$$

And define a protocol-independent refinement by

$$am \sqsubseteq am' \Leftrightarrow \forall \Pi \bullet am \sqsubseteq_{\Pi} am',$$

where the quantification ranges over all protocols that satisfy the disjoint encryption property. We will tend to work with the protocol-independent version, and will give message refinement rules under this relation: these rules are more generic and more easily reusable than protocol-dependent ones.

The following lemma follows directly from the definition.

Lemma 10 Refinement is a preorder.

The following rules show how refinement can be used within annotations: refining a sent or received message preserves the correctness of an annotation.

Annotation rule 8 (Refine sent message)

$$\frac{a : \{pre\} \text{ send } m \{post\} \quad m \sqsubseteq_{\Pi} m'}{a : \{pre\} \text{ send } m' \{post\}}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \hat{\ } e' \hat{\ } es_1 \wedge e' \sqsupseteq \text{ send } m' \wedge \\ \forall \sigma \in \text{States}(\Pi) \mid \sigma(i).prog = e' \hat{\ } es_1 \bullet pre(\sigma)[i]. \end{aligned}$$

Then $e' \sqsupseteq \text{ send } m$ by the second hypothesis. So by the first hypothesis,

$$\forall \sigma' \in \text{States}(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i],$$

Hence $\{pre\}a : \text{ send } m' \{post\}$. □

Annotation rule 9 (Refine received message)

$$\frac{b : \{pre\} \text{ receive } m \{post\} \quad m \sqsubseteq_{\Pi} m'}{b : \{pre\} \text{ receive } m' \{post\}}$$

Proof: Similar to the previous proof. □

6.2 Concrete messages

We consider concrete messages to be a particular type of abstract messages. Using a concrete message as an abstract message allows a protocol designer to limit the protocol to a specific form of message. It is the simplest way to specify directly that information should be passed without change.

6.2.1 Semantics

The semantics of a concrete message is simply the singleton set containing the concrete message.

$$\llbracket x \rrbracket_{\Pi} = \{x\}.$$

6.3 Conjunction

Abstract messages can be combined by conjunction: the conjoined abstract message represents the conjunction of the requirements of the components.

The semantics of a conjunction is the intersection of the semantics of the two components:

$$\llbracket m_1 \wedge m_2 \rrbracket_{\Pi} = \llbracket m_1 \rrbracket_{\Pi} \cap \llbracket m_2 \rrbracket_{\Pi}.$$

It is worth considering the case where $\llbracket m_1 \wedge m_2 \rrbracket_{\Pi} = \{\}$, which is the case when m_1 and m_2 represent incompatible requirements. Such a specification is infeasible: it suggests that the protocol designer has made an error, leaving too many requirements in one abstract message.

The following lemma follows directly from the definition.

Lemma 11 Conjunction represents the least upper bound relation with respect to refinement.

The following two rules relate conjunction to refinement.

Refinement rule 1 (Refinement by conjunction)

$$m \sqsubseteq m \wedge m'$$

Refinement rule 2 (Conjunction of requirements)

$$\frac{\begin{array}{l} m_1 \sqsubseteq m \\ m_2 \sqsubseteq m \end{array}}{m_1 \wedge m_2 \sqsubseteq m}$$

From these and earlier rules, we can deduce the following corollaries.

Annotation rule 10 (Conjunction of sent messages)

$$\frac{\begin{array}{l} \{pre\} \text{ send } m_1 \{post_1\} \\ \{pre\} \text{ send } m_2 \{post_2\} \end{array}}{\{pre\} \text{ send } m_1 \wedge m_2 \{post_1 \wedge post_2\}}$$

Proof: From Refinement Rule 1, $m_1 \sqsubseteq m_1 \wedge m_2$. Hence from Annotation Rule 8 and the first hypothesis, $\{pre\} \text{ send } m_1 \wedge m_2 \{post_1\}$. Similarly, $\{pre\} \text{ send } m_1 \wedge m_2 \{post_2\}$. Hence by Annotation Rule 4,

$$\{pre\} \text{ send } m_1 \wedge m_2 \{post_1 \wedge post_2\}.$$

□

Annotation rule 11 (Conjunction of received messages)

$$\frac{\begin{array}{l} \{pre\} \text{ receive } m_1 \{post_1\} \\ \{pre\} \text{ receive } m_2 \{post_2\} \end{array}}{\{pre\} \text{ receive } m_1 \wedge m_2 \{post_1 \wedge post_2\}}$$

Proof: Similar to the previous rule. □

6.4 any

The abstract message *any* represents a completely arbitrary message. It is useful when a message has no security requirements from the perspective of the current annotation, but is needed to fit in with the message flow of the annotation of some other agent: we will demonstrate this use in Section 7.

The semantics of *any* is the set of all concrete message templates:

$$\llbracket any \rrbracket_{\Pi} \hat{=} Template.$$

It is refined by all concrete message templates:

Annotation rule 12 For all concrete message templates *m*:

$$any \sqsubseteq m.$$

6.5 canExtract and keepSecret

The abstract message $canExtract_k(x)$ represents those concrete templates *m*, such that when they are sent, no other agent can learn the value of *x* unless they know the value of *k*; this should remain true until this agent performs another event; however, if the intruder knew the value of *x* or *k* when the message was sent, then he can pass off *x* to other agents so that they can learn it, so the above condition no longer holds. More precisely, suppose an instantiation of *m* is sent from state σ by node *i* (i.e. $i : \text{send } m[\sigma(i).\rho]$), and *i* performs no subsequent events before state σ' , and the intruder didn't know *i*'s values *K* of *k* or *X* of *x* in state σ (i.e. $K = \sigma(i).\rho(k)$ and $X = \sigma(i).\rho(x)$); then those nodes that know *X* in σ' knew either *X* or *K* before.

$$\begin{aligned} \llbracket canExtract_k(x) \rrbracket_{\Pi} = & \{m \mid \forall tr \hat{\wedge} \langle \sigma, i : \text{send } m[\sigma(i).\rho] \rangle \hat{\wedge} tr' \hat{\wedge} \langle \sigma' \rangle \in traces(\Pi) \mid \\ & tr' \upharpoonright i = \langle \rangle \wedge \sigma(0) \not\vdash K \wedge \sigma(0) \not\vdash X \bullet \\ & knows_{id}(X)(\sigma') \subseteq knows_{id}(K)(\sigma) \cup knows_{id}(X)(\sigma) \\ & \text{where } X = \sigma(i).\rho(x), K = \sigma(i).\rho(k)\}. \end{aligned}$$

Recall that if $k \notin \text{dom } \sigma(i).\rho$, $k^{-1} \in \text{dom } \sigma(i).\rho$ then $\sigma(i).\rho(k)$ is shorthand for $(\sigma(i).\rho(k^{-1}))^{-1}$; this will be useful when an agent has a public key pk in its state, but not the corresponding secret key sk ; in this case, the message $canExtract_{sk}(x)$ will have the expected meaning: that *x* can be extracted only by a holder of the appropriate secret key.

