Using CSP to detect errors in the TMN protocol

Gavin Lowe* Bill Roscoe†

August 6, 1997

Abstract

In this paper we use FDR, a model checker for CSP, to detect errors in the TMN protocol [TMN90]. We model the protocol and a very general intruder as CSP processes, and use the model checker to test whether the intruder can successfully attack the protocol. We consider three variants on the protocol, and discover a total of ten different attacks leading to breaches of security.

Index terms: Security protocols, key establishment, CSP, FDR, model checking, cryptography, protocol failure.

1 Introduction

In this paper we consider a protocol due to Tatebayashi, Matsuzaki and Newman [TMN90]. The protocol concerns a mobile communications system. In order for two agents to set up a secure session, communicating over an open channel, they must first decide upon a cryptographic session key, which should be kept secret from all eavesdroppers.

The protocol is subject to a number of attacks; indeed, one such attack (due to Simmons) was presented in the original paper, and a correction suggested. In this paper we present a number of other attacks, including several that succeed against the suggested correction.

Our approach is to use the process algebra CSP [Hoa85], and its model checker FDR [Ros94, For97]. We encode the protocol in CSP, and produce a CSP description of the most general intruder that can interact with the protocol. We then use FDR to detect a number of attacks upon the protocol (FDR searches the state space of the system until it either finds an attack or exhausts the state space; this search is automatic in the sense that it does not require user guidance once the system has been modelled in CSP). Some of the attacks allow an intruder to imitate another agent in a fake session; other attacks allow the intruder to learn the key being used in a session between two other agents, and so eavesdrop on that session.

In the next section we describe the TMN protocol. In Section 3 we describe how the protocol can be modelled in CSP, and in Section 4 we use FDR, the model checker for CSP, to discover that an intruder can attack the protocol in a number of ways, leading to breaches of security. In Section 5 we adapt the protocol to prevent these attacks, but we then use FDR to discover different attacks (these attacks are also applicable to the original protocol, modulo a few minor changes). We adapt the protocol again, in Section 6, to produce a protocol that appears to be safe, in so far as it prevents any secrets from being leaked. In Section 7 we analyze the protocol from Section 6 to see whether it correctly achieves authentication.

Some of the attacks we discover are known, but most are new. We believe that the number and variety of attacks we discover demonstrate that our approach is a very successful way of automatically detecting errors in security protocols.

*Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK; `gavin.lowe@mcs.le.ac.uk`

†Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; `bill.roscoe@comlab.ox.ac.uk`

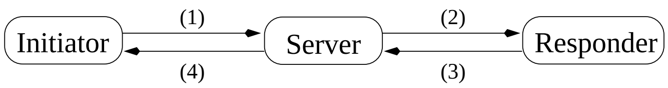
2 The TMN protocol

The TMN protocol concerns three players: an *initiator* A, a *responder* B, and a *server* S who mediates between them. The protocol employs two sorts of encryption:

Standard encryption: This uses an encryption function, which we shall write as E. Every initiator and responder knows how to produce $E(m)$ given message m, but only the server knows how to decrypt such a message to obtain the original message m. This encryption can be implemented using, for example, RSA [RSA78].

Vernam encryption: The Vernam encryption of two keys k_1 and k_2, which we will write as $V(k_1, k_2)$, is their bit-wise exclusive-or. Note that $V(k_1, V(k_1, k_2)) = k_2$, so if an agent knows k_1, then he can decrypt $V(k_1, k_2)$ to obtain k_2. We assume that the keys contain enough redundancy that an agent can know if he has correctly decrypted a message.

The TMN protocol for establishing a session key involves the exchange of four messages; it is illustrated below:



and can be defined as follows:

Message 1.	$A \to S$	:	$A.S.B.E(k_a)$
Message 2.	$S \to B$	:	$S.B.A$
Message 3.	$B \to S$	:	$B.S.A.E(k_b)$
Message 4.	$S \to A$	:	$S.A.B.V(k_a, k_b)$

When the initiator A wants to connect with the responder B, he chooses a key k_a (which the server will later use to return a message to A), encrypts it, and sends it to the server (Message 1). The server sends a message to the responder, telling him that A wants to start a session (Message 2). The responder acknowledges by choosing a session key k_b, encrypting it, and sending it to S (Message 3). The server forms the Vernam encryption of the two keys he has received, and returns this to A (Message 4). When A receives this message, he can decrypt it using the key k_a, to recover the session key k_b.

