

forward

a future of reliable wireless ad hoc networks of roaming devices

Towards a Calculus for Security Protocol Synthesis

6th January 2004

Record of Changes

Date	Version	Comment
6-1-2004	1.0	First Issue

Authorisation

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

Date: 6th January 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 D7 of the FORWARD project. It is the second 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 the work conducted on Task 1.3 *Synthesis and Composition of Security Protocols* in the second six months of research.

The research presented here provides the initial steps 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. The calculus is illustrated with the development of a small example protocol.

Contents

1	Introduction	1
2	Example	3
2.1	First steps	3
2.2	Creating nonces	4
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	8
3	State and protocol semantics	10
3.1	Local states	10
3.2	Global states	11
3.2.1	<i>knows</i>	11
3.2.2	<i>session</i>	11
3.2.3	<i>honest</i>	11
3.3	Protocol semantics	11
4	Abstract messages	13
4.1	Concrete messages	13
4.1.1	Semantics	14
4.2	Conjunction	14
4.3	<i>canExtract</i>	14
4.3.1	Semantics	15
4.3.2	Proof rules	15
4.3.3	Example refinements	15
4.4	<i>bind</i>	15
4.4.1	Semantics	16
4.4.2	Proof rule	16
4.4.3	Example refinements	16
4.4.4	Note	16
4.5	<i>provesKnowledgeOf</i>	16
4.5.1	Semantics	16
4.5.2	Proof rule	17
4.5.3	Example refinements	17
4.6	<i>provesKnowledgeOfNR</i>	17
4.6.1	Semantics	17
4.6.2	Proof rule	17
4.7	<i>keepSecret</i>	17
4.7.1	Semantics	18
4.7.2	Proof rule	18
5	Conclusions	19
5.1	Summary	19
5.2	Related Work	19
5.3	Future Work	19

This page is intentionally blank

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 vital difference 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. For this we provide a set of rules for the refinement of abstract messages, as a set of protocol-independent refinements that can always be used.

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 study abstract messages in more detail in Section 4: we present a semantic definition for each abstract message, rules to allow the abstract messages to be used in annotations, and rules to refine the abstract messages to concrete messages. Finally, in Section 5, 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 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.

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

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 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 *could* know the piece of data x at this point in the protocol, on the assumption that they have overheard all messages sent on the network, and performed all possible decryptions, etc. 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 *does* have 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.

The difference between *knows* and *session* is that *knows* specifies all participants who could possibly know a value. For instance, suppose a participant a sends a value x with b as the intended recipient; b may not receive the message, but there is the possibility that he did, and so *could* know x . At this point $b \in \text{knows}(x)$ but without confirmation of knowledge, it cannot be confirmed by a that $\text{session}(b; x)$.

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.

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

Figure 2: Protocol annotation showing nonce creation

The $\text{new}(na)$ 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 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:

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

provided x is not free in P .

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

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

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:

$$a \rightarrow : m$$

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 by, and 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 $\{a, b\}$.

$$\begin{aligned}
& \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
& \{ I \} \\
& \quad a \quad : \text{new}(na) \\
& \{ I \wedge \text{knows}(na) = \{a\} \} \\
& \quad a \rightarrow \quad : \text{canExtract}_k(na) \wedge \text{keepSecret}(k) \\
& \{ I \wedge \text{knows}(na) = \{a\} \cup \text{knows}(k) \} \\
& \{ I \wedge \text{knows}(na) = \{a, b\} \} \\
& \dots \\
& \{ I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na) \}
\end{aligned}$$

Figure 3: Annotation including nonce challenge

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 $I \wedge \text{knows}(na) = \{a\} \cup \text{knows}(k)$ holds: the $\text{keepSecret}(k)$ message requires that $\text{knows}(k)$ after 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 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 $I \wedge \text{knows}(na) = \{a, b\}$. 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:

$$\rightarrow a : m$$

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.

$$\begin{aligned}
 & \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
 & \left\{ I \right\} \\
 & \quad a \quad : \text{new}(na) \\
 & \left\{ I \wedge \text{knows}(na) = \{a\} \right\} \\
 & \quad a \rightarrow \quad : \text{canExtract}_k(na) \wedge \text{keepSecret}(k) \\
 & \left\{ I \wedge \text{knows}(na) = \{a\} \cup \text{knows}(k) \right\} \\
 & \left\{ I \wedge \text{knows}(na) = \{a, b\} \right\} \\
 & \quad \rightarrow a \quad : \text{provesKnowledgeOfNR}(na) \wedge \text{keepSecret}(k) \wedge \text{keepSecret}(na) \\
 & \left\{ I \wedge \text{knows}(na) = \{a, b\} \wedge \exists b' \bullet \text{session}(b'; na) \wedge b' \neq a \right\} \\
 & \left\{ I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na) \right\}
 \end{aligned}$$

Figure 4: Completed protocol annotations

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.