The abstract message $canExtract_k(xs)$, where *xs* is a set of variables, places the same condition for all elements of *xs*:

$$\llbracket canExtract_k(xs) \rrbracket_{\Pi} \hat{=} \bigcap_{x \in xs} \llbracket canExtract_k(x) \rrbracket_{\Pi}.$$

We abuse notation and omit set brackets where this does not cause confusion; for example we write $canExtract_k(x_1, x_2, x_3)$ as a shorthand for $canExtract_k(\{x_1, x_2, x_3\})$. Note that

$$canExtract_k(x_1, \dots, x_m) = canExtract_k(x_1) \wedge \dots \wedge canExtract_k(x_m).$$

The abstract message $\text{canExtract}_{ks}(x)$ specifies that if an agent learns the value of x , then he or the intruder must have known the value of some $k \in ks$; in other words, all elements of ks must be protected in order to protect x :

$$\begin{aligned} \llbracket \text{canExtract}_{ks}(x) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \wedge \langle \sigma, i : \text{send } m[\sigma(i).\rho] \rangle \wedge tr' \wedge \langle \sigma' \rangle \in \text{traces}(\Pi) \mid \\ & \quad tr' \uparrow i = \langle \rangle \wedge (\forall k \in ks \bullet \sigma(0) \not\vdash \sigma(i).\rho(k)) \wedge \sigma(0) \not\vdash X \bullet \\ & \quad \text{knows}_{id}(X)(\sigma') \subseteq \\ & \quad \quad \cup \{ \text{knows}_{id}(\sigma(i).\rho(k))(\sigma) \mid k \in ks \} \cup \text{knows}_{id}(X)(\sigma) \\ & \quad \text{where } X = \sigma(i).\rho(x) \}. \end{aligned}$$

The following is an obvious extension:

$$\llbracket \text{canExtract}_{ks}(xs) \rrbracket_{\Pi} \hat{=} \bigcap_{x \in xs} \llbracket \text{canExtract}_{ks}(x) \rrbracket_{\Pi}.$$

The abstract message $\text{keepSecret}(x)$ specifies that x must not be revealed to anyone. It can be defined as syntactic sugar, as follows:

$$\text{keepSecret}(x) \hat{=} \text{canExtract}_{\{\}}(x).$$

This gives the following semantic equation:

$$\begin{aligned} \llbracket \text{keepSecret}(x) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \wedge \langle \sigma, i : \text{send } m[\sigma(i).\rho] \rangle \wedge tr' \wedge \langle \sigma' \rangle \in \text{traces}(\Pi) \mid \\ & \quad tr' \uparrow i = \langle \rangle \wedge \sigma(0) \not\vdash X \bullet \\ & \quad \text{knows}_{id}(X)(\sigma') = \text{knows}_{id}(X)(\sigma) \\ & \quad \text{where } X = \sigma(i).\rho(x) \}. \end{aligned}$$

The keepSecret message is extended to sets of variables in the obvious way:

$$\text{keepSecret}(xs) \hat{=} \text{canExtract}_{\{\}}(xs),$$

so

$$\llbracket \text{keepSecret}(xs) \rrbracket_{\Pi} = \bigcap_{x \in xs} \llbracket \text{keepSecret}(x) \rrbracket_{\Pi}.$$

6.5.1 Annotation rules

The following proof rule shows how canExtract can be used in annotations.

Annotation rule 13 (canExtract) For all sets s_k and s_x of variables representing agent identities,

$$\begin{aligned} a : & \left\{ \text{knows}(k) = s_k \wedge \text{knows}(x) = s_x \wedge \text{honest}(s_k \cup s_x) \right\} \\ & \text{send } \text{canExtract}_k(x) \\ & \left\{ s_x \subseteq \text{knows}(x) \subseteq s_k \cup s_x \right\} \end{aligned}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \wedge e' \wedge es_1 \wedge e' \sqsupseteq \text{send } \text{canExtract}_k(x) \wedge \\ \forall \sigma \in \text{States}(\Pi) \mid \sigma(i).prog = e' \wedge es_1 \bullet \\ (\text{knows}(k) = s_k \wedge \text{knows}(x) = s_x \wedge \text{honest}(s_k \cup s_x))(\sigma)[i]. \end{aligned}$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the event immediately before the event corresponding to e' . Let

$$\begin{aligned} K = \sigma(i).\rho(k), \quad X = \sigma(i).\rho(x), \\ S_k = \{ \sigma(i).\rho(b) \mid b \in s_k \}, \quad S_x = \{ \sigma(i).\rho(b) \mid b \in s_x \}. \end{aligned}$$

Then substituting in the assumption corresponding to the precondition, we have

$$\mathit{knows}(K)(\sigma) = S_k \wedge \mathit{knows}(X)(\sigma) = S_x \wedge \mathit{honest}(S_k \cup S_x).$$

Hence $\sigma(0) \not\vdash K$ and $\sigma(0) \not\vdash X$. So from the definition of $\mathit{canExtract}_k(x)$

$$\mathit{knows}_{id}(X)(\sigma') \subseteq \mathit{knows}_{id}(K)(\sigma) \cup \mathit{knows}_{id}(X)(\sigma).$$

Hence

$$\mathit{knows}(X)(\sigma') \subseteq \mathit{knows}(K)(\sigma) \cup \mathit{knows}_{id}(X)(\sigma) = S_k \cup S_x.$$

But $\sigma'(i).\rho(x) = \sigma(i).\rho(x)$, and similarly for the elements of s_k and s_x , and so

$$(\mathit{knows}(x) \subseteq s_k \cup s_x)(\sigma')[i]$$

as required. □

The following rule shows how $\mathit{keepSecret}$ can be used in annotations.

Annotation rule 14 (**keepSecret**)

$$\frac{\left\{ \mathit{knows}(x) = s \wedge \mathit{honest}(s) \right\}}{\text{send } \mathit{keepSecret}(x)} \left\{ \mathit{knows}(x) = s \right\}$$

Proof: Similar to the proof of the previous rule. □

6.5.2 Refinement rules

The $\mathit{canExtract}$ messages are monotonic in their main argument, and anti-monotonic in their key argument:

Refinement rule 3 If $xs \subseteq xs' \wedge ks \supseteq ks'$ then

$$\mathit{canExtract}_{ks}(xs) \subseteq \mathit{canExtract}_{ks'}(xs').$$

Proof: Direct from the semantics. □

The following is an immediate corollary:

Refinement rule 4

$$\mathit{canExtract}_{ks}(xs) \subseteq \mathit{keepSecret}(xs).$$

It seems difficult to produce rules that allow the $\mathit{canExtract}$ and $\mathit{keepSecret}$ messages to be refined to concrete messages. The reason for this is that secrecy of values is very much a property of the protocol as a whole, rather than just individual messages. For example, one might think that $\mathit{canExtract}_k(x)$ is refined by $\{x\}_k$; however, this is not true if the recipient of the message then sends back x , or any message from which x can be obtained. We intend to develop such rules, building on the foundations provided by Theorem 2.

6.6 *provesKnowledgeOf*

The abstract message $\mathit{provesKnowledgeOf}(x)$ proves to the recipient of the message that some agent knows the recipient's value of x , and, if that agent is not the intruder, he has that value bound to his own variable x . This allows the receiver to verify state information about the sender concerning the variable x . $\mathit{provesKnowledgeOf}$ specifies nothing about who may learn data from this message.

6.6.1 Semantics

The semantics of $provesKnowledgeOf(x)$ is the set of messages m that if received by some node i , means that in the previous state σ , either the intruder knew i 's value X for x , or some other honest node j had its x variable bound to X :

$$\begin{aligned} \llbracket provesKnowledgeOf(x) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \cap \langle \sigma, i : \text{receive } m, \sigma' \rangle \in traces(\Pi) \bullet \\ & \quad \sigma(0) \vdash X \vee \\ & \quad \exists j > 0 \bullet j \neq i \wedge \sigma(j). \rho(x) = X \\ & \quad \text{where } X = \sigma'(i). \rho(x)\}. \end{aligned}$$

The following is an obvious extension:

$$\begin{aligned} \llbracket provesKnowledgeOf(x_1, \dots, x_m) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \cap \langle \sigma, i : \text{receive } m, \sigma' \rangle \in traces(\Pi) \bullet \\ & \quad (\forall l \in 1..m \bullet \sigma(0) \vdash X_l) \vee \\ & \quad \exists j > 0 \bullet j \neq i \wedge \forall l \in 1..m \bullet \sigma(j). \rho(x_l) = X_l \\ & \quad \text{where } X_l = \sigma'(i). \rho(x_l) \text{ for } l \in 1..m\}. \end{aligned}$$

Note that the abstract message $provesKnowledgeOf(x_1, \dots, x_m)$ is not the same as $provesKnowledgeOf(x_1) \wedge \dots \wedge provesKnowledgeOf(x_m)$: in the latter abstract message, it might be different agents who know the different x_l .

