# Modelling and verifying key-exchange protocols using CSP and FDR

A.W. Roscoe
Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD United Kingdom

**Abstract**

*We discuss the issues involved in modelling and verifying key-exchange protocols within the framework of CSP and its model-checking tool FDR. Expressing such protocols within a process algebra forces careful consideration of exception handling, and makes it natural to consider the closely connected issues of commitment and no-loss-of service. We argue that it is often better to specify key exchange mechanisms in the context of an enclosing system rather than in isolation.*

## 1 Introduction

In this paper I show how key exchange protocols can be brought into the framework of CSP and potentially verified using the FDR model-checker. In particular I will look at them in the context of the ongoing programme of work on the use of different forms of CSP abstraction to characterise the views of (legitimate or otherwise) users of a system. The theory behind these abstractions has already been developed in [9, 7], where techniques for showing the absence of information-flow are presented and applied.

It is hard to be sure what the right specification of a key-exchange protocol is in isolation. Most of the existing literature, for example [2, 3, 10], concentrates either on logics of knowledge or on tools and strategies for looking for attacks.

It is my thesis that it is easier to formulate desirable properties of an enclosing system such as a session protocol using a key-exchange. Certainly some issues such as no-denial-of-service can *only* be addressed in this context. Therefore much of this paper is devoted to the embedding of key-exchange within a more general protocol.

There are some features of key-exchange protocols which make this subject well-suited to CSP:

- CSP is appropriate for modelling multi-agent systems which communicate via messages;
- CSP forces close attention to issues such as the behaviour of the insecure message transport system which underlies one of these protocols.
- Some forms of insecurity appear not in a single action or individual state, but over series of actions. Since CSP models a process as the possible behaviours over time it can perform, it gives an excellent framework for understanding such subtleties.
- It is possible to build time into CSP modelling, either continuous time (in abstract models) or discrete (for model checking). These both seemingly relate well to uses of time and time-stamping in security protocols.

What limitations there are apply not to the theory but to the difficulties caused to automated verification by the large or infinite data types that a natural presentation of these protocols contains.

### Status of this work

At the time of writing (January 1995) the work reported in this paper is under active development as part of a programme investigating how new insights into the specification and verification of security properties can be applied. This paper reports on the modelling assumptions and methods used, both for key-exchanges and session protocols using them, and also on the appropriate formulation of security properties and expected verification methods.

A session protocol (correct and robust against arbitrary message loss) has been constructed and verified using FDR. This will be reported in [4]. Over the next few months we expect to be able to verify extensions building in (i) encryption using known keys and then (ii) the use of keys obtained using Needham-Schroeder and variants. By this means we expect to have a system that can be shown – in a suitably specified sense – secure and free from denial of service (necessarily under limiting assumptions to bring the state-space size within the bounds of available tools).

Where reference is made to the "session protocol", it refers to either the verified unencrypted one or to these further developments.

This paper is an abbreviated and updated version of the technical report [8]. The main omission from the earlier paper is the CSP 'code' of a prototype session protocol with key exchange and encryption.

# 2 Base model: formalising informal assumptions

## 2.1 Assumptions about encryption

Encrypted messages are assumed to be created using a universally-known algorithm by reference to keys $k$. We will usually that the same key suffices to encrypt and decrypt a message (i.e., this is a *symmetric* algorithm), and that the algorithm is good enough, and the range of keys so large, that we can discount the possibilities of any of the following

(i) decryption by an agent who does not possess the right key;

(ii) encryption in key $k$ by any agent not possessing $k$ (other than by copying an already-observed encrypted message);

(iii) guessing a key;

(iv) the recovery of a key from an encrypted message (even when the contents of the message is accurately guessed).

(v) anyone reasonably believing they have decoded some data when they have not: in other words encryption, even of "random" data contains sufficient redundancy to eliminate mistaken decryptions.

Given these assumptions, it is appropriate to choose an abstract representation $e(k, x)$ of a message $x$ encrypted under key $k$.

We will adopt, at least for now, the principle that keys are inviolable tokens that can be invented, combined with other messages, encrypted and decrypted like any other data, but which have no other operations on them. In other words we forbid the following

- the use of anything other than a specifically-created key as a key;
- the modification by any operation of any key to produce another key.

This is essentially an assumption about the behaviour of the type *KEY* in our programs. It is an assumption that can be enforced by the use of public-key encryption to validate keys, but the real reason for making the assumption is related to the different ways we will wish to treat keys and other data when we consider verification. Of course there are a number of key exchange protocols that do not fit into this pattern. The point of keeping keys and the rest of data separate is that it provides a mechanism for assuming facts about what can be done with, and guessed with, keys without placing constraints on what other messages a user can generate.

For the time being we will not assume (and thus effectively assume the contrary of) any algebraic properties of encryption. Thus we will assume that $e(k_1, e(k_2, x))$ and $e(k_2, e(k_1, x))$ are distinct and that neither confers information about the other in the absence of the keys. In subsequent work it may be possible to relax this assumption, though doing so would require either explicit or implicit symbolic reasoning in any automated verification.