3 Modelling the protocol in CSP

In this section we give a brief description of how we can model the TMN protocol in CSP. We give a brief overview of CSP in Appendix A for the reader unfamiliar with the language; the syntax of CSP has evolved since [Hoa85]: we highlight some of the differences in footnotes.

We assume the existence of the sets *Initiator* of initiators, *Responder* of responders, and *Key* of keys. We define four different sorts of message, correspond to the four steps of the protocol. Each message includes a tag from the set $\{Msg1, Msg2, Msg3, Msg4\}$; subsequent fields depend upon which protocol step we are dealing with.

$$\begin{aligned}
MSG1 &\mathrel{\widehat{=}} \{Msg1.a.S.b.Encrypt.k \mid \\
&\qquad a \in Initiator, b \in Responder, k \in Key\},\\
MSG2 &\mathrel{\widehat{=}} \{Msg2.S.b.a \mid a \in Initiator, b \in Responder\},\\
MSG3 &\mathrel{\widehat{=}} \{Msg3.b.S.a.Encrypt.k \mid \\
&\qquad a \in Initiator, b \in Responder, k \in Key\},\\
MSG4 &\mathrel{\widehat{=}} \{Msg4.S.a.b.Vernam.k.k' \mid \\
&\qquad a \in Initiator, b \in Responder,\\
&\qquad k \in Key, k' \in Key\},\\
MSG &\mathrel{\widehat{=}} MSG1 \cup MSG2 \cup MSG3 \cup MSG4.
\end{aligned}$$

We are representing a component of the form $E(k)$ by $Encrypt.k$; we are representing a component of the form $V(k, k')$ by $Vernam.k.k'$; the *Encrypt* and *Vernam* components are logically redundant, but remind us of the encryption.

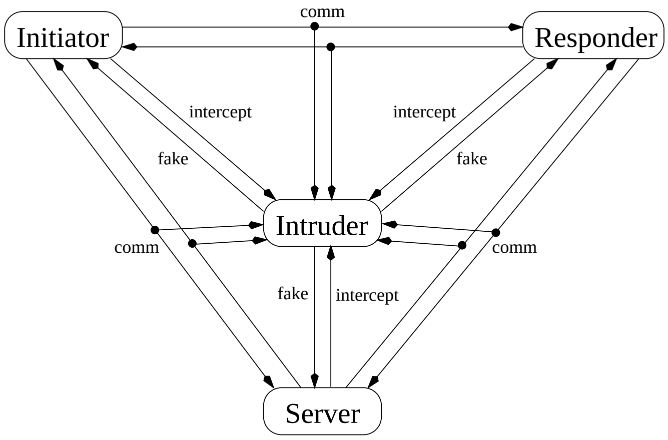
We use three channels to model the communications in the system:

- The channel *comm* will represent standard communications between two honest agents;
- The channel *fake* will represent messages introduced by the intruder: the receiver of these messages should not be aware that they are fakes;
- The channel *intercept* will represent messages sent by an honest agent that are intercepted by the intruder: the sender should not be aware that the message was intercepted.

We declare these channels:

$$\mathsf{channel}\ comm\,,\ fake\,,\ intercept : MSG.$$

These channels are illustrated below:



We also define a number of other channels. The channel *user* will represent requests from a user for sessions to be initiated:

$$\mathsf{channel}\ user : Initiator.Responder.$$

The event $user.A.B$ will represent a request from the user for initiator A to connect with responder B. We will represent a subsequent session using key k by the event $session.A.B.k$; in other words, this event represents the exchange of messages between A and B using key k. We will also model that sessions might be fakes, with an intruder impersonating another agent: the channel *init_fake_session* will represent the initiator taking part in a fake session, where the intruder impersonates the responder; the channel *resp_fake_session* will represent the responder taking part in a fake session, where the intruder impersonates the initiator. Finally, sessions might be overheard by the intruder if they use a key that he knows: we use the channel *leak_session* to represent this:

$$\begin{aligned}
&\mathsf{channel}\ session\,,\ init_fake_session\,,\\
&\qquad resp_fake_session\,,\ leak_session :\\
&\qquad\qquad Initiator.Responder.Key.
\end{aligned}$$

The protocol should prevent these latter events from occurring, but as we shall see, it fails in this respect.