It is useful to define an extension of $provesKnowledgeOf$ where the recipient receives evidence of the role played by the other agent; the abstract message $provesKnowledgeOf(x_1, \dots, x_m, id = b)$ tells the recipient that the other agent was following a role with identity variable b :

$$\begin{aligned} \llbracket provesKnowledgeOf(x_1, \dots, x_m, id = b) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \cap \langle \sigma, i : \text{receive } m, \sigma' \rangle \in traces(\Pi) \bullet \\ & \quad (\forall l \in 1..m \bullet \sigma(0) \vdash X_l) \vee \\ & \quad \exists j > 0 \bullet j \neq i \wedge \sigma(j). id = b \wedge \forall l \in 1..m \bullet \sigma(j). \rho(x_l) = X_l \\ & \quad \text{where } X_l = \sigma'(i). \rho(x_l) \text{ for } l \in 1..m\}. \end{aligned}$$

The $provesKnowledgeOfNR$ abstract messages are slightly stronger, as they give the recipient the additional guarantee that the message was not replayed from himself: the other node j has an identity different from the receiving node i :

$$\begin{aligned} \llbracket provesKnowledgeOfNR(x_1, \dots, x_m) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \cap \langle \sigma, i : \text{receive } m, \sigma' \rangle \in traces(\Pi) \bullet \\ & \quad (\forall l \in 1..m \bullet \sigma(0) \vdash X_l) \vee \\ & \quad \exists j > 0 \bullet \sigma(j). \rho(\sigma(j). id) \neq \sigma(i). \rho(\sigma(i). id) \wedge \\ & \quad \quad \forall l \in 1..m \bullet \sigma(j). \rho(x_l) = X_l \\ & \quad \text{where } X_l = \sigma'(i). \rho(x_l) \text{ for } l \in 1..m\}, \\ \llbracket provesKnowledgeOfNR(x_1, \dots, x_m, id = b) \rrbracket_{\Pi} = & \\ & \{m \mid \forall tr \cap \langle \sigma, i : \text{receive } m, \sigma' \rangle \in traces(\Pi) \bullet \\ & \quad (\forall l \in 1..m \bullet \sigma(0) \vdash X_l) \vee \\ & \quad \exists j > 0 \bullet \sigma(j). id = b \wedge \sigma(j). \rho(b) \neq \sigma(i). \rho(\sigma(i). id) \wedge \\ & \quad \quad \forall l \in 1..m \bullet \sigma(j). \rho(x_l) = X_l \\ & \quad \text{where } X_l = \sigma'(i). \rho(x_l) \text{ for } l \in 1..m\}. \end{aligned}$$

6.6.2 Annotation rules

The following proof rule shows how $provesKnowledgeOf$ can be used in annotations.

Annotation rule 15 (provesKnowledgeOf.1)

$$a : \left\{ \begin{array}{l} true \\ \text{receive } \text{provesKnowledgeOf}(x_1, \dots, x_m) \\ \left\{ \exists b \in \text{Var}; B \in \text{Val} \bullet \text{session}(b \rightsquigarrow B; x_1, \dots, x_m) \right\} \end{array} \right\}$$

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id &= a \wedge \sigma_0(i).prog = es_0 \hat{\wedge} e' \hat{\wedge} es_1 \wedge \\ e' &\sqsupseteq \text{receive } \text{provesKnowledgeOf}(x_1, \dots, x_m) \wedge \\ \forall \sigma \in \text{States}(\Pi) \mid \sigma(i).prog &= e' \hat{\wedge} es_1 \bullet \text{true}(\sigma)[i]. \end{aligned}$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the state corresponding to e' . Let $X_l = \sigma'(i).\rho(x_l)$ for $l = 1 \dots m$. Then, from the semantics of $\text{provesKnowledgeOf}(x)$, there are two possibilities:

- Case $\forall l \in 1 \dots m \bullet \sigma(0) \vdash X_l$, so $\forall l \in 1 \dots m \bullet \sigma'(0) \vdash X_l$. Then, for arbitrary $b \in \text{Var}$,

$$\text{session}(b \rightsquigarrow \text{intruder}; x_1 \rightsquigarrow X_1, \dots, x_m \rightsquigarrow X_m)(\sigma')$$

from the definition of *session*. So

$$\exists b, B \bullet \text{session}(b \rightsquigarrow B; x_1 \rightsquigarrow X_1, \dots, x_m \rightsquigarrow X_m)(\sigma'),$$

and so

$$(\exists b, B \bullet \text{session}(b \rightsquigarrow B; x_1 \rightsquigarrow x_1, \dots, x_m \rightsquigarrow x_m))(\sigma')[i],$$

as required.

- Case for some $j > 0$, $j \neq i \wedge \forall l \in 1 \dots m \bullet \sigma(j).\rho(x_l) = X_l$. Let $b = \sigma(j).id$ and $B = \sigma(j).\rho(b)$. Then $\text{session}(b \rightsquigarrow B; x_1 \rightsquigarrow X_1, \dots, x_m \rightsquigarrow X_m)(\sigma)$ and so $\text{session}(b \rightsquigarrow B; x_1 \rightsquigarrow X_1, \dots, x_m \rightsquigarrow X_m)(\sigma')$. The proof then proceeds as in the previous case. □

The following three rules show how the variants of *provesKnowledgeOf* give the recipient extra information about the other agent.

Annotation rule 16 (provesKnowledgeOf.2)

$$a : \left\{ \begin{array}{l} true \\ \text{receive } \text{provesKnowledgeOf}(x_1, \dots, x_m, id = b) \\ \left\{ \exists B \in \text{Val} \bullet \text{session}(b \rightsquigarrow B; x_1, \dots, x_m) \right\} \end{array} \right\}$$

Proof: This is a straightforward adaptation of the proof of the previous rule. □

Annotation rule 17 (provesKnowledgeOfNR.1)

$$a : \left\{ \begin{array}{l} true \\ \text{receive } \text{provesKnowledgeOfNR}(x_1, \dots, x_m) \\ \left\{ \exists b \in \text{Var}; B \in \text{Val} \bullet \text{session}(b \rightsquigarrow B; x_1, \dots, x_m) \wedge B \neq a \right\} \end{array} \right\}$$

Proof: This is a straightforward adaptation of the proof of Rule 15. In the first case, $B = \text{intruder} \neq \sigma'(i).\rho(a)$; but this is equivalent to $(B \neq a)(\sigma')[i]$, as required. In the second case, we have the additional condition that $\sigma(j).\rho(\sigma(j).id) \neq \sigma(i).\rho(\sigma(i).id)$, which is equivalent to $B \neq \sigma'(i).\rho(a)$, i.e. $(B \neq a)(\sigma')[i]$, as required. □

Annotation rule 18 (provesKnowledgeOfNR.2)

$$a : \left\{ \begin{array}{l} true \\ \text{receive } \text{provesKnowledgeOfNR}(x_1, \dots, x_m, id = b) \\ \left\{ \exists B \in Val \bullet \text{session}(b \rightsquigarrow B; x_1, \dots, x_m) \wedge B \neq a \right\} \end{array} \right\}$$

Proof: Straightforward adaptation of the previous two proofs. □

6.6.3 Example refinements

Developing and proving refinement rules for the *provesKnowledgeOf* abstract messages will be the subject of future work. We briefly discuss here how we expect those rules to be.