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 $\text{knows}(na) = \{a, b\}$), that the person who has na in their state must be 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:

$$\frac{m_1 \sqsubseteq m \quad m_2 \sqsubseteq m}{m_1 \wedge m_2 \sqsubseteq m}$$

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

$$\text{provesKnowledgeOfNR}(na) \wedge \text{keepSecret}(k) \wedge \text{keepSecret}(na) \sqsubseteq h(na, b)$$

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

$$\text{provesKnowledgeOfNR}(na) \wedge \text{keepSecret}(k) \wedge \text{keepSecret}(na) \sqsubseteq \{b\}_{na}$$

This gives us two possible, but equally valid, verified concrete protocols, given in Figures 5 and 6.

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

Figure 5: Concrete implementation 1

$$\begin{aligned}
& \text{Invariant } I \triangleq \text{knows}(k) = \{a, b\} \wedge a \neq b \wedge \text{honest}(a, b) \\
& \left\{ I \right\} \\
& \quad a \quad : \text{new}(na) \\
& \left\{ I \wedge \text{knows}(na) = \{a\} \right\} \\
& \quad a \rightarrow \quad : \{na\}_k \\
& \left\{ I \wedge \text{knows}(na) = \{a\} \cup \text{knows}(k) \right\} \\
& \left\{ I \wedge \text{knows}(na) = \{a, b\} \right\} \\
& \quad \rightarrow a \quad : \{b\}_{na} \\
& \left\{ I \wedge \text{knows}(na) = \{a, b\} \wedge \exists b' \bullet \text{session}(b'; na) \wedge b' \neq a \right\} \\
& \left\{ I \wedge \text{knows}(na) = \{a, b\} \wedge \text{session}(b; na) \right\}
\end{aligned}$$

Figure 6: Concrete implementation 2

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.

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

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

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

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

$$\text{Message } \beta.2 \quad b \rightarrow a \quad : na$$

There is an obvious interaction between these protocols:

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

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

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

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.

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 message and the same 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 State and protocol semantics

We will require semantics for the various features of the calculus, in order to make clear the meaning of constructs, and to verify the proof rules. We outline the semantics in this section. Note that this model is still work in progress: the precise form of the model has not been decided; the semantic definitions of state predicates are preliminary; and the proof rules have not all been verified. In this section we proceed by outlining semantics for local states, global states, events, traces and protocols.

3.1 Local states

We need some way to model the *local state* of honest agents, i.e. show precisely what variables are associated with what values. Once we can make statements about local states, then when looking at both sides of the protocol we should be able to match up the states and ensure that they are consistent with each other. We will model each local state by a mapping from variable names to values.

There are two notions here, one of variables, and the other of values. Consider the following simplistic protocol:

Message 1 $a \rightarrow b : x, y$

This states that both a and b possess variables named x and y . a sends the values that he has contained within x and y , and b upon receiving this message, will take the two values from the message, and place them in x and y .

The values get indexed in the participant's state by the names of the variables they are supposed to represent. This means, however, that a and b could conceivably think very different things. The following shows how important the difference between variables and values is when an intruder I is able to masquerade as any participant x (denoted as I_x):

Message 1 $a \rightarrow I_b : X, Y$

Message 1' $I_a \rightarrow b : Z, X$

At the end of the example, a thinks that the value X is in the variable x , and Y is in y . However, b has a state in which the value X is in variable y , and the value Z is in variable x . This obviously can have drastic consequences on further communications between a and b , especially if one of the values stored in the variables is security sensitive and performing operations on it (such as signing it) may break the protocol.

We represent local states as mappings from variables to values. We will use the notation $b : \langle x \rightsquigarrow X, y \rightsquigarrow Y, z \rightsquigarrow \perp \rangle$ to represent that b has a state in which X is assigned to the variable x , the value Y is assigned to the variable y , and variable z has not yet been assigned to: b may or may not have other variables bound to other values.