We now produce a CSP process representing each of the agents in the protocol. First, we consider an initiator with identity a. In order to keep the state space finite, we assume that the initiator has access to a finite sequence ks of keys. Ignoring for the moment the possibility of interference from the intruder, the initiator is represented by the process $INITIATOR_1(a, ks)$, below. The initiator first receives a request from the user to set up a session with some responder b. If he has run out of keys, he stops. Otherwise he chooses a key ka, and sends an appropriate Message 1. He then waits for a corresponding Message 4 encrypted with ka; he decrypts this message to obtain the session key kb, and carries out a session using this key.

$$
\begin{aligned}
&INITIATOR_1(a, ks) \mathrel{\widehat{=}} \\
&\quad user.a?b \rightarrow \\
&\quad \text{if } null(ks) \text{ then } Stop \\
&\quad \text{else } INITIATOR_2(a, tail(ks), b, head(ks)), \\
&INITIATOR_2(a, ks, b, ka) \mathrel{\widehat{=}} \\
&\quad comm.Msg1.a.S.b.Encrypt.ka \rightarrow \\
&\quad comm.Msg4.S.a.b.Vernam.ka?kb \rightarrow \\
&\quad session.a.b.kb \rightarrow INITIATOR_1(a, ks).
\end{aligned}
$$

As stated above, we assume that messages contain enough redundancy that an agent can know if he has correctly decrypted a message, so we allow the initiator to accept a Message 4 only if it is encrypted with ka.

We must allow the possibility of intruder action: we must allow instances of Message 1 to be intercepted, instances of Message 4 to be faked, and sessions to be either faked or overheard. We do this via a renaming:[1]

$$
\begin{aligned}
&INITIATOR(a, ks) \mathrel{\widehat{=}} \\
&\quad INITIATOR_1(a, ks) \\
&\qquad [[comm.Msg1 \leftarrow comm.Msg1, \\
&\qquad\ \ comm.Msg1 \leftarrow intercept.Msg1, \\
&\qquad\ \ comm.Msg4 \leftarrow comm.Msg4, \\
&\qquad\ \ comm.Msg4 \leftarrow fake.Msg4, \\
&\qquad\ \ session \leftarrow session, \\
&\qquad\ \ session \leftarrow init\_fake\_session, \\
&\qquad\ \ session \leftarrow leak\_session]].
\end{aligned}
$$

We can define CSP processes representing the responder and server, similarly.

4 Attacks upon the protocol

We will analyze the security of the protocol by putting it in parallel with an intruder. We want to model the intruder as a process that can perform any attack that we would expect a real-world intruder to be able to perform. Thus our model will allow the intruder to send messages using keys that he knows, or replay encrypted messages (even if he does not understand the contents), but we will not allow the intruder to correctly guess any keys. More precisely, we model an intruder who can:

- Overhear messages so as to learn their contents, possibly intercepting these messages;
- Derive new messages from ones he already knows, via encryption, Vernam encryption or Vernam decryption;
- Fake new messages using messages he knows;
- Use keys he knows to take part in fake sessions or overhear sessions between two other agents.

[1] This is the process that will perform either a $comm.Msg1$ or $intercept.Msg1$ event whenever $INITIATOR_1(a, ks)$ performs a corresponding $comm.Msg1$ event, etc.

We consider an intruder with identity C who initially knows a key Kc (distinct from keys invented by other agents). We assume that the intruder is a user of the protocol in his own right, so can use the protocol to establish sessions with other agents, and other agents might try to establish sessions with him. The CSP model of the intruder is given in Appendix B.

We now consider a system with an intruder. Firstly we form the system without the intruder:[2]

$$
\begin{aligned}
AGENTS &\mathrel{\widehat{=}} INITIATOR_1(A, \langle Ka \rangle) \\
&\qquad\quad [|\{|session.A.B, leak\_session.A.B|\}|] \\
&\quad\ RESPONDER_1(B, \langle Kb \rangle), \\
SYSTEM_0 &\mathrel{\widehat{=}} AGENTS\ [|\{|comm|\}|]\ SERVER.
\end{aligned}
$$

We have given the initiator and responder one key each, so each can run the protocol at most once; in later sections, we will give them more keys to allow them to run the protocol more times, although this will lead to a larger state space to be checked.