We believe that if m is any message that contains x , and the protocol satisfies the disjoint encryption property, then $\text{provesKnowledgeOf}(x) \sqsubseteq_{\Pi} m$. If the value X of x is sent unencrypted, then the intruder knows X , so the first disjoint of the definition is satisfied. If X is within an encrypted component, then either that component was created by the intruder in which case he knows X , or it was created by an honest agent, who must have X bound to x because of the disjoint encryption assumption.

Further, if $\text{provesKnowledgeOf}(xs) \sqsubseteq m$, and the only role in the protocol that can ever send the message m is associated with b — which will be the case if m is an encrypted message, b sends a message of this form, and the protocol satisfies the disjoint encryption property — then we believe $\text{provesKnowledgeOf}(xs, id = b) \sqsubseteq m$.

Finally, if $\text{provesKnowledgeOf}(x) \sqsubseteq m$, and x is bound within m to either the identity of the claimed sender or intended recipient, then we believe that $\text{provesKnowledgeOfNR}(x) \sqsubseteq m$: the recipient would be able to spot if the message is replayed in this case.

6.7 bind

The abstract message $\text{bind}_x(y)$ represents the requirement that the values of x and y should be bound together within the state of the recipient. More precisely, it means that if the intruder does not learn x , then any other agent who obtains the value of x will also obtain the value of y . The intention is that x will normally be a secret. This message allows the sender to deduce that any other agent who receives x associates it with y .

6.7.1 Semantics

The semantics of $\text{bind}_x(y)$ is the set of messages m such that if participant i sends m , then anyone who subsequently has their x variable bound to the same value, will also have their y variable bound to the same value, unless the intruder learns x .

$$\begin{aligned} \llbracket \text{bind}_x(y) \rrbracket_{\Pi} = & \\ & \{ m \mid \forall tr \wedge \langle \sigma, i : \text{send } m \rangle \wedge tr' \wedge \langle \sigma' \rangle \in \text{traces}(\Pi) \bullet \\ & \quad \sigma'(0) \vdash \sigma(i). \rho(x) \vee \\ & \quad \forall j > 0 \bullet \sigma'(j). \rho(x) = \sigma(i). \rho(x) \Rightarrow \sigma'(j). \rho(y) = \sigma(i). \rho(y) \}. \end{aligned}$$

Note that the clause $\sigma'(j). \rho(y) = \sigma(i). \rho(y)$ implies that $y \in \text{dom } \sigma'(j). \rho$.

If ys is a set of variables, then the abstract message $\text{bind}_x(ys)$ binds the values of all of the members of ys to x . The following semantic definition is an obvious extension of the one above.

$$\begin{aligned} \llbracket \text{bind}_x(ys) \rrbracket_{\Pi} = & \\ & \{ m \mid \forall tr \wedge \langle \sigma, i : \text{send } m \rangle \wedge tr' \wedge \langle \sigma' \rangle \in \text{traces}(\Pi) \bullet \\ & \quad \sigma'(0) \vdash \sigma(i). \rho(x) \vee \\ & \quad \forall j > 0 \bullet \sigma'(j). \rho(x) = \sigma(i). \rho(x) \Rightarrow \\ & \quad \quad \forall y \in ys \bullet \sigma'(j). \rho(y) = \sigma(i). \rho(y) \}. \end{aligned}$$

6.7.2 Proof rule

The following proof rule shows how *bind* can be used in annotations.

Annotation rule 19 (Bind.1)

$$a : \left\{ \begin{array}{l} true \\ \text{send } bind_x(y) \\ \left[\square \left(\begin{array}{l} honest(knows(x)) \Rightarrow \\ \forall b \in Var; B \in Val; Y \in Val^\perp \bullet \\ session(b \rightsquigarrow B; x \rightsquigarrow x, y \rightsquigarrow Y) \Rightarrow Y = y \end{array} \right) \right] \end{array} \right\}$$

Note that the postcondition implies that, assuming $honest(knows(x))$, if there is a $session(b \rightsquigarrow B; x \rightsquigarrow x)$, then in that session y is bound to a 's value of y .

Proof: Suppose

$$\begin{aligned} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \wedge e' \wedge es_1 \wedge e' \sqsupseteq \text{send } bind_x(y) \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \wedge es_1 \bullet true(\sigma)[i]. \end{aligned}$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the event corresponding to e' . Suppose $\sigma' \Longrightarrow \sigma''$. Let $X = \sigma(i).\rho(x)$ and $Y = \sigma(i).\rho(y)$. From the semantics of *bind*,

$$\begin{aligned} \sigma''(0) \vdash X \vee \\ \forall j > 0 \bullet \sigma''(j).\rho(x) = X \Rightarrow \sigma''(j).\rho(y) = Y. \end{aligned}$$

Suppose $honest(knows(x))(\sigma'')[i]$. Then $\sigma''(0) \not\vdash X$. Hence, for every b, B, Y' :

$$\begin{aligned} & session(b \rightsquigarrow B; x \rightsquigarrow X, y \rightsquigarrow Y')(\sigma'') \\ \Rightarrow & \langle \text{definition of } session; \text{ intruder doesn't know } X \rangle \\ & \exists j > 0 \bullet \sigma''(j).\rho(x) = X \wedge \sigma''(j).\rho(y) = Y' \\ \Rightarrow & \langle \text{from the above} \rangle \\ & Y' = Y. \end{aligned}$$

So we have shown

$$\begin{aligned} & honest(knows(x))(\sigma'')[i] \Rightarrow \\ & \forall b, B, Y' \bullet session(b \rightsquigarrow B; x \rightsquigarrow X, y \rightsquigarrow Y')(\sigma'') \Rightarrow Y' = Y \\ \equiv & \left(\begin{array}{l} honest(knows(x)) \Rightarrow \\ \forall b, B, Y' \bullet session(b \rightsquigarrow B; x \rightsquigarrow x, y \rightsquigarrow Y') \Rightarrow Y' = y \end{array} \right) (\sigma'')[i], \end{aligned}$$

for all σ'' such that $\sigma' \Longrightarrow \sigma''$. Hence

$$\square \left(\begin{array}{l} honest(knows(x)) \Rightarrow \\ \forall b, B, Y' \bullet session(b \rightsquigarrow B; x \rightsquigarrow x, y \rightsquigarrow Y') \Rightarrow Y' = y \end{array} \right) (\sigma')[i],$$

as required. □

The following rule is an easy extension of the rule above.

Annotation rule 20 (Bind.2)

$$a : \left\{ \begin{array}{l} true \\ \text{send } bind_x(y_1, \dots, y_k) \\ \left[\square \left(\begin{array}{l} honest(knows(x)) \Rightarrow \\ \forall b \in Var; B \in Val; Y_1, \dots, Y_k \in Val^\perp \bullet \\ session(b \rightsquigarrow B; x \rightsquigarrow x, y_1 \rightsquigarrow Y_1, \dots, y_k \rightsquigarrow Y_k) \Rightarrow \\ Y_1 = y_1 \wedge \dots \wedge Y_k = y_k. \end{array} \right) \right] \end{array} \right\}$$

6.7.3 Example refinements

As noted above, in an abstract message $bind_x(y)$, the value of x needs to be kept secret in order for useful deductions to be made. Further, x should normally be set to a fresh secret: clearly, a particular value for x cannot be used with two different values for y if the semantics is to be respected.

We believe that the following refinement laws hold, provided x is a fresh secret:

$$\begin{aligned} bind_x(y, z) &\sqsubseteq h(x, y, z), \\ bind_x(y, z) &\sqsubseteq \{y, z\}_x, \\ bind_x(y) \wedge bind_y(x) &\sqsubseteq h(x, y). \end{aligned}$$

In these laws, h represents a hash function; we intend to extend our semantic model to support hash functions in the future.

7 The Needham Schroeder Public Key Protocol

In this section we give a derivation of the Adapted Needham Schroeder Public-Key Protocol [Low95]. We give derivations of both sides of the protocol.