We will assume that if an agent, in possession of key $k$, is able to decipher a message $e(k, x)$, then it may not only recognise this fact but also identify $k$ (necessarily in $K$).

Similar symbolic abstractions from the mechanics of encryption are used almost universally in the literature of crypto protocols, and are sometimes called *envelopes* [1]. A variety of notations are used.

The extent to which the underlying postulates about information-flow from, and creation of, these objects are stated varies widely. Nevertheless they are of vital importance since they represent the specification interface between the protocol designer and the cryptographer. They also allow a precise evaluation and perhaps justification of protocol analysis techniques.

We must treat *nonces*, from a structural point of view, in a similar way. The only difference in a typical protocol is which process is able to generate them.

We are assuming the following forms for data:

- Text, including names of users/nodes and any special signals such as acknowledgements.
- Keys from a given set $K$.
- Nonces from a given set $N$.
- The pairing of two items $d_1.d_2$.
- The encryption $e(k, d)$ for a key $k$ and data $d$.

At any time a node or other observer will have seen a set $S$ of such items, including any with which it was given knowledge initially.

Two related questions that arise are, given the set $S$, what keys does an observer have knowledge of (which we might define to be $keys(S)$) and what messages can the node generate (which we might define to be $span(S)$, borrowing terminology from algebra)?

In fact, $keys(S) = span(S) \cap K$, and so it will suffice to determine $span(S)$; this is the smallest set containing $S$ and which is closed under the following rules

- It contains all text.
- It contains $d_1$ and $d_2$ iff it contains $d_1.d_2$.
- If it contains $k$ then it contains $e(k, d)$ iff it contains $d$.

Thus $k_2$ is in $span\{k_1, e(k_1, e(k_1, k_3).e(k_3, k_2))\}$ but not in $span\{k_1, e(k_1, e(k_2, k_2))\}$.

Informally, $span(S)$ denotes the set of those messages that an observer can generate from a given state of knowledge: as well as the keys it can fully deduce and the messages it can thereby construct, it also contains all of the undeciphered messages it happens to have seen and which, though it cannot understand them, can use in the construction of others (for example by encoding further).

### 2.2 Assumptions about message transmission

Throughout this work I will assume that the processes using encryption communicate via a "medium" process.[1] What passes over the medium is not secure, so we have to rely on the proper use of codes and secure exchanges of keys to achieve security.

This still leaves open a lot of detail of what sort of mechanisms are provided by the medium, and the details of these are really very important in assessing what sort of "attacks" or security breaches can occur.

Our basic model will be that the medium is a process to which messages are output by the nodes, and which delivers them. I think it breaks into two essentially separate cases which require consideration. This depends on the form of the messages and the intelligence of the medium. Can we (and the medium) see openly who the sender and addressee of a message are, or are these encrypted?

[1] An alternative approach is possible, namely identifying the medium with the enemy agent. While elegant, this has the disadvantages from our point of view that specifying no-denial-of-service becomes highly problematic, and that it largely removes our ability to make different assumptions about the disruptive power of the intruder.

1. Messages are openly labelled and the medium acts as a conventional postal service between nodes.

2. Messages are entirely encrypted, and are therefore broadcast by the medium to all nodes. Each node must therefore attempt to decipher every message and will recognise those intended for it, provided it possesses the appropriate key or keys.

Of course there is nothing, in the first case, to prevent a message carrying encrypted information that confirms (or otherwise) the openly-carried information. But there is a clear division between the modelling of these two cases. The first gives considerably greater scope for users not in the possession of keys to interfere with encrypted traffic. While the second apparently puts encrypted messages directly into the hands of all users, this is actually no hardship since we will almost certainly have to assume, in the first case, that a copy of any message might fall into the hands of anybody.

While there are seemingly a number of advantages as well as disadvantages in the closed addressing case – which are discussed in [8] – I will concentrate in this paper on the more usual open address case, not least because it is more interesting from the point of view of CSP modelling.

## 3 Towards the CSP model

In this section and the next we will give an outline description of the Needham-Schroeder protocol in the context of an encrypted session protocol which it supports. The reason for choosing this one is that it is the most extensively researched, and therefore best understood, protocol of this sort and therefore an excellent means of comparison between analysis methods.

My overall model will be of a number of nodes, each a process for handling security aspects of communication for a single user, plus a communications medium that handles messages between them. The external view to a user will be that sessions are open and closed between nodes, and while they are open messages can pass to and fro between the connected pair. We will assume that a fresh key is sought for each session.

In constructing the node process it is simpler to separate the functions of initiating a session and responding to a request for a session. Therefore I will assume that each node is of one of two sorts: an *initiator* that can seek connections with the other class, the *responders*. Thus two nodes of the same sort never have sessions. Of course, in most applications, we would expect each physical user to be served by at least one of each.