In the above example the following bindings would exist:

$a : \langle x \rightsquigarrow X, y \rightsquigarrow Y \rangle$

$b : \langle x \rightsquigarrow Z, y \rightsquigarrow X \rangle$

It is important to note that once a variable has had a value bound to it, whenever that variable is received again, its value must be checked with that stored in the state. This means that a variable cannot ever be rebound during a single run of a protocol. This check should be carried out by all participants when receiving information that they already possess. In our model, all messages received have an implicit post-condition which states the value received is the same as that already bound in the state.

The local state of the intruder is much simpler: it is simply the set of messages that the intruder knows, i.e., those messages that he can produce from the messages he knew initially and those he has seen on the network, by encrypting or decrypting with known keys, by concatenating or splitting pairs, or by hashing.

3.2 Global states

A *global state* is a collection of local states of honest agents, together with the state of the intruder. We typically denote global states by σ , σ' , etc.

When using macro constructs such as $knows(x)$ or $session(a; x)$ about a particular global state σ , the construct is subscripted with the state (e.g. $knows_\sigma(x)$). When the global state in question is obvious from context, for instance within protocol annotations, the state subscript is omitted.

We describe below the constructs used for specifying states.

3.2.1 *knows*

$knows_\sigma(x)$ returns the set of participants who *could* know all data items in the set x had they received all messages sent in the execution of the protocol up to state σ . The results of the function in one state cannot be relied upon to stay the same in the next state, since the next event may transmit information even if it outside of the perspective in question. It should be noted that agents cannot be removed from the set, nor can information be forgotten, meaning that the set $knows(x)$ can only grow from state to state.

3.2.2 *session*

We assume that each local state of an honest agent has a distinguished variable representing that agent's identity. If B is an honest agent then the notation $session_\sigma(b \rightsquigarrow B; x \rightsquigarrow X, y \rightsquigarrow Y)$ means that the identity variable is b , and $B : \langle b \rightsquigarrow B, x \rightsquigarrow X, y \rightsquigarrow Y \rangle$ in σ , i.e. B has a session playing the role represented by variable b , and using the values X and Y for x and y (and possibly has other variables bound). If B is dishonest then the notation means that $B \in knows_\sigma(X) \cap knows_\sigma(Y)$, i.e. that B knows X and Y : a dishonest agent is not forced to bind values to variables in any predictable way.

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. For example, $session_\sigma(b; x, y \rightsquigarrow Y)$ means $b : \langle b \rightsquigarrow b, x \rightsquigarrow x, y \rightsquigarrow Y \rangle$ in σ .

Note that $session_\sigma(a; x)$ implies that $a \in knows_\sigma(x)$.

3.2.3 *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. In most annotations, we will assume that $honest(X)$ holds as an invariant where X is the set of all participants involved. (Being an invariant, it is true at all states, of the protocol and hence we shall omit the state identifier from the notation.)

3.3 Protocol semantics

We consider three types of *events* performed by protocol participants:

send The event $a : \text{send } m$ represents agent a sending message m ;

receive The event $a : \text{receive } m$ represents agent a receiving message m ;

new The event $a : \text{new } n$ represents agent a freshly generating the value n .

A *system trace* is an alternating sequence of the form $\langle \sigma_0, e_1, \sigma_1, e_2, \sigma_2, \dots \rangle$, where each σ_i is a global state, and each e_i is an event. This trace represents a protocol run in which the initial state is σ_0 , then event e_1 occurs and the state evolves into σ_1 , and so on.

The semantics of a protocol is then the set of all traces that can be observed of it. We leave as future work formally defining the traces of a given protocol.

We say that an annotation is correct if every assertion is true whenever the participant in question reaches the appropriate point in the protocol.

For the sake of completeness, we give below the proof rules about annotations that we presented earlier.

Proof rule 1 (Strengthen precondition)

$$\frac{\begin{array}{l} \{pre\}e\{post\} \\ pre' \Rightarrow pre \end{array}}{\{pre'\}e\{post\}}$$

Proof rule 2 (Weaken postcondition)

$$\frac{\begin{array}{l} \{pre\}e\{post\} \\ post \Rightarrow post' \end{array}}{\{pre\}e\{post'\}}$$

Proof rule 3 (Sequential composition)

$$\frac{\begin{array}{l} \{pre\}e_1\{mid\} \\ \{mid\}e_2\{post\} \end{array}}{\{pre\}e_1e_2\{post\}}$$

Proof rule 4 (New)

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

provided x is not free in P .

We also give a rule concerning conjunctions of postconditions.