The protocol works by combining two one-way authentication tests, based on decryption of a public-key encrypted nonce, with identity information included to avoid man-in-the-middle attacks. The derivation demonstrates use of all the common abstract messages, most notably *bind*, which is used to ensure two pieces of data are received associated and that unassociated pieces of data cannot be mistaken for them.

The protocol makes use of public keys. Below, we will write pka and pkb for a 's and b 's public keys, and ska and skb for the corresponding secret keys, so $ska = pka^{-1}$ and $skb = pkb^{-1}$.

The protocol will make use of an invariant that says that a and b are distinct honest agents, and that only the appropriate agents know the secret keys:

$$I \triangleq a \neq b \wedge \text{honest}(a, b) \wedge \text{knows}(ska) = \{a\} \wedge \text{knows}(skb) = \{b\}.$$

Note that a 's state will include b 's public key pkb , but not his secret key skb ; recall that in this case the notation $\text{knows}(skb) = \{b\}$, in an annotation for a , means that only b knows the secret key corresponding to the public key held in pkb .

The first two conjuncts of the invariant are clearly invariant statements. The last two conjuncts will be maintained by all the sent messages containing *keepSecret*(ska, skb) clauses, which suffices by Annotation Rule 14; the receive messages maintain these conjuncts because of Annotation Rule 5.

We start by considering the perspective of agent a , as shown in Figure 7. The protocol proceeds as follows:

- a generates a new nonce na ;
- a sends a message from which only a holder of skb (i.e. b) can extract na , and which binds a to na ;
- a receives a message that proves knowledge of na , nb and b , from somebody in role b ; because only a and b know na , we can deduce that it must be b who has the corresponding session;
- finally a sends a message that keeps na secret (this message is mainly relevant to b from a security point of view).

Each assertion is justified in the annotation by reference to the relevant rule.

The protocol from the perspective of agent b is shown in Figure 8. This annotation uses the same invariant as for a , which is maintained for the same reasons. The protocol proceeds as follows:

- b receives any message; this step is necessary to fit in with a 's initial send;
- b generates a fresh nonce nb ;
- b sends a message from which only a holder of ska (i.e. a) can extract na and nb , and which binds b and na to nb ;
- finally b receives a message that proves knowledge of nb in role a ; because only a and b know na , we can deduce that it must be a who has the corresponding session.

We can strengthen the postconditions in the two annotations. Note that the annotation for a shows that $\text{session}(a; nb, na, b) \Rightarrow \text{knows}(na) \subseteq \{a, b\}$; hence we may add the conjunct $\text{knows}(na) \subseteq \{a, b\}$ to the postcondition in the annotation for b . Similarly, the annotation for b shows that $\text{session}(b; na, nb, a) \Rightarrow \text{knows}(nb) \subseteq \{a, b\}$; hence we may add the conjunct $\text{knows}(nb) \subseteq \{a, b\}$ to the postcondition in the annotation for b .

$$\begin{array}{l}
\{I\} \\
\text{new } na \\
\{I \wedge \text{knows}(na) = \{a\} \langle \text{Annotation Rule 7} \rangle\} \\
\text{send } \text{canExtract}_{skb}(na) \wedge \text{bind}_{na}(a) \wedge \text{keepSecret}(ska, skb) \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(na) \subseteq \{a, b\} \langle \text{Annotation Rule 13} \rangle \wedge \\
\Box \forall b', B, A \bullet \text{session}(b' \rightsquigarrow B; na \rightsquigarrow na, a \rightsquigarrow A) \Rightarrow A = a \\
\langle \text{Annotation Rule 19} \rangle
\end{array} \right\} \\
\text{receive } \text{provesKnowledgeOf}(na, nb, b, id = b) \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(na) \subseteq \{a, b\} \langle \text{Annotation Rule 5} \rangle \wedge \\
\exists B \bullet \text{session}(b \rightsquigarrow B; na, nb, b) \langle \text{Annotation Rule 16} \rangle \wedge \\
\forall b', B, A \bullet \text{session}(b' \rightsquigarrow B; na \rightsquigarrow na, a \rightsquigarrow A) \Rightarrow A = a
\end{array} \right\} \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(na) \subseteq \{a, b\} \wedge \text{session}(b; na, nb, a) \\
\langle \text{knows}(na) \subseteq \{a, b\} \text{ so } B \neq \text{intruder, above; predicate calculus} \rangle
\end{array} \right\} \\
\text{send } \text{keepSecret}(na) \wedge \text{keepSecret}(ska, skb) \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(na) \subseteq \{a, b\} \langle \text{Annotation Rule 14} \rangle \\
\wedge \text{session}(b; na, nb, a)
\end{array} \right\}
\end{array}$$
Figure 7: The Adapted Needham Schroeder Public Key Protocol: a 's perspective
$$\begin{array}{l}
\{I\} \\
\text{receive } any \\
\{I \langle \text{Annotation Rule 5} \rangle\} \\
\text{new } nb \\
\{I \wedge \text{knows}(nb) = \{b\} \langle \text{Annotation Rule 7} \rangle\} \\
\text{send } \text{canExtract}_{ska}(na, nb) \wedge \text{bind}_{nb}(b, na) \wedge \text{keepSecret}(ska, skb) \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(nb) \subseteq \{a, b\} \langle \text{Annotation Rule 13} \rangle \wedge \\
\Box \forall a', A', B, Na \bullet \\
\text{session}(a' \rightsquigarrow A'; nb \rightsquigarrow nb, b \rightsquigarrow B, na \rightsquigarrow Na) \Rightarrow \\
B = b \wedge Na = na \\
\langle \text{Annotation Rule 20} \rangle
\end{array} \right\} \\
\text{receive } \text{provesKnowledgeOfNR}(nb, id = a) \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(nb) \subseteq \{a, b\} \langle \text{Annotation Rule 5} \rangle \wedge \\
\exists A \bullet \text{session}(a \rightsquigarrow A; nb \rightsquigarrow nb) \wedge A \neq b \langle \text{Annotation Rule 18} \rangle \wedge \\
\forall a', A', B, Na \bullet \\
\text{session}(a' \rightsquigarrow A'; nb \rightsquigarrow nb, b \rightsquigarrow B, na \rightsquigarrow Na) \Rightarrow \\
B = b \wedge Na = na
\end{array} \right\} \\
\left\{ \begin{array}{l}
I \wedge \text{knows}(nb) \subseteq \{a, b\} \wedge \\
\text{session}(a; nb, na, b) \langle \text{knows}(nb) \subseteq \{a, b\} \text{ so } A = a \text{ above} \rangle
\end{array} \right\}
\end{array}$$
Figure 8: The Adapted Needham Schroeder Public Key Protocol: b 's perspective

Putting the two annotations together, we may refine the protocol to obtain the normal definition:

Message 1. $a \rightarrow b : \{a, na\}_{pkb}$

Message 2. $b \rightarrow a : \{b, na, nb\}_{pka}$

Message 3. $a \rightarrow b : \{nb\}_{pkb}$.

This step is justified by the following refinements of abstract messages:

$$\begin{aligned} & \left(\begin{array}{l} \text{canExtract}_{skb}(na) \wedge \text{bind}_{na}(a) \wedge \\ \text{keepSecret}(ska, skb) \wedge \text{any} \end{array} \right) \sqsubseteq \{a, na\}_{pkb}, \\ & \left(\begin{array}{l} \text{provesKnowledgeOf}(na, nb, b, id = b) \wedge \\ \text{canExtract}_{ska}(na, nb) \wedge \\ \text{bind}_{nb}(b, na) \wedge \text{keepSecret}(ska, skb) \end{array} \right) \sqsubseteq \{b, na, nb\}_{pka}, \\ & \left(\begin{array}{l} \text{keepsSecret}(na) \wedge \text{keepSecret}(ska, skb) \wedge \\ \text{provesKnowledgeOfNR}(nb, id = a) \end{array} \right) \sqsubseteq \{nb\}_{pkb}. \end{aligned}$$

It is worth considering how the development would proceed if we were developing the standard Needham Schroeder Public Key Protocol [NS78], which does not contain a b inside the encryption of message 2. In this case, in b 's annotation, the bind statement would be replaced by $\text{bind}_{nb}(na)$; the $B = b$ clauses are then removed from the subsequent assertions, and the final session assertion becomes $\text{session}(a; nb, na)$: in other words, b can be sure that a is running a session using the nonces nb and na , but cannot be sure that a associates that session with him; this corresponds to the well-known attack.