### 3.1 CSP model of medium

A message has sender and receiver fields, plus its contents: *NODE.NODE.CONTENTS*. A pure and secure delivery system would then naturally be modelled with a pair of channel arrays *in* and *out* for the insertion and delivery of messages respectively. Node $a$ uses the actions $in.a.b.m$ and $out.a.b.m$ to insert and receive message $m$ for/from $b$. Thus the message that is inserted as $in.a.b.m$ will be delivered as $out.b.a.m$.

We need channels to allow for the interception and faking of messages by intruders. Channels *fkin* and *leak* of the above type, $a.b.m$ represent (respectively) the false insertion or leaking of the message properly delivered as $out.a.b.m$.

The leaking of a message does not itself prevent the proper delivery. To give agents the ability to interfere with delivery an additional channel *kill* is used. Its natural type is *NODE.NODE*, where the *kill.a.b* removes one message (if any) in passage from $b$ to $a$. Note that the re-routing of a message can be achieved by a combination of leaking, killing and faked insertion.

We can actually change *kill* to be a data-free action (i.e., of trivial type), where *kill* has the effect to removing an arbitrary message in transit. The point is that to prove a system secure one would have to prove it secure against this whatever was nondeterministically removed, including any that the agent might have taken out by design.

The most general model might therefore be to create a delivery system *MEDIUM* whose state is a bag (multi-set) of those messages inserted (one way or another) but not yet delivered. The buffering regime will be, following the usual definition of the most nondeterministic buffer, to force input on *in* when empty, output on *out* when nonempty, and to nondeterministically allow input when nonempty.

```
MEDIUM(B) = if card(B)==0 then
                in?x -> MEDIUM({dual(x)})
             else
                ((in?x ->
                  MEDIUM(union(B,{dual(x)}))
                  |~| STOP)
          []  fkin?n?x -> MEDIUM(union(B,{x}))
          [] (if card(B)==0 then STOP else
             |~| x:B @ out!x ->
                       MEDIUM(diff(B,{x}))
          [] leak?n?x:B -> MEDIUM(B))
          [] kill ->
             (if card(B)==0 then MEDIUM(B)
              else |~| x:B @ MEDIUM(diff(B,{x})
              ))
```

The function $dual(x)$ is assumed to turn an 'input' message into an 'output' message by swapping address and sender fields.

A system with less insecurities could be created by simply removing the appropriate clauses (the most likely one being the *kill* one). Equally we could easily place a limit on buffering by reducing the availability of the *in* and *fkin* clauses.

Clearly it would be possible to specify the order of delivery of messages more precisely (perhaps becoming FIFO or just FIFO between a given pair of nodes), though this would introduce questions such as the possible ability of an intruder to insert a message at an arbitrary point in a queue. Note that the intruder can in any case re-order messages by a combination or leaking, deletion and re-insertion.

Given the nature of the protocols which are used to cope with message loss, with repeated sending of messages, and reply to messages by acknowledgements, it is sometimes both natural and necessary (for untimed correctness) to force the medium to accept reply messages in preference to others. This is what is done in the CSP session protocol. Specifically, upon delivering a message which needs a reply, the medium will reserve a slot for this reply (presumably the one it has just used in delivering to the same place). Of course this has to be used carefully, otherwise deadlocks can appear. In the CSP session protocol, the medium is allowed to determine which messages will need an immediate reply. When encryption is built in (and it is no longer reasonable to have the medium understand message contents) then *either* messages expecting replies will need to be specifically tagged as such, *or* probably better (since it eliminates the mischief that can be caused by an intruder changing this tag), the medium will expect a reply to *every* message, though some of these replies will be null and the medium will throw them away.

The above definition does not allow for this, but it could easily be modified to do so.

### 3.2 Needham-Schroeder Protocol

This protocol [5], for establishing session keys between pairs of processes based on communication with a central server, appears to be one of the best-studied examples. It is the one I will concentrate on in this paper, though most of what is said would translate with modifications to any protocol with the same aims.

The actors in this story are a number of legitimate nodes, $A$, $B$, etc., the key-server $S$, the insecure communications medium described above, and whatever agents happen to be intruding on, and interfering with, this medium. Initially each node $A$ knows its *private* key $k_{as}$ which is know only to it and to $S$: it is used for communication between $A$ and $S$. The role of the protocol is to establish, at the request of one participant ($A$, say), a session key $k_{ab}$ for use with a second node $B$.

The protocol is generally expressed as a series of messages that pass between the principals:

1. $A \rightarrow S$: $A, B, N_a$
2. $S \rightarrow A$: $e(k_{as}, N_a, B.k_{ab}.e(k_{bs}, k_{ab}.A))$
3. $A \rightarrow B$: $e(k_{bs}, k_{ab}.A)$
4. $B \rightarrow A$: $e(k_{ab}, N_b)$
5. $A \rightarrow B$: $e(k_{ab}, N_b - 1)$

Here, $N_a$ and $N_b$ are nonces invented by, respectively, $A$ and $B$.