Proof rule 5 (Conjunction of postconditions)

$$\frac{\begin{array}{l} \{pre\}e\{post_1\} \\ \{pre\}e\{post_2\} \end{array}}{\{pre\}e\{post_1 \wedge post_2\}}$$

4 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 the conjunction of abstract messages, and concrete 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.

The idea behind abstract messages is that most protocol designers know what they are trying to achieve in the remote state of the participant that the message is intended for, but have to write concrete messages 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 define the semantics of an abstract message to be the set of all the concrete messages 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 messages) for that abstract message. We write $\llbracket m \rrbracket_{\Pi}$ for the semantics of abstract message m in protocol Π .

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

$$m \sqsubseteq_{\Pi} m' \Leftrightarrow \llbracket m \rrbracket_{\Pi} \supseteq \llbracket m' \rrbracket_{\Pi}$$

and define a protocol-independent refinement by

$$m \sqsubseteq m' \Leftrightarrow \forall \Pi \bullet m \sqsubseteq_{\Pi} m'$$

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 1 *Refinement is a preorder.*

The following rules show how refinement can be used within annotations.

Proof rule 6 (Refine sent message)

$$\frac{\begin{array}{l} \{pre\}a \rightarrow : m\{post\} \\ m \sqsubseteq_{\Pi} m' \end{array}}{\{pre\}a \rightarrow : m'\{post\}}$$

Proof rule 7 (Refine received message)

$$\frac{\begin{array}{l} \{pre\} \rightarrow b : m\{post\} \\ m \sqsubseteq_{\Pi} m' \end{array}}{\{pre\} \rightarrow b : m'\{post\}}$$

4.1 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.

4.1.1 Semantics

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

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

4.2 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:

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

The following lemma follows directly from the definition.

Lemma 2 *Conjunction represents the least upper bound relation with respect to refinement.*

The following two rules relate conjunction to refinement.

Proof rule 8 (Refinement by conjunction)

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

Proof rule 9 (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.

Proof rule 10 (Conjunction of sent messages)

$$\frac{\begin{array}{l} \{pre\}a \rightarrow : m_1\{post_1\} \\ \{pre\}a \rightarrow : m_2\{post_2\} \end{array}}{\{pre\}a \rightarrow : m_1 \wedge m_2\{post_1 \wedge post_2\}}$$

Proof rule 11 (Conjunction of received messages)

$$\frac{\begin{array}{l} \{pre\} \rightarrow b : m_1\{post_1\} \\ \{pre\} \rightarrow b : m_2\{post_2\} \end{array}}{\{pre\} \rightarrow b : m_1 \wedge m_2\{post_1 \wedge post_2\}}$$

It is worth considering the case where $\llbracket m_1 \wedge m_2 \rrbracket_{\Pi} = \emptyset$, 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.

4.3 canExtract

The abstract message $canExtract_k(x)$ means that only agents possessing k should be able to recover x from the message. Note that this makes no mention of the recoverability of the key k : if it is required that no agent should learn k from the message, then this requirement should be added as an extra conjunct. Note that the recipient of a $canExtract$ message has no knowledge concerning who created it.

4.3.1 Semantics

The semantics of $canExtract_k(x)$ is the set of messages m such that for all traces in which a sends this message: (1) any agent that knows x afterwards must have known x before or known k before; and (2) any agent who knew k before could know x afterwards.

$$\begin{aligned} \llbracket canExtract_k(x) \rrbracket_{\Pi} = & \{ m \mid \forall tr \wedge \langle \sigma, a : \text{send } m, \sigma' \rangle \in traces(\Pi) \bullet \\ & (\forall b \in knows_{\sigma'}(x) \bullet b \in knows_{\sigma}(k) \vee b \in knows_{\sigma}(x)) \wedge \\ & (\forall b \in knows_{\sigma}(k) \bullet b \in knows_{\sigma'}(x)) \} \end{aligned}$$

Here and subsequently we combine quantifiers (\forall, \exists) with pattern matching: the above universal quantification ranges over all traces of the given form.

The predicate $canExtract_k(X)$ is an extension of the standard $canExtract$, which specifies who can extract a set of data items X . The semantics are defined as:

$$\llbracket canExtract_k(X) \rrbracket_{\Pi} = \bigcap_{x \in X} \llbracket canExtract_k(x) \rrbracket_{\Pi}$$

4.3.2 Proof rules

The following proof rules show how $canExtract$ can be used in annotations.