We then add the intruder:

$$
\begin{aligned}
&SYSTEM \mathrel{\widehat{=}} \\
&\quad SYSTEM_0\ [|INTRUDER\_ALPH|]\ INTRUDER, \\
&INTRUDER\_ALPH \mathrel{\widehat{=}} \\
&\quad \{|comm, fake, intercept, init\_fake\_session, \\
&\quad\ resp\_fake\_session, leak\_session|\}.
\end{aligned}
$$

Note that in the above definition:

- $session$ events are shared between the initiator and responder, so the intruder takes no part in these events;
- $init\_fake\_session$ events are shared between the initiator and intruder, so the responder takes no part in these events;
- $resp\_fake\_session$ events are shared between the responder and intruder, so the initiator takes no part in these events;
- $leak\_session$ events are shared by all three agents, so the intruder overhears the session between the initiator and the responder.

The definition of the intruder is such that an $init\_fake\_session$, $resp\_fake\_session$ or $leak\_session$ event will occur only if the intruder knows the key used, and so is able to decrypt messages in those sessions.

We want to know whether the intruder can ever spy upon sessions or cause fake sessions to be set up between A and B; i.e., we want to know whether the system with an intruder will ever perform $leak\_session.A.B$, $init\_fake\_session.A.B$ or $resp\_fake\_session.A.B$ events.

[2] Our notation differs a little from that of [Hoa85]. We write $P\ [|A|]\ Q$ for the parallel composition of P and Q, synchronizing on the set of events A. We write $\{|c_1, \ldots, c_n|\}$ for the set of all communications over channels $c_1, \ldots, c_n$.

Thus we will test our system against the specifications:[3]

$$SPEC_l \mathrel{\widehat{=}} CHAOS(\Sigma - \{|leak\_session.A.B|\}),$$
$$SPEC_i \mathrel{\widehat{=}} CHAOS(\Sigma - \{|init\_fake\_session.A.B|\}),$$
$$SPEC_r \mathrel{\widehat{=}} CHAOS(\Sigma - \{|resp\_fake\_session.A.B|\}).$$

If the system with the intruder refined these specifications, then it would indeed be secure. However, FDR can be used to discover that $SYSTEM$ does not refine any of the above specifications; it discovers the following attacks upon the protocol.

Attack 4.1. $SYSTEM$ does not refine $SPEC_r$. It can perform the trace:

$$\begin{array}{l}\langle fake.Msg1.A.S.B.Encrypt.Kc,\\ comm.Msg2.S.B.A,\\ comm.Msg3.B.S.A.Encrypt.Kb,\\ intercept.Msg4.S.A.B.Vernam.Kc.Kb,\\ resp\_fake\_session.A.B.Kb\rangle.\end{array}$$

We can rewrite the attack in more conventional style; we write, for example, C_A to represent the intruder C imitating A:

Message 1.	$C_A \to S$	:	$A.S.B.E(Kc)$
Message 2.	$S \to B$	:	$S.B.A$
Message 3.	$B \to S$	:	$B.S.A.E(Kb)$
Message 4.	$S \to C_A$	:	$S.A.B.V(Kc, Kb)$

The intruder sends a Message 1, pretending to be A, sending his own key. No attempt is made to authenticate the identity of the initiator, so the server and receiver are fooled into thinking that A did indeed send this message. The responder therefore agrees to carry out a session, believing he is talking to A. □

Attack 4.2. $SYSTEM$ does not refine $SPEC_i$. It can perform the trace:

$$\begin{array}{l}\langle user.A.B,\\ comm.Msg1.A.S.B.Encrypt.Ka,\\ intercept.Msg2.S.B.A,\\ fake.Msg3.B.S.A.Encrypt.Kc,\\ comm.Msg4.S.A.B.Vernam.Ka.Kc,\\ init\_fake\_session.A.B.Kc\rangle.\end{array}$$

We can rewrite the attack as follows:

Message 1.	$A \to S$	:	$A.S.B.E(Ka)$
Message 2.	$S \to C_B$	:	$S.B.A$
Message 3.	$C_B \to S$	:	$B.S.A.E(Kc)$
Message 4.	$S \to A$	:	$S.A.B.V(Ka, Kc)$

When the initiator tries to connect with responder B, the intruder intercepts the Message 2, and sends a Message 3, pretending to be B. No attempt is made to authenticate the identity of the responder, so the server and initiator are fooled into thinking that B did indeed send this message. The initiator therefore establishes a session with the intruder, believing he is talking to B. □