8 Tool support for creating concrete messages

We now consider tool support for our calculus. There are two aspects where tool support could usefully be provided: annotating a protocol, and finding concrete messages to implement the abstract messages used within the annotation. In this section we discuss just the latter; the former is left for future work.

This implementation has been developed concurrently to the semantic model described earlier, and so is, in places, slightly inconsistent with that model. Reuniting the two should prove straightforward.

The intention is to be able to translate a protocol annotation using abstract messages into a concrete implementation of a protocol. Typically, abstract messages comprise several conjuncts. However, we do not need to consider all the conjuncts simultaneously: the semantics of a conjunction is the intersection of the semantics of the conjuncts, so, at least in principle, we can form the set of refinements of each conjunct in turn, and then take the intersection.

Our implementation will not produce *every* possible implementation for a given abstract message: certain implementations will work only for very specific protocols. Instead, we will use implementations based on refinement rules that we believe hold for large classes of protocols.

So how can we produce sets of refinements? Each of our refinement rules will produce a family of similar implementations for an abstract message, which differ only by adding arbitrary fields to that refinement. For example, consider the following abstract message and its refinement:

$$\text{canExtract}_s(m) \sqsubseteq \{m\}_s.$$

There are many similar refinements, such as all the following:

$$\begin{aligned} \text{canExtract}_s(m) &\sqsubseteq \{m, x\}_s, \\ \text{canExtract}_s(m) &\sqsubseteq \{m\}_s, y, \\ \text{canExtract}_s(m) &\sqsubseteq \{m, x\}_s, y, \\ \text{canExtract}_s(m) &\sqsubseteq \{x, m, y\}_s, \end{aligned}$$

where x and y are suitable fields that do not break the requirements of the abstract message (we will describe how to enforce this later). We can represent this family using the notation $(\{m, _ \}_s, -)$, where each $_$ represents zero or more variables. Since these can be instantiated with an infinite number of different variables, we have created an infinite family of possible refinements.

8.1 A prolog representation for sets of messages

We have used prolog for the implementation: its built in unification will be used to form the intersections, meaning we don't have to deal explicitly with set operations.

We will represent sets of messages using open lists, i.e. lists that end with a variable that can be unified to an arbitrary value. For example, the family of messages, above, can be represented by the open list

```
[encryptable(s, [m|_]), _]
```

where `encryptable` is a constructor modelling encryption (defined below), and where each $_$ is an unnamed prolog variable, which can be unified with a list of zero or more protocol variables.

In fact, we can obtain further refinements by reordering the fields in a concatenation, or by repeating fields. We therefore take the above representation to include all messages formed by reordering or repetition of fields.

We begin by defining a function to close a list, i.e. remove a variable from the end of the list:

```
closeol([], []).
closeol([A|X], [A]) :- var(X), !.
closeol([A|X], [A|Y]) :- closeol(X, Y).
```

A closed version of an empty list in an empty list; a closed version of an open list that ends in a variable is simply the list up to that point; if the next element in the open list is not a variable, then the closed list is formed by including this element, and then closing the remainder of the list.

We now define a function `include` to add an element `A` to an open list, if it is not already there. For example, `include(a,B)` will return `[a|_]` for `B`.

```
include(A, B):- closeol(B,C), member(A,C).
include(A, B):- var(B),!,B=[A|_].
include(A, [C|B]) :- A\==C, include(A,B).
```

The first rule considers the case that `A` is already in the open list `B`; `B` is closed first (as `C`), to avoid unification, which would case the addition of `A` if it is not already there. The second rule states that if the list `B` is just a variable (contains no elements) then `B` should be the list starting with `A` and then being variable afterwards; the cut in the middle of these two statements ensures that even with backtracking for multiple answers, the list will not continually be extended by extending `B`. The third rule traverses the list ensuring that no match occurs (again to avoid backtracking problems and multiple inclusions); since the list is open, this will eventually append `A` to the end of the list, and then should terminate (thanks to the earlier cut).

We now consider our representation of concrete messages formed using more than just concatenation. We write `encryptable(k, m)` for an encryption of the form $\{m\}_k$; it should be read as a message which was originally created by someone in possession of `k`. We write `decryptable(k, m)` for a message that reveals `m` to anyone in possession of `k`, i.e. a message of the form $\{m\}_{k^{-1}}$. Finally, we write `hash(m)` for a hash of message `m`.

We will implement intersection of these families using prolog unification. For example if we define

```
example(A,B,R) :- include(A,R), include(B,R).
```

Then `example(a,b,R)` will return `[a,b,_]` for `R`.

8.2 Encoding refinement rules

Above we have shown how to represent a set of messages using a single message with placeholders, or unbound values, to show where further values can be substituted. Replacing any of these unbound values with semi-bound values, such as other representations of sets, will produce a new set. However, this new set might not meet the requirements placed upon the original set by the abstract message definition. Therefore, a list of restrictions is carried with the message; these are checked once the whole message has been constructed.

We will deal with values of the form `result(R,L)`, where `R` is a representation of a set of messages, as above, and `L` is a list of restrictions of the following forms:

`notpresent(X)` This represents that the atomic field `X` is not present anywhere in the message, other than possibly within hashes or as an encryption key (fields used in these latter ways could not be learnt, even by an agent who has learnt all keys).

`notderivable(X)` This represents that the atomic field `X` is not derivable from the other fields in the protocol.

In this section we give refinement rules to create these values; in the next section, we show how to test whether the candidate implementation `R` meets the restrictions in `L`.

We start with a rule corresponding to $canExtract_k(s)$; it produces refinements of the form $(\{s, -\}_{k^{-1}}, -)$:

```
canextract(K,S,result(R,L)) :- include(decryptable(K,R1), R),
                               include(S, R1),
                               include(notderivable(S), L),
                               include(notpresent(K), L).
```

The first two subgoals state that the result R must include a decryptable element, whose contents $R1$ can be revealed by possession of K , and such that $R1$ must include S ; note that the decryptable element may be part of a concatenation. However, these two subgoals are not enough, for they allow the outer concatenation to include any other elements, including K and S . We therefore require further restrictions, which are added to L , namely that S should not be derivable from the message, and that K should not be present.

We can implement *provesKnowledgeOf*(x) using messages of the form $(x, -)$:

```
provesKnowledgeOf(X,result(R,-)) :- include(X,R).
```

We provide two implementations of *bind*, corresponding to messages of the form $(h(s, v, -), -)$ and $(\{v, -\}_s, -)$:

```
bind(S,V,result(R,L)) :- include(hash(R1),R), include(S,R1),
                          include(V,R1), include(notderivable(S), L).
```

```
bind(S,V,result(R,L)) :- include(encryptable(S,R1),R), include(V,R1),
                          include(notderivable(S), L).
```

Note that both implementations include a restriction that S cannot be derived from the message.

The *keepSecret* message can be implemented by specifying that the relevant field is not present:

```
keepsecret(K,result(_,L)) :- include(notpresent(K), L).
```

8.3 The verification step

As described above, once a possible concrete implementation has been derived, it needs to be checked to ensure that any restriction placed on the message as a whole by the individual abstract components has not been broken by the additions of implementations for other abstract components; further we need to consider whether knowledge that might have been obtained by the intruder elsewhere in the protocol could mean that the proposed implementation fails.