One can paraphrase this protocol thus:

1. $A$ asks $S$ (openly) for a key to $B$.
2. $S$ returns a package encrypted using $A$'s private key containing
    - the return of $N_a$ to reassure $A$ of freshness;
    - the label of $B$ (which appears strictly unnecessary given $N_a$, which ought to identify $B$);
    - the new key $k_{ab}$;
    - an encrypted package (not comprehensible to $A$, which can now be forwarded to $B$).
3. $A$ then forwards the embedded package, containing the name of $A$ and the new key, to $B$.
4. $B$ acknowledges receipt by using the new key on a nonce.
5. $A$ re-acknowledges (the change in the value of the nonce by subtraction is simply to change the message from the previous one).

This protocol is known to be secure except for certain insecurities arising relative to previous compromises.

The protocol, of course, does not tell us everything about how the individual nodes behave. And indeed the list of communications above fails to describe several potentially important features of the protocol, specifically what happens when something goes wrong. Any implementation must have features to deal both with the non-appearance of expected communications within a given time-limit (essential if nonces are to mean anything with respect to timeliness) and the receipt of incorrect communications.

Let us first consider the behaviour of a node outside the protocol. In this state we must be interested in what is communicated since a security leak is clearly possible, particularly since what is to be communicated may be less governed than during the protocol itself. Obviously we would wish that the communications be sufficiently restricted that (i) all communications are encrypted using an appropriate key and (ii) no key is compromised. It is also important that a node correctly recognises when it is supposed to enter the protocol (such as when a new node is attempting to establish a link with it).

The right way to deal with this is to think of the role of the "node" process in our system as being purely a device for managing secure communication. The conceptual model is of a secure telephone, where each call corresponds to a "session", and the calling process instigates the key-distribution protocol for establishing the session key. We can thus make the following division between the node and the external user.

- The external user of the "telephone" decides whom to call, thereby opening a *session* and what substantive messages to send to the user at the other end of a live session. There is also some mechanism for closing a session – hanging up. The messages between user and the local node are unencrypted.
- The external user has no knowledge of the security protocol or of the keys it uses.
- Thus the devices of encryption, such as keys, nonces and encrypted messages play no part in the interface between the node and its user.
- The node operates the protocol as necessary to acquire keys for communication with the other

nodes to which its user wants to send messages, and encrypts/decrypts its user's communications as necessary.

In other words a clear distinction is made between the provision of communications security and the generation of what messages are sent over the service thus provided. Recall that each node is either a receiver or an initiator, and that each user might well possess at least one of each.

Now let us consider the treatment of exceptions by the node. If the node detects some irregularity in its dealings with the rest of the system, there are two ways it might deal with this. The first is to work around the exception – re-establishing links etc. as necessary to recover in a way which is invisible to the outside. The second is to generate an externally-visible effect, perhaps an error event, either because no work-around can be found or because it is thought appropriate to report a security threat. From a verification standpoint one should realise that it will be necessary to handle these two cases differently: the former corresponds to external intruders being unable to produce effects visible to users – which provides a convenient specification in the latter we are giving them an explicit way of doing just that. We will later discuss these two views as the "paranoid" and "confident" methods of system design.

We might also bear in mind that it will not only be the communications within a key-distribution protocol that are potential targets for interference, but also the messages that constitute the session using the key.

Considerable care needed in designing not only key exchange and subsequent session management, but also the transition between them. For example we would not want the system to deadlock because one end of a session believes it to be live and the other does not.

While it is not *necessary* to include all of this extra detail into a CSP presentation of the protocol, it is in my view *natural* to do so. And by doing so properly we can hope to prove much more, with more intuitive specifications. After all, key exchange is a means to an end and not an end in itself: we want to achieve secure communication. The specifications of these higher-level aims can be much clearer – and more clearly "right" – than those of key-exchange. For the external view would (except for the intruder) be entirely divorced from keys.

While we are assuming the intruder can stop any individual message from being delivered, there are arguments for assuming there is a mechanism for avoiding this (by repetition etc.) for certain messages which occur in a session after the key-exchange is complete. There is a far greater need to have this sort of message delivered reliably if the operation of the security system is to be invisible to the external users, because there must come a point where both nodes are committed to the session and cannot be broken out of it by external influences, and the external users should be able to rely on the transmittal of data within a session.

The usual mechanism for reliably transmitting data over a lossy medium is to use a sub-protocol which sends tagged data packets and acknowledgements. While this sort of device cannot be used properly over an enemy-infested network in general (for an enemy could introduce fake messages and acknowledgements to corrupt it), it is possible to use them between pairs of users who share a reliable key. For we can encrypt the communications, thereby preventing this type of corruption. It will certainly be possible to include an explicit coding of such a protocol into our overall system. It is also possible to *assume* that it is provided at a lower level and to make communications between nodes that have a shared key (an sufficient knowledge of this fact) unstoppable via an *event refinement*: the series of messages and acknowledgements implementing a message become one. In this case the approach would be:

- code the main process description as though the given class of messages are certain to be delivered (adjusting the communications service description accordingly), and

- analyse separately the implementation of the chosen protocol in the context of encryption, using the medium where messages can be intercepted.