Attack 4.3. $SYSTEM$ does not refine $SPEC_l$. It can perform the trace:

$$\begin{array}{l}\langle fake.Msg1.A.S.B.Encrypt.Kc,\\ comm.Msg2.S.B.A,\\ comm.Msg3.B.S.A.Encrypt.Kb,\\ intercept.Msg4.S.A.B.Vernam.Kc.Kb,\\ user.A.B,\\ comm.Msg1.A.S.B.Encrypt.Ka,\\ intercept.Msg2.S.B.A,\\ fake.Msg3.B.S.A.Encrypt.Kb,\\ comm.Msg4.S.A.B.Vernam.Ka.Kb,\\ leak\_session.A.B.Kb\rangle.\end{array}$$

The attack uses two runs, which we denote α and β, writing, for example, $\alpha.1$ for Message 1 of run α:

Message $\alpha.1$.	$C_A \to S$	:	$A.S.B.E(Kc)$
Message $\alpha.2$.	$S \to B$	:	$S.B.A$
Message $\alpha.3$.	$B \to S$	:	$B.S.A.E(Kb)$
Message $\alpha.4$.	$S \to C_A$	:	$S.A.B.V(Kc, Kb)$
Message $\beta.1$.	$A \to S$	:	$A.S.B.E(Ka)$
Message $\beta.2$.	$S \to C_B$	:	$S.B.A$
Message $\beta.3$.	$C_B \to S$	:	$B.S.A.E(Kb)$
Message $\beta.4$.	$S \to A$	:	$S.A.B.V(Ka, Kb)$

In run α, the intruder impersonates A to set up a session with B using a key Kb, as in Attack 4.1. Then in run β, when A tries to set up a session with B, the intruder intercepts the Message 2, and impersonates B to send the key Kb in a Message 3, as in Attack 4.2. Hence A and B carry out a session using key Kb, which the intruder can overhear. □

From now on, when describing attacks we omit the trace produced by FDR, and simply describe the attack in standard notation.

The reason these attacks succeeded was that Messages 1 and 3 are not authenticated. In the next section we aim to fix this problem.

5 The second protocol

As suggested above, a weakness in the protocol is that the server has no way of authenticating the origins of Messages 1 and 3. We will therefore change the protocol so that each initiator and responder shares a secret with the server, and that these secrets are included inside Messages 1 and 3. The server can check that the secret received is the expected one, and so can be assured as to the origin of the message.

[3] $CHAOS(A)$ is the most nondeterministic, non-divergent process with alphabet A; it can perform any sequence of events from this alphabet. Σ is the set of all events.

The improved protocol can be written as:

$$\begin{array}{lll} \text{Message 1.} & A \to S & : A.S.B.E(s_a.k_a) \\ \text{Message 2.} & S \to B & : S.B.A \\ \text{Message 3.} & B \to S & : B.S.A.E(s_b.k_b) \\ \text{Message 4.} & S \to A & : S.A.B.V(k_a, k_b) \end{array}$$

where s_a and s_b are the secrets that A and B share with the server, respectively. Note that this protocol is very similar to the correction suggested in the original TMN paper [TMN90].

The CSP model of the system is easily adapted to fit the new protocol. The models of the initiator and responder are altered to include their secrets in Messages 1 and 3. The model of the server is altered to check that the secret received is the expected one. The model of the intruder is altered so as to deal with encrypted (secret, key) pairs.

FDR can be used to discover that this protocol is still not secure; it discovers the following attacks.

Attack 5.1. $SYSTEM$ does not refine $SPEC_r$. FDR finds the following attack:

$$\begin{array}{lrl} \text{Message 1.} & C \to S & : C.S.B.E(Sc.Kc) \\ \text{Message 2.} & S \to C_B & : S.B.C \\ \text{Message 2}'. & C_S \to B & : S.B.A \\ \text{Message 3.} & B \to C_S & : B.S.A.E(Sb.Kb) \\ \text{Message 3}'. & C_B \to S & : B.S.C.E(Sb.Kb) \\ \text{Message 4.} & S \to C & : S.C.B.V(Kc, Kb) \end{array}$$