The checking function, is built up in several stages:

```
check(result(R,L)) :- deriveTree(R, [], Tree),
                      determineAtoms(Tree, Atoms),
                      reduceDerivation(Tree, Derivable),
                      verify(L, reqResults(Atoms, Derivable), Output),
                      !,
                      Output = true.
```

The check function takes in a possible concrete implementation, R , and its associated list of restrictions, L , and will return if the verify function returns true for $Output$. A cut is included to ensure no back tracking, or rechecking of the verification (since the result of a verification should remain constant).

The verify function requires two lists as parameters:

- *Atoms* contains the atoms present in the concrete implementation, other than within hashes or as encryption keys; this is used to test *notpresent* restrictions;
- *Derivable* contains those atoms that can be learnt from the message itself; this is used to test *notderivable* restrictions.

In order to produce these lists, we begin by creating from the given concrete implementation a requirements tree: a tree showing which keys are required in order to decrypt components to reach a particular atom.

If other values are deemed public knowledge, these is another form of the check function that prepends public knowledge to the tree.

```

check(result(R,L),P) :- deriveTree(R, [], PreTree),
                        addPreknowledge(PreTree,P,Tree),
                        determineAtoms(Tree,Atoms),
                        reduceDerivation(Tree,Derivable),
                        verify(L,reqResults(Atoms,Derivable),Output),
                        !,
                        Output = true.

```

We explain each of the steps of the two check functions below.

DeriveTree The first step is to build the requirements tree. In fact, the “tree” is a list of terms of the form `requires(X,L)`, which represents that the list of keys `L` is required to reach the atom `X`. The function maintains a list `L` containing the keys required to reach the current message.

```

deriveTree([], _, []).
deriveTree([A|B], L, Tree) :- deriveTree(A, L, T1),
                              deriveTree(B, L, T2),
                              append(T1, T2, Tree).
deriveTree(decryptable(K,S), L, Res) :- append([K],L,N),
                                        deriveTree(S, N, Res).
deriveTree(encryptable(K,S), L, Res) :- append([K],L,N),
                                        deriveTree(S, N, Res).
deriveTree(hash(_), _, []).
deriveTree(X, L, [requires(X,L)]) :- atom(X).

```

The first rule states that an empty concrete implementation returns an empty tree. The second rule deals with concatenation: it processes each component and appends the results. The third and fourth rules deal with keys required to decrypt and encrypt messages respectively; they add the keys to the current requirements list, `L`, and continue with the contained messages; at the moment the implementation does not deal with asymmetric keys: in the `encryptable` case, the inverse key should really be used. The fifth rule deals with hash functions, which should not reveal anything about the contents. Finally the sixth rule concerns when an atom `X` is encountered; the requirements to this point, `L`, are the requirements to reach this atom, and so a term `requires(X,L)` is inserted into the final requirements tree.

addPreknowledge The `addPreknowledge` function adds the list of preknowledge to the tree, each with an empty list of requirements.

```

addPreknowledge(Req, [], Req).
addPreknowledge(Req, [A|B], [requires(A, []) | PreReq]) :-
    addPreknowledge(Req, B, PreReq).

```

determineAtoms The `determineAtoms` function simply extracts the atoms from the requirements tree.

```

determineAtoms([], []).
determineAtoms([requires(X,_)|B], [X|C]) :- determineAtoms(B,C).

```

reduceDerivation The function `reduceDerivation` uses the requirements tree to calculate the atoms that can be derived. This may require several iterations.

The main function `reduceDerivation` simply calls `recursiveReduction` with an empty accumulator:

```

reduceDerivation(D, Res) :- recursiveReduction(D, [], Res).

```

The function `recursiveReduction` takes a reduction tree, a set of elements already known, and a result list of all elements that can be learnt from the message.

```
recursiveReduction(D, Preknown, Preknown) :-
    findKnown(D, []).
recursiveReduction(D, Preknown, Res)      :-
    findKnown(D, Known),
    endIfAllKnown(D, Preknown, Known, Res).
```

This function uses two subsidiary functions (which we define later):

`findKnown` calculates all the atoms that can be produced immediately, i.e. that have an empty requirements list;

`endIfAllKnown` adds the items just learnt, `Known`, to those already known, `PreKnown`; removes all elements of `Known` from requirements lists; and recursively calls `recursiveReduction`.

The first rule for `recursiveReduction` says that if `findKnown` returns the empty list then the iteration must have finished (as nothing new has been learnt this step), and `recursiveReduction` should return the list of all the elements it has been able to derive so far. The second rule determines what can be learnt from the message using `findKnown`, and then calls `endIfAllKnown` to append the items now known to those previously known, and remove all mention of them from the derivation tree before recursing.

We illustrate the operation of this function using the following example, which requires three iterations:

1. `Preknown = []`
`Tree = [requires(a,[]), requires(b,[a]), requires(c,[b,d])]`
`Known = [a]`
2. `Preknown = [a]`
`Tree = [requires(b,[]), requires(c,[b,d])]`
`Known = [b]`
3. `Preknown = [a,b]`
`Tree = [requires(c,[d])]`
`Known = []`

The `findKnown` function simply returns a list of those atoms that have empty derivation lists:

```
findKnown([], []).
findKnown([requires(A,[])|B], [A|R]) :- findKnown(B, R).
findKnown([requires(_,D)|B], R)      :- D \= [], findKnown(B, R).
```

The function `endIfAllKnown` appends the new known values `Known` to the previously known values `Preknown` to produce `TotalKnown`; removes occurrences of elements of `Known` from the tree; and recursively calls `recursiveReduction`:

```
endIfAllKnown(_, Preknown, [], Preknown).
endIfAllKnown(D, Preknown, Known, Res) :-
    append(Preknown, Known, Totalknown),
    reduceUsingKnown(D, Known, ReducedD),
    recursiveReduction(ReducedD, Totalknown, Res).
```

The function `reduceUsingKnown` runs through the list of new known values, removing each in turn using `removeRequirement`:

```

reduceUsingKnown(D, [], D).
reduceUsingKnown(D, [A|B], ReducedD) :-
    removeRequirement(D, A, IntermediateD),
    reduceUsingKnown(IntermediateD, B, ReducedD).

```

The function `removeRequirement` takes a requirement tree, and a single value `A`, and removes `A` from each element of the tree:

```

removeRequirement([], _, []).
removeRequirement([requires(A,_)|D], A, R) :-
    removeRequirement(D, A, R).
removeRequirement([requires(B,C)|D], A, [requires(B,NewC)|R]) :-
    removeItem(A,C,NewC), removeRequirement(D, A, R).

```

The second rule says that if `A` appears as the first argument of a `requires`, then that item can be removed, as the atom has now been dealt with. The third rule deals with the case of an atom `B` with requirements `C`: `removeItem` is called to remove `A` from `C` to produce the new requirements `NewC`.

Finally `removeItem` removes an item from a list:

```

removeItem(_, [], []).
removeItem(A, [A|B], D) :- removeItem(A, B, D).
removeItem(A, [C|B], [C|D]) :- removeItem(A, B, D).

```

Verify Recall that `verify` takes a list of restrictions, lists of atoms and derivable atoms, and tests whether all the restrictions are satisfied.

```

verify([],_, true).
verify([notderivable(K)|_], reqResults(_, Derivable), false) :-
    member(K, Derivable).
verify([notderivable(_)|B], R, Res) :- verify(B, R, Res).
verify([notpresent(K)|_], reqResults(Present, _), false) :-
    member(K, Present).
verify([notpresent(_)|B], R, Res) :- verify(B, R, Res).