Overall correctness in this context would then require only the assumption that an enemy is not able to prevent delivery of infinitely many consecutive messages, which is clearly the weakest possible assumptions under which any progress will be provable.

Use of unstoppable messages is a clear abstraction from what is actually implemented. It simplifies both the coding of the protocol and the state-space size in a verification. It does, however create difficulties in formally justifying the abstraction, demonstrating that language constructs (in particular alternatives) are reasonable, and in demonstrating that the assumptions required of keys are valid.

For these reasons I am presently using unstoppable communications as a prototyping tool for CSP descriptions of this sort of protocol. Further theoretical work will be required before we can be confident of what has been proved in a verification of a system abstracted in this way.

The abstraction I have been discussing here is part of a larger question of how much it is (a) possible and (b) desirable to follow the conventional approach of factoring a protocol involving key-exchange and encryption into multiple "layers": the traditional way of overcoming the formidable technical and conceptual difficulties of handling protocol design. To some extent that is what this assumption about reliable message transmission is doing. On the whole I am not convinced that it is often possible to get much useful factorisation between the message-transport level and the reliable establishment of sessions, chiefly because the actions of an intruder have to be dealt very carefully with and recognised at every layer at which they are still visible. If there is a layer lower than the encryption one generating packets or data fields that are not encrypted, then it will be very difficult to prevent that layer being catastrophically disrupted by an enemy (for example, by the faked insertion of acknowledgements).

# 4 Process descriptions

There is insufficient space in this summary to give full CSP descriptions of the processes. Much fuller ones are contained in [8]. What I will attempt to do here is to give a flavour of what each process is doing in each phase of its actions, and to give the main design considerations.

## 4.1 Server

This is the one process we can describe fully, since it is so simple. Its actions are simply to respond to correctly-formed responses to keys. In the following definition, `InsServer` denotes the set of all inputs of the server process (i.e., communications on `in.s.r` and `in.s.i`).

```
SERVER(issued) =
      (in.s?A?B?N ->
      |~|k:diff(K,issued) @
        (out.s.A.e(pk(A),N.B.k.e(pk(B),k.A)) ->
                    SERVER(union(issued,{k}))
      +| x:InsServer -> SERVER(issued))
```

This simple example does illustrate several important features of how the processes are coded:

1. Because we are assuming that all inputs are responded to (even if by a null), we can be sure that the communication medium will accept the output immediately. It is, of course, our obligation to provide an output.

2. In any other state, this and all other processes must be willing to accept any input whatsoever to deny an intruder the ability to deadlock the system. We are, specifically, adopting a *receptive* style of coding CSP. This explains the last line of the above: the operator `+|` is an operator that generalises Hoare's guarded alternative operator in CSP[2]: if an event is possible for the process on the left, it is taken there, otherwise it is accepted on the right. In this case, any input that matches the pattern of the input `in.s?A?B?N` (`A` and `B` being respectively initiator and responder nodes and `N` being a nonce) will be accepted on the last line and ignored. In the input `in.s.r?A?B?N` we are implicitly typing $A$, $B$ and $N$ respectively to node, node and nonce. Any communication where these types are incorrect will fail this clause and thus fall in the exception one.

3. The description of encryption in the CSP retains the symbolic structure of the earlier discussion. Equality is symbolic equivalence, and we can decide decodability by using simple pattern-matching coding of the form: `in.a.b.e(k,?x)` meaning input any message to over `in.a.b` encrypted with key `k`.

## 4.2 Initiator node

Next, let us consider the behaviour of an initiator node. This is always in one of the following phases:

1. Idle: in this state it can accept requests from its user to establish a session with any responder.

2. Establishing a key: attempting to perform a key-exchange protocol with the server and the intended partner.

3. Establishing a session: making the final negotiations before hopefully opening the session.

4. In a session (with communications flowing both ways between the user and the partner).

5. Rejected: a request for a session has been turned down, and the user has to be informed.

6. Closing a session.

[2] Hoare's operator must be used between processes with disjoint sets of explicit prefixes. My generalisation is to drop the disjointness condition, but retaining the requirement that the initial events of both sides are immediately calculable, giving a left-to-right priority allowing the right hand argument to be an "else" clause.

The complexity of this process is such that it is difficult to be precise about its behaviour in a limited space. There are some points that I would like to emphasise here: for further details see [8].

At any stage during the run the node $N$ may receive communications it is not expecting as part of the given session. These can come from three sources:

- The intruder.
- Other processes communicating legitimately, but where $N$ has been previously confused by the intruder into a state where the given communication is no longer expected.
- Communications from other users which represent tail-ends of sessions now complete. (Where both participants know the it is complete, but perhaps the other node does not know $N$ knows! In the context of a network with arbitrary message loss and no time bound on transmission, this must always remain possible.)