The intruder initially sends a Message 1, requesting a session with responder B, using his own identity, secret and key. When the server checks this message, he finds the correct secret, and so continues the protocol, trying to send a Message 2 to B. However, the intruder intercepts this, and replaces his own identity with A's, so B believes that A is trying to establish a session with B. B returns a Message 3 with a session key Kb, which the intruder intercepts, replacing A's identity with his own. The server then forwards the session key to the intruder. Hence the intruder has successfully impersonated A to set up a session with B. □

Attack 5.2. $SYSTEM$ does not refine $SPEC_i$. FDR finds the following attack:

$$\begin{array}{lrl} \text{Message 1.} & A \to C_S & : A.S.B.E(Sa.Ka) \\ \text{Message 1}'. & C_A \to S & : A.S.C.E(Sa.Ka) \\ \text{Message 2.} & S \to C & : S.C.A \\ \text{Message 3.} & C \to S & : C.S.A.E(Sc.Kc) \\ \text{Message 4.} & S \to C_A & : S.A.C.V(Ka, Kc) \\ \text{Message 4}'. & C_S \to A & : S.A.B.V(Ka, Kc) \end{array}$$

Initially the user asks A to connect with B, and A sends a Message 1. The intruder intercepts this, and replaces B's identity with his own. The intruder then receives the Message 2 from the server, and sends back a Message 3 using his own identity, secret and key; this message is of the form expected by the server. When the server tries returning the session key to A, the intruder intercepts the message, and replaces his own identity with B's; so A receives a message of the expected form. Hence A is fooled into thinking he has established a session with B, but the intruder knows the key. □

Attack 5.3. $SYSTEM$ does not refine $SPEC_l$. FDR finds the following attack:

$$\begin{array}{lrl} \text{Message } \alpha.1. & A \to S & : A.S.B.E(Sa.Ka) \\ \text{Message } \alpha.2. & S \to B & : S.B.A \\ \text{Message } \alpha.3. & B \to S & : B.S.A.E(Sb.Kb) \\ \text{Message } \alpha.4. & S \to A & : S.A.B.V(Ka, Kb) \\ \text{Message } \beta.1 & C \to S & : C.S.B.E(Sc.Kc) \\ \text{Message } \beta.2. & S \to C_B & : S.B.C \\ \text{Message } \beta.3. & C_B \to S & : B.S.C.E(Sb.Kb) \\ \text{Message } \beta.4. & S \to C & : S.C.B.V(Kc, Kb) \end{array}$$

In run α, A and B establish a session using key Kb. The intruder then sends a Message 1, apparently seeking to establish a session with B. When the server sends the Message 2, the intruder intercepts it, and fakes a Message 3, replaying the component $Encrypt.Sb.Kb$ seen earlier. The server therefore returns the session key Kb to the intruder, Vernam encrypted with the intruder's key. Hence the intruder can learn the session key Kb, and understand messages in the session between A and B. □

If we consider a slightly weaker intruder, who can learn encrypted (secret, key) pairs only from instances of Message 1, but not from instances of Message 3, then FDR finds a slightly different attack.

Attack 5.4. The system described above does not refine $SPEC_l$. FDR finds the following attack:

$$\begin{array}{lrl} \text{Message } \alpha.1. & A \to S & : A.S.B.E(Sa.Ka) \\ \text{Message } \alpha.2. & S \to B & : S.B.A \\ \text{Message } \alpha.3. & B \to S & : B.S.A.E(Sb.Kb) \\ \text{Message } \alpha.4. & S \to A & : S.A.B.V(Ka, Kb) \\ \text{Message } \beta.1 & C_A \to S & : A.S.C.E(Sa.Ka) \\ \text{Message } \beta.2. & S \to C & : S.C.A \\ \text{Message } \beta.3. & C \to S & : C.S.A.E(Sc.Kc) \\ \text{Message } \beta.4. & S \to C_A & : S.A.C.V(Ka, Kc) \end{array}$$

In run α, A and B establish a session using key Kb, as in Attack 5.3. The intruder then fakes a Message 1, apparently from A to the intruder, replaying the component $Encrypt.Sa.Ka$ seen earlier. He receives a Message 2 from the server, and returns his own key and secret in a Message 3. The server then tries returning $Vernam.Ka.Kc$ to A, but the intruder intercepts this, and decrypts it to obtain the key Ka. He can then decrypt the component $Vernam.Ka.Kb$ seen earlier, to obtain the session key Kb, and so can overhear messages in the session between A and B.