```

The first rule returns `true` if all the restrictions have been checked. The second rule returns `false` if there is a restriction that `K` should not be derivable, yet `K` is derivable. The third rule says that if the second rule has not triggered, then we should move onto the next restriction. The fourth and fifth rules are similar to the second and third, except concerning where an atom should not be present.

8.4 Examples

We begin with a simple example, corresponding to the abstract message $canExtract_k(s) \wedge canExtract_k(t)$:

```

example1(K,S,T,R) :- canExtract(K, S, R),
                    canExtract(K, T, R),
                    check(R).

```

This is then run by asking for `example1(k,s,t,R)`, which returns all possible concrete refinements together with their restrictions. This produces many results, most of which are duplicates (due to the way concrete messages are constructed). The individual different results are displayed below; for conciseness, the restrictions are omitted:

```

[decryptable(k, [s,t])]
[decryptable(k, [s]),decryptable(k, [t])]

```

These correspond to the concrete messages $\{s, t\}_k$ and $\{s\}_k, \{t\}_k$.

We now try to produce concrete messages to refine the abstract messages for the Needham Schroeder Public Key protocol³.

The first message is produced as follows:

```
message1(SKA,SKB,NA,NB,A,R) :- canextract(SKB,NA,R),
                               bind(NA,A,R),
                               keepsecret(SKA,R),
                               keepsecret(SKB,R),
                               keepsecret(NB,R),
                               check(R, [A]).
```

which when run, returns the following possibilities:

```
[decryptable(skb, [na]), hash([na, a | _])]
[decryptable(skb, [na]), encryptable(na, [a])]
```

corresponding to $(\{na\}_{pkb}, hash(na, a, -))$ and $(\{na\}_{pkb}, \{na\}_a)$.

The second abstract message includes the requirement $bind_{nb}(b, na)$; this is implemented below by requiring NB to be bound to a message T that includes B and NA:

```
message2(SKA,SKB,NA,NB,B,R) :- canextract(SKA,NB,R),
                               canextract(SKA,NA,R),
                               include(B,T),
                               include(NA,T),
                               bind(NB,T,R),
                               keepsecret(SKA,R),
                               keepsecret(SKB,R),
                               provesKnowledgeOf(NA,B,R),
                               check(R, [B]).
```

After a long search, prolog simply returns no to this request. The reason is that the only implementation we have allowed for the $provesKnowledgeOf(NA,B,R)$ abstract message, namely NA, does not satisfy the requirement of $canextract(SKA,NA,R)$. The obvious solution would be to provide more implementations for $provesKnowledgeOf$.

Finally the third message can be produced as follows:

```
message3(SKA,SKB,NA,NB,A,R) :- provesKnowledgeOfNR(NB,A,R),
                               keepsecret(SKA,R),
                               keepsecret(SKB,R),
                               keepsecret(NA,R),
                               check(R, [A]).
```

which returns the following result

```
[decryptable(a, [nb])]
```

³ The precise abstract messages considered here differ slightly from those considered in Section 7.

9 Conclusions

9.1 Summary

We have created a calculus for protocol development, based upon the idea of annotating protocols: we add assertions to the protocol description, stating properties that will be true when that point in the protocol is reached. A novel feature of our calculus is the idea of abstract messages, which state what a message is intended to achieve, rather than giving a concrete implementation.

We have presented proof rules that can be used to justify assertions, and refinement rules that allow abstract messages to be implemented. We have produced a semantic model, and used it to formalise the meaning of annotations, and to verify annotation rules; we have proved theorems that we believe will provide the underpinnings to verify message refinement rules. We have illustrated the calculus by using it to develop the Adapted Needham Schroeder Public Key Protocol.

9.2 Related Work

There have been a few previous studies of protocol synthesis and composition.

Hassen Saïdi [Sai02] investigated the synthesis of protocols from a specification based on BAN Logic; he derived the Needham-Schroeder Public Key protocol by applying simple inference rules.

Datta, Derek, Mitchell and Pavlovic [DDMP03] investigated the derivation of protocols from smaller, well-used ideas, such as Diffie-Hellman key exchange, nonces and certificates, and derived three different (pre-existing) forms of key authentication from two simple protocol forms. The formal logic used in this paper is based upon the work of Durgin, Mitchell and Pavlovic [DMP01] which introduced the cord calculus, designed to allow composition of protocols.

Canetti and Krawczyk [CK02] also developed a composable notion of key exchange leading to secure channels; this allows for individual components such as key exchange to be separated from a single protocol, and so be reused by many protocols. This work, along with others, is built upon the work of Mateus, Mitchell and Scedrov [MMS] who have developed a probabilistic polynomial-time process calculus to derive compositionality properties of protocols.

9.3 Future Work

The main part of the calculus that needs further work is the development of message refinement rules. We intend to produce and verify rules of the form alluded to earlier. We expect that Theorems 1 and 2 will prove very useful in the verification: the theorems relate closely to the *provesKnowledgeOf* and *bind* macros, and the *canExtract* and *keepSecret* macros, respectively.

We also want to extend our semantic model to deal with cryptographic hash functions, and then produce message refinement rules that produce hashed messages; we expect this to be reasonably straightforward, as hashes are rather like encryption with a key whose inverse is not known.

We also intend to undertake more case studies in protocol development. These case studies will help us to identify additional useful abstract messages, together with their associated proof rules; they will also help us to develop techniques and experience, showing the best way to approach a protocol development. A goal would be to produce developments of a significant number of protocols, perhaps most of those from Clark and Jacob's library [CJ97].

Much needs to be done on the tool support for refining abstract messages. It needs to be made consistent with the current version of the semantic model. As new abstract messages and refinement rules are developed, they need to be included in the implementation. Another enhancement to this implementation would be a method of encoding an entire protocol into prolog at once, rather than in a message by message fashion; this should carry across derivable information from one message to preknowledge in the next. Further, full support for asymmetric cryptography should be added.

Further, we would like to provide tool support for the process of annotating a protocol.

An ambitious, and slightly speculative, long term goal would be to develop a system for calculating and composing protocols on the fly: the idea would be to develop an evolving protocol, the form of

which is sent down a less advanced protocol, and automatically checked, compiled and appended to the currently running protocol. These are of course ambitious goals, but the calculus proposed here suggests a starting point.

Acknowledgements

We would like to thank Michael Goldsmith, Bill Roscoe and Sadie Creese for helpful comments on this work.

References

- [BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, 1989.
- [CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0. Available via <http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz>, 1997.
- [CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *Theory and Application of Cryptographic Techniques*, pages 337–351, 2002.
- [Coh00] Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 144–158, 2000.
- [DDMP03] A. Datta, A. Derek, J. Mitchell, and D. Pavlovic. A derivation system for security protocols and its logical formalization. In *Proceedings of The 16th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2003.
- [DMP01] Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for proving security properties of protocols. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 241–255, 2001.
- [DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.
- [GNY90] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about belief in cryptographic protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.
- [GTF00] Joshua Guttman and Javier Thayer Fábrega. Protocol independence through disjoint encryption. In *Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.
- [HLS03] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.
- [Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996. Also in *Software—Concepts and Tools*, 17:93–102, 1996.
- [Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.
- [MCJ97] Will Marrero, Edmund Clarke, and Somesh Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Available via URL <http://dimacs.rutgers.edu/Workshops/Security/program2/program.html>.

- [Mea96] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
- [MMS] P. Mateus, J. Mitchell, and A. Scedrov. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus.
- [MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Mur ϕ . In *IEEE Symposium on Security and Privacy*, 1997.
- [NS78] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
- [Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
- [Sai02] Hassen Saidi. Towards automatic synthesis of security protocols. In *In Logic-Based Program Synthesis Workshop, AAAI 2002 Spring Symposium*, 2002.
- [SBP01] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1, 2):47–74, 2001.
- [THG99] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2, 3):191–230, 1999.