Obviously the protocol must prevent ones of the first sort gaining significant access, but we must expect that $N$ can be deceived for at least part of a key-exchange run, leading to the second possibility above. While a node cannot be confident that a series of communications are genuine, it cannot become committed to them, and must realise that its partner is behaving similarly. The crucial analysis we need to carry out in setting up a session – during which we will certainly have a protocol which relies on retransmission until suitable acknowledgements are received – is to identify the point at which we can be confident that the partner node is listening and knows a suitable private key. For then we can repeat communications until they are received and acknowledged in the certainty that unless infinitely many messages are lost by the network, progress will be made. Until that time the node must back out of the current run if either (i) no response is received to a message within a reasonable time or (ii) an unexplained communication is received that is not clearly from the intruder.

I have implemented the Needham-Schroeder protocol in the obvious orientation: with the initiator process also being the one that initiates the key request. It is only after the receipt of the acknowledgement of key receipt from the reponder (the penultimate ommunication of the protocol $e(k_{ab}, N_b)$) that the initiator (thanks to the postulate that a node can always recognise that a message has been encrypted using a specific key, if it knows that key) knows the responder knows $k_{ab}$. Therefore all communications before this point cannot be regarded as unstoppable in the sense discussed earlier, and the initiator must be able to time out. The final communication of Needham-Schroeder $(e(k_{ab}, N_b - 1))$ can be unstoppable (i.e., repeated until a response) and double as a request for commitment to the session.

The initiator must take into account, however, the fact that the responder does not know that the penultimate communication got through, and therefore that the responder might have timed out or been otherwise moved away from the protocol run. Therefore, even after getting so far, the answer might be "no".

I have implemented the unstoppable communications during the final setup phase and the closing phase using a "stop and go" variant of the alternating bit protocol, only with all packets encrypted and using a unique numbering scheme rather than alternating bits to prevent replay attacks.

Closing the connection is relatively straightforward compared to setting it up. In my coding it is the initiator node which requests closure, but there is nothing essential in this. There are various decisions one can make about what happens to pending communications, etc. We should note that one node or the other will certainly be left in the position of not knowing that the other one is entirely through the session. This is the cause of the final type of unexpected communication discussed above.

### 4.3 Responder process

The responder side of a node must obviously be set up to mirror the activities of the initiator. It is simpler in the way it handles a specific run of the protocol, but more complex in other ways since it has to arbitrate between requests.

Specific points to note are

- It must remember the keys of any sessions that were abandoned after it sent the penultimate Needham-Schroeder message, for it can always get a commit request in such a key that it must refuse even though it (the responder) has moved on.
- It must be able to handle requests for connection even when busy in a session. It can refuse these most conveniently by modifying the penultimate Needham-Schroeder message to a special encrypted refusal message.
- Like the initiator, it has to be able to handle the final message of sessions that it knows to have closed. One of the nodes (and in my implementation it is this one) has not only to absorb these messages but also respond to them.

## 5 Verification issues

The way we model as complex a thing as a key exchange protocol has a significant influence on what type of properties we can seek to verify of it. For example it will certainly be influenced by

- Whether time has been used, and if so how: the accurate incorporation of time should sharpen the description of communicating behaviour (particularly where timeouts are used in protocol design), and thereby perhaps into the liveness specification, and also bring time-dependency into the behaviour of the imagined agent (such as minimal time to break a key) and thereby into the security specification.

- Whether the protocol is being considered in isolation – in which case the most we can realistically hope to prove are properties relating to which keys an agent can hold relative to what states of knowledge of the legitimate participants, and perhaps time, all under explicit or implicit assumptions about how the nodes behave outside the protocol – or is encapsulated in a higher-level protocol which uses it to achieve secure session communication. In this latter case what we can prove will certainly depend, *inter alia* on how the enclosing protocol reacts to errors detected at the lower level. One can imagine either a "paranoid" approach, in which any misbehaviour is reported to the external users and possibly causes a deliberate loss of service because of the perceived existence of an intruder, or a "confident" approach where we believe (hopefully because we have proved it!) that an agent can gain nothing useful from his activities, and therefore it is better for the communication system to overcome errors in a way that is invisible to the external users. It should then be possible to prove that the system suffers no loss of service because of the intruder, as well as that the intruder gains no information.

The second major parameter to verifying that a protocol is secure is the threat level: i.e., our assumptions about the abilities of a potential intruder. The abilities of an intruder to intercept, stop and fake messages was discussed in an earlier section, and I think that any reasonable model of threat must include an intruder who can do these things within the limitations that

- The intruder's ability to invent messages is limited to be within the *span* of the messages he has seen and any keys, etc, we are assuming he possesses.

- The proof of some properties may be subject to an assumption that the intruder cannot interfere with a sufficiently high proportion of messages to prevent progress (though the *only* effect of him exceeding this bound must be lack of progress).

Given this, we may seek to prove either an *absolute* security property: this will assume that the intruder never obtains by any mechanism external to our reasoning (such as breaking a code) any key in use; or a *relative* security property where it is assumed that one or more keys do become available and we try to prove that the damage is suitably contained (for example, that breaking one key does not allow the intruder to obtain other ones). All of the following relates to the specification of absolute properties. Most relative properties will involve the incorporation of timing into the specification and implementation.