This attack is very similar to the attack found by Simmons [TMN90, Sim94]. □

Note that attacks similar to the ones in this section could be applied to the original protocol.

The reason these attacks succeed is that the intruder is able to alter the meaning of legitimate instances of Message 1 and Message 3. In Attacks 5.2 and 5.4, the initiator sent a Message 1 using B's identity, which the intruder could replay, replacing B's identity with his own. In Attacks 5.1 and 5.3, the responder sent a Message 3 using A's identity, which the intruder could replay, replacing A's identity with his own. In the next section we see how to fix these problems.

The protocol considered in this section is very similar to the correction suggested in the original TMN paper (in that paper, the encrypted components in Messages 1 and 3 also included timestamps, but this seems to add little to the security of the protocol; the above attacks will still work provided the intruder can replay messages within the lifetime of the timestamps). Hence we have shown that the supposed correction does not work.

6 The third protocol

In order to overcome the problems discovered above, we change Messages 1 and 3 so that the encrypted part includes the identity of the other agent involved in the protocol run. The improved protocol can be written as:

Message 1.	$A \to S$	: $A.S.E(B.s_a.k_a)$
Message 2.	$S \to B$	: $S.B.A$
Message 3.	$B \to S$	: $B.S.E(A.s_b.k_b)$
Message 4.	$S \to A$	: $S.A.B.V(k_a, k_b)$

As before, it is straightforward to adapt the CSP model of the protocol to this new form. FDR can then be used to verify that the resulting system refines each of $SPEC_r$, $SPEC_i$ and $SPEC_l$; we have checked this where A and B can each run the protocol at most twice. We can deduce that the protocol ensures the secrecy of the session keys, at least for the small system considered. This gives us some confidence that the protocol might be secure on a larger system, but does not absolutely prove this: it might be that if we were to consider a larger system, where each agent can perform several runs of the protocol, maybe taking different roles in each run, that we would find an attack. It is fairly easy to see that it is enough to consider only the two honest agents: if an attack made use of other agents, then there would be a similar attack where the intruder took the part of those other agents (we are assuming here that the server allows the intruder to run the protocol with himself). A preliminary investigation of how many runs each agent should be allowed to perform before we can deduce that the protocol is secure appeared in [Low96]; this question deserves further study.

Note that the security of the protocol depends upon the secrets remaining secret: if the intruder were to learn a secret, he would be able to use it to mount an attack similar to the ones in Section 4.

7 Testing for authentication

We now ask whether the protocol we considered in Section 6 authenticates the two agents to one another; in other words, if an agent completes a run of the protocol, can he be sure that the other agent really is present? The authors of the TMN protocol never claimed that the protocol did achieve authentication, so it would be unfair to criticize the protocol if it does not achieve these goals; however, this gives us the opportunity to show how our techniques can be applied to this question.

We want to test whether the following two properties are satisfied:

- If a responder b completes a run of the protocol, apparently with a, then a has previously been trying to run the protocol with b (of course, b cannot be sure that a completed the protocol run, because he cannot be sure that a received the final message; but b should be assured that a at least sent the first message); further, there should be a one-one relationship between the runs of a and the runs of b.

- If an initiator a completes a run of the protocol, apparently with b, then b has previously been trying to run the protocol with a; there should be a one-one relationship between the runs of a and the runs of b; also, the two agents should agree on the value of the key established.

To test these two properties, we introduce four new events into our model of the honest agents.

- Before sending the first message, the initiator a should perform an event $INITIATOR\_running.a.b$; this represents that he is trying to run the protocol with b.

- Before sending the third message (the last message he sends), the responder b should perform an event $RESPONDER\_running.b.a.kb$; this represents that he is trying to run the protocol with a using key kb.

- After receiving the last message, the initiator a should perform an event $INITIATOR\_commit.a.b.kb$; this represents that he thinks he has completed the protocol with b using key kb, and is willing to commit to a session.

- After sending the third message, the responder b should perform an event $RESPONDER\_commit.b.a.kb$; this represents that he thinks he has completed the protocol with a using key kb, and is willing to commit to a session.