Proof rule 12 (CanExtract.1)

$$\begin{aligned} & \{ knows(m) = s_m \} \\ & a \rightarrow : canExtract_k(m) \\ & \{ knows(m) = s_m \cup knows(k) \} \end{aligned}$$

Proof rule 13 (CanExtract.2)

$$\begin{aligned} & \{ knows(m) = s_m \} \\ & \rightarrow b : canExtract_k(m) \\ & \{ knows(m) = s_m \cup knows(k) \} \end{aligned}$$

Proof rule 14 (CanExtract.3)

$$\begin{aligned} & \{ session(b; k) \} \\ & \rightarrow b : canExtract_k(m) \\ & \{ session(b; k, m) \} \end{aligned}$$

4.3.3 Example refinements

We give here some examples showing how $canExtract$ can be refined by concrete messages.

$$\begin{aligned} canExtract_s(x) & \sqsubseteq \{x\}_s \\ canExtract_s(x, y) & \sqsubseteq \{x\}_s, \{y\}_s \\ canExtract_s(x, y) & \sqsubseteq \{x, y\}_s \end{aligned}$$

4.4 bind

The abstract message $bind_s(x)$ represents the requirement that s and x should be bound together. More precisely, it means that no agent can replace x with a new value within the message unless they possess s .

4.4.1 Semantics

The semantics of $bind_s(x)$ is the set of messages m such that if participant a can send m , and can also send m with the value of x changed, then a must have known the secret s before-hand.

$$\llbracket bind_s(x) \rrbracket_{\Pi} = \{m \mid \forall tr \wedge \langle \sigma, a : \text{send } m \rangle \in \text{traces}(\Pi) \bullet \forall x' \neq x \bullet \\ tr \wedge \langle \sigma, a : \text{send } m[x'/x] \rangle \in \text{traces}(\Pi) \Rightarrow a \in \text{knows}_{\sigma}(s)\}$$

Note that the semantics effectively requires every binding to use a fresh secret: if the secret is reused, then a third party could capture two instances of the message, and substitute one for the other without knowing the secret, thus breaking the semantics.

The abstract message $bind_s(\underline{x})$ is an extension of the standard $bind$, that binds all the data items within the list \underline{x} to the secret s , and hence binds them to each other. The semantics are defined as:

$$\llbracket bind_s(\underline{x}) \rrbracket_{\Pi} = \{m \mid \forall tr \wedge \langle \sigma, a : \text{send } m \rangle \in \text{traces}(\Pi) \bullet \forall \underline{x}' \neq \underline{x} \bullet \\ tr \wedge \langle \sigma, a : \text{send } m[\underline{x}'/\underline{x}] \rangle \in \text{traces}(\Pi) \Rightarrow a \in \text{knows}_{\sigma}(s)\}$$

4.4.2 Proof rule

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

Proof rule 15 (Bind)

$$\left\{ \begin{array}{l} \text{session}(b; x) \wedge b \in \text{knows}(s) \\ \rightarrow b : bind_s(x) \\ \left\{ \text{session}(b; x, s) \wedge \exists a \bullet \text{session}(a; x, s) \right\} \end{array} \right\}$$

4.4.3 Example refinements

We give here some examples showing how $bind$ can be implemented using concrete messages.

$$\begin{aligned} bind_s(x, y) &\sqsubseteq h(k, x, y) \\ bind_s(x, y) &\sqsubseteq \{x, y\}_k \\ bind_x(y) \wedge bind_y(x) &\sqsubseteq h(x, y) \end{aligned}$$

4.4.4 Note

Note that $bind_s(x, z) \not\sqsubseteq bind_s(x, y) \wedge bind_s(y, z)$: the right hand side could be refined to $h(s, x, y), h(s, y, z)$, which is not a valid refinement of the left hand side, because the two components could have been replayed from different runs of the protocol by somebody who does not know s .

4.5 provesKnowledgeOf

The abstract message $provesKnowledgeOf(x)$ proves to the recipient of the message that someone knows the value x . $provesKnowledgeOf$ specifies nothing about who may learn data from this message.

4.5.1 Semantics

The semantics of $provesKnowledgeOf(x)$ is the set of messages m that if received by some agent b , means that in some earlier state σ there was an agent a who knew x and who sent a message m' . This allows the receiver to verify state information about the sender, concerning the variable x .

$$\llbracket provesKnowledgeOf(x) \rrbracket_{\Pi} = \\ \{m \mid \forall tr \wedge \langle b : \text{receive } m \rangle \in \text{traces}(\Pi) \bullet \\ \exists a \bullet \exists tr' \wedge \langle \sigma, a : \text{send } m' \rangle \leq tr \bullet \text{session}_{\sigma}(a; x)\}$$

Note that in the semantics, m and m' may not be the same message: a third party could change the message in some unimportant way between sending and receipt; future work will investigate if there is some link between the messages.