### 5.1 Confidentiality *versus* robustness

There are two things we will typically want to prove of a "confident" session protocol involving key-exchange and encryption. We want to prove that no information can leak from the trusted users to an intruder, and also that nothing the intruder does can prevent a proper service to these users. The first of these is a *confidentiality* property, whereas the second is a *robustness* or *non-disruption* property. They subtly different flavours of non-interference. One says that nothing the users can do will affect the agent, whereas the other says that nothing the agent can do will affect the users. And indeed, the methods of abstraction developed for CSP in [7] are the key to both. What we need to do is to form an abstracted view of what the system looks like in some alphabet: in our case either the alphabet of the intruder or the alphabet of all legitimate users. Having done this we can reason about how the actions of the other party affect this view.

In the earlier paper, I argued in detail that, in the case where we want to prove that there is no information flow from alphabet $H$ to alphabet $L$, it is necessary to prove that the abstraction $A(P, L)$ (the abstracted view of $P$ in alphabet $L$) is deterministic, for only then can we know that the view in $L$ is independent of any choices that might be made in $H$ (even though the effects might not be apparent in the semantics of the operators used). This is what is needed for a confidentiality property: we would like the abstracted view of an intruder to be deterministic. This type of security property has the advantage that it is certainly closed under refinement.

The situation is subtly different in the case of robustness, and we are trying to show that specific actions of the intruder do not adversely affect the system. For it seems reasonable to take as a base-point that it has either been proved, or is assumed, that the semantics of the system is satisfactory on the assumption that the intruder does nothing. So that if we can prove that the abstracted view of the legitimate users is a refinement of the view from the same perspective assuming no intrusion, then this should be a sufficient demonstration of non-interference. In other words,

$$A(P \underset{L}{\parallel} STOP, H) \sqsubseteq A(P, H)$$

where $L$ is the alphabet of the intruder and $H$ is the complement of $L$. When the abstraction mechanism is the *lazy abstraction* $\mathcal{L}_H(P) = P \mathbin{|||} RUN_L$ of [7] (as we will assume for the rest of this paper, since it is the most appropriate one in the type of example we are considering), the above inequality is a consequence of, but does not imply, the determinism of $A(P, H)$. In other words the above is weaker than the confidentiality property. For it can hold when $A(P, H)$ is nondeterministic: since the process with no intrusion is deemed to be satisfactory, the process with intrusion is still satisfactory since it is a refinement. The point is that we do not care if there might be ways in which the intruder's behaviour can change the ways in which nondeterminism is resolved towards the legitimate users; all we care is that the *range* of nondeterminism is not increased.

This view of non-interference as robustness would have made equal sense as a definition of fault tolerance, with the malicious events of the intruder being replaced by error actions. The only difference is that we are more likely to want to prove fault tolerance on the assumption that the number of faults is bounded. In this case the correct definition would be that

$$\mathcal{L}_H(P \underset{L}{\parallel} STOP) \sqsubseteq \mathcal{L}_H(P \underset{L}{\parallel} Err)$$

where *Err* is a deterministic process allowing whatever sequences of error actions we wish to consider.

An excellent example is given by the FDR session protocol that has already been developed and verified as a prototype. This is specifically designed to be immune from message loss, and this immunity is demonstrated by proving the refinement

$$\mathcal{L}_{\Sigma \setminus L}((SYSTEM \underset{L}{\parallel} STOP)) \sqsubseteq \mathcal{L}_{\Sigma \setminus L}(SYSTEM)$$

(where $L$ is the set of error events, one being necessary for each loss of a message) even though the process on the left hand side is, quite naturally, nondeterministic because of the contention between initiators for one responder.

In conclusion, the specifications of confidentiality and of robustness are similar, but that of robustness can permit more nondeterminism.

### 5.2 Does robustness imply confidentiality?

The previous section shows what we have to do within our framework to prove the two sorts of property we are looking for. It quickly becomes clear, on seeing how either of them must be formulated in the presence of encryption that we must take account of what an intruder can understand: i.e., his *span*. The messages he can introduce into the system are limited to that set, and his ability to decode messages intercepted depends on which keys are presently in the span.

This requires a re-modelling of the specification. In the case of robustness it is relatively easy to see what we must do: the occurrence of intruder outputs to the system can be limited to those events in the span of the set of events the intruder has received from it (plus any we assume are present). This is, in fact, merely a sophisticated version of the case discussed above where, in a fault tolerance specification, we might limit the occurrence of error events. Specifically, if $L$ is the event-set of the intruder (the channels *fkin, kill, leak*), then the precise specification of robustness becomes

$$\begin{aligned} &RUN(L) \mathbin{|||} (SYSTEM \underset{L}{\parallel} STOP) \\ &\sqsubseteq RUN(L) \mathbin{|||} (SYSTEM \underset{L}{\parallel} Knows(I)) \end{aligned}$$

Where $I$ is the assumed initial state of knowledge of the intruder and

$$\begin{aligned} Knows(X) \quad = \quad & kill \rightarrow Knows(X) \\ & \Box\ leak?x \rightarrow Knows(X \cup \{x\}) \\ & \Box\ fkin?x : span(X) \rightarrow Knows(X) \end{aligned}$$

This restriction will certainly be necessary to prove the refinement, for without it the intruder could easily disrupt the protocol by sending messages labelled with a current key.

This gives a remarkably succinct definition of robustness, which is also, in some sense, clearly "right". We should note that in addition to proving that the intruder cannot disrupt communication *between* legitimate users, establishing this robustness property shows that he cannot establish any pseudo-connection *with* one of them.

Unfortunately, with *confidentiality* properties of the sort discussed in [9], we would be forced to take account of the changing span in the abstraction itself.

There are two ways in which our intruder can gain information: he might obtain a key and thereby decrypt messages or he might gain some knowledge of what is happening because of an analysis of network traffic. We have, in fact, gone to elaborate lengths to avoid the former while ignoring the second possibility entirely. It undoubtedly would be possible for some information to be gained about the users' activities of the system we created by observing traffic. The only possible way of eliminating this type of inference is by introducing camouflage traffic to conceal which messages are genuine.

Given that this has little to do, in fact, with keys and encryption, it seems appropriate to concentrate on the issue of which keys the intruder can obtain relative to those used by the system (and perhaps the times when they are used). There seem to me to be two reasonable ways of proving facts about this: one is by building the network so that the keys legitimately used are visible on special channels, and building a process that extracts which keys are visible from the span of the intruder. The confidentiality property can then be specified like any other safety property. We will thus term this the "safety property" method of specification.

The other method is to observe that if our intruder ever knows a key that is used for communication between legitimate nodes (which is all that the keys in our protocol are used for) then if that key is current it would allow him to break the protocol used for the session. Since this would create visible interference it follows that the interference-freedom property must show that no current key is ever known by the intruder. It would be straightforward to alter the responder nodes so that they reflect illegitimate use of old keys in external behaviour: this would be a less severe coding trick than the safety property method and would extend the robustness-implies-confidentiality argument to old as well as current keys.

In the case the we were examining the key-exchange protocols in isolation rather than in the context of a session protocol, the most satisfactory method of specification appears to be the safety property method.

Model checking programs deal with finite-state systems, whereas there are a number of things both in the modelling of the protocols and in the specifications we have discussed here that, naturally, lead to infinite or huge finite numbers of states. Three of these are: choice of keys and nonces; incrementing counter labels on encrypted packets; and remembering the span of the intruder process. Practical verifications will require partial systems or approximate checks to get around this. In this they will be little different from existing protocol verification tools (for example [3]) which in any finite run only try a subset of an infinite collection of attacks. With a little care we expect to be able to obtain significant results here.


### Acknowledgements

Peter Ryan has been an excellent guide to a field which I had not encountered less than a year ago, and frequent meetings with him have been essential in formulating my ideas. Dave Jackson and Paul Gardiner of Formal Systems have helped both in general discussions and in producing the tool modifications which this work has required. Jim Woodcock and Lars Wulf have been my partners in developing the CSP theory of non-interference upon which this work rests.

This work was funded by a contract from DRA, Malvern.


## References


[1] P Bieber, N Boulahia-Cuppens, T Lehmann and E van Wickeren, *Abstract Machines for Communication Security* Proceedings of Franconia Workshop.

[2] M. Burrows, M. Abadi and R.M. Needham, *A logic of authentication*, ACM transactions on Computer Systems, **8**, 1 (1990), pp18-36.

[3] C. Meadows, *A system for the specification and analysis of key management protocols*, Proc. 1991 IEEE Computer Society Symposium on Security and Privacy, pp 182-195, IEEE Computer Society Press, 1991.

[4] P.H.B. Gardiner and A.W. Roscoe *The development and verification of a fault-tolerant session-management protocol* In preparation.

[5] R.M. Needham and M.D. Schroder, *Using encryption for authentication in large networks of computers*, CACM, **21**,, 12 1978, 993-999.

[6] A.W. Roscoe and H. McCarthy, *Verifying a replicated database: A Case Study in Model-checking CSP* submitted for publication.

[7] A.W. Roscoe, CSP and determinism in security modelling to appear in the proceedings of 1995 IEEE Symposium on Security and Privacy.

[8] A.W. Roscoe, *Prospects for describing, specifying and verifying Key-exchange protocols in CSP and FDR* Formal Systems Technical Report, December 1994.

[9] A.W. Roscoe, J.C.P. Woodcock and L. Wulf, *Non-interference through determinism*, Proc. ESORICS 94, Springer LNCS 875, pp 33-53.

[10] E. Snekkenes, *Exploring the BAN approach to protocol analysis*, Proc. 1991 IEEE Computer Society Symposium on Security and Privacy, pp 171-181, IEEE Computer Society Press, 1991.