4.5.2 Proof rule

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

Proof rule 16 (ProvesKnowledgeOf)

$$\begin{array}{l} \{ b \in \text{knows}(x) \} \\ \rightarrow b : \text{provesKnowledgeOf}(x) \\ \{ \exists a \bullet \text{session}(a; x) \} \end{array}$$

4.5.3 Example refinements

We give here some examples to show how *provesKnowledgeOf* can be implemented by concrete messages.

$$\begin{array}{l} \text{provesKnowledgeOf}(x) \sqsubseteq x \\ \text{provesKnowledgeOf}(x) \sqsubseteq h(x) \\ \text{provesKnowledgeOf}(x) \sqsubseteq s, \{s\}_x \end{array}$$

4.6 *provesKnowledgeOfNR*

The abstract message *provesKnowledgeOfNR*(x) is almost identical to *provesKnowledgeOf*(x), apart from it ensures the very important property that the message has not been reflected. This means that the message will not be accepted by the person who created it, and allows the receiver to deduce that the participant who knows the knowledge is not, in fact, themselves. This is a more useful construction than *provesKnowledgeOf*(x) and will often be used instead of it.

4.6.1 Semantics

The semantics are almost identical to those for *provesKnowledgeOf*(x), except that the receiver is not equal to the sender.

$$\begin{aligned} \llbracket \text{provesKnowledgeOfNR}(x) \rrbracket_{\Pi} = \\ \{ m \mid \forall tr \wedge \langle b : \text{receive } m \rangle \in \text{traces}(\Pi) \bullet \\ \exists a \neq b \bullet \exists tr' \wedge \langle \sigma, a : \text{send } m' \rangle \leq tr \bullet \text{session}_{\sigma}(a; x) \} \end{aligned}$$

4.6.2 Proof rule

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

Proof rule 17 (ProvesKnowledgeOfNR)

$$\begin{array}{l} \{ b \in \text{knows}(x) \} \\ \rightarrow b : \text{provesKnowledgeOfNR}(x) \\ \{ \exists b' \bullet (\text{session}(b'; x) \wedge b \neq b') \} \end{array}$$

4.7 *keepSecret*

The abstract message *keepSecret*(s) is identical to the abstract message *canExtract_s*(s): it ensures that the value of s is never learnt from the message sent.

4.7.1 Semantics

The semantics of $keepSecret(s)$ specify that if the value s is known by an agent after the message is sent, then it must have also been known beforehand; further, if it was known beforehand, it will be known afterwards, because it is assumed that agents do not forget facts.

$$\llbracket keepSecret(s) \rrbracket_{\Pi} = \{ m \mid \forall tr \cap \langle \sigma, a : send\ m, \sigma' \rangle \in traces(\Pi) \bullet \\ knows_{\sigma'}(s) = knows_{\sigma}(s) \}$$

The abstract message $keepSecret(S)$ is an extension of the standard $keepSecret$ that specifies that a set of data items S cannot be retrieved from the message. The semantics are defined as:

$$\llbracket keepSecret(S) \rrbracket_{\Pi} = \bigcap_{s \in S} \llbracket keepSecret(s) \rrbracket_{\Pi}$$

4.7.2 Proof rule

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

Proof rule 18 (KeepSecret)

$$\left\{ \begin{array}{l} knows(x) = s_x \wedge \forall y \in s_x \bullet honest(y) \end{array} \right\} \\ \rightarrow : keepSecret(x) \\ \left\{ \begin{array}{l} knows(x) = s_x \wedge \forall y \in s_x \bullet honest(y) \end{array} \right\}$$

5 Conclusions

5.1 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 illustrated the calculus using a small example. We have outlined a semantic model, and used it to help explain some of the constructs of the language; this model could potentially be used to prove the soundness of the calculus.

5.2 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.

5.3 Future Work

A pressing need is to build the foundations upon which the calculus is constructed. We need to develop the semantic model outlined in Section 3, and use it to verify the postulated proof rules. We expect that our attempts to do this verification will reveal changes that are necessary to the semantics of abstract messages, or to the proof rules.

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].

At present, proofs using the calculus are performed by hand. A longer term goal is to provide tool support for protocol developments. The tool would provide support both in producing correct annotations, and in refining abstract messages to concrete messages.

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.
- [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.