For example, the process representing the initiator is as below (the description is adapted from that on page 3, so as to deal with the protocol of Section 6; sa represents the agent's secret):

$$
\begin{aligned}
&INITIATOR_1(a, ks, sa) \mathrel{\widehat{=}} \\
&\quad user.a?b \rightarrow \\
&\quad \text{if } null(ks) \text{ then } Stop \\
&\quad \text{else } INITIATOR_2(a, tail(ks), sa, b, head(ks)), \\
&INITIATOR_2(a, ks, sa, b, ka) \mathrel{\widehat{=}} \\
&\quad INITIATOR\_running.a.b \rightarrow \\
&\quad comm.Msg1.a.S.Encrypt.b.sa.ka \rightarrow \\
&\quad comm.Msg4.S.a.b.Vernam.ka?kb \rightarrow \\
&\quad INITIATOR\_commit.a.b.kb \rightarrow \\
&\quad session.a.b.kb \rightarrow INITIATOR_1(a, ks, sa).
\end{aligned}
$$

We will again consider a system with a single initiator A and a single responder B. To test whether the initiator is correctly authenticated, we must test whether whenever the responder commits to a session, represented by an event $RESPONDER\_commit.B.A.k$, the initiator has previously been running the protocol, represented by an event $INITIATOR\_running.A.B$. We consider the abstraction of the system to these events:

$$
\begin{aligned}
&SYSTEM' \mathrel{\widehat{=}} \\
&\quad SYSTEM \setminus (\Sigma - ALPHA\_AUTH\_INIT), \\
&ALPHA\_AUTH\_INIT \mathrel{\widehat{=}} \\
&\quad \{| INITIATOR\_running.A.B, \\
&\quad\ \ RESPONDER\_commit.B.A |\},
\end{aligned}
$$

and must test whether this process refines the following specification:

$$
\begin{aligned}
AUTH\_INIT \mathrel{\widehat{=}}\ & INITIATOR\_running.A.B \rightarrow \\
& RESPONDER\_commit.B.A?k \rightarrow \\
& AUTH\_INIT.
\end{aligned}
$$

FDR finds that this refinement fails:

Attack 7.1. $SYSTEM'$ can perform the trace:

$$\langle RESPONDER\_commit.B.A.Kb \rangle.$$

This corresponds to the following trace for $SYSTEM$ (FDR includes a debugger which can be used to find the trace performed by any subsystem of the complete system):

$$
\begin{aligned}
&\langle fake.Msg2.S.B.A, \\
&\ RESPONDER\_running.B.A.Kb, \\
&\ intercept.Msg3.B.S.Encrypt.A.Sb.Kb, \\
&\ RESPONDER\_commit.B.A.Kb \rangle,
\end{aligned}
$$

which can be rewritten as:

Message 2.	$C_S \rightarrow B$	:	$S.B.A$
Message 3.	$B \rightarrow C_S$	:	$B.S.E(A.Sb.Kb)$

The intruder can simply fake a Message 2 and intercept the response: the responder receives no assurance from Message 2 that A is present. □

Similarly, to test whether the responder is correctly authenticated, we must test whether each $INITIATOR\_commit.A.B.k$ event is matched by a corresponding $RESPONDER\_running.B.A.k$ event. We test whether the abstraction of the system to these events:

$$
\begin{aligned}
&SYSTEM'' \mathrel{\widehat{=}} \\
&\quad SYSTEM \setminus (\Sigma - ALPHA\_AUTH\_RESP), \\
&ALPHA\_AUTH\_RESP \mathrel{\widehat{=}} \\
&\quad \{| RESPONDER\_running.B.A, \\
&\quad\ \ INITIATOR\_commit.A.B |\}.
\end{aligned}
$$

refines the following specification:

$$
\begin{aligned}
AUTH\_RESP \mathrel{\widehat{=}}\ & RESPONDER\_running.B.A?k \rightarrow \\
& INITIATOR\_commit.A.B.k \rightarrow \\
& AUTH\_RESP.
\end{aligned}
$$

The refinement succeeds when the initiator has a single key (so can run the protocol once), but if the initiator has two keys, then FDR discovers the following attack:

Attack 7.2. $SYSTEM''$ can perform the trace:

$$
\begin{aligned}
&\langle RESPONDER\_running.B.A.Kb, \\
&\ INITIATOR\_commit.A.B.Kb, \\
&\ INITIATOR\_commit.A.B.Kb \rangle.
\end{aligned}
$$

The error is that A commits to two runs, while B thought he was running the protocol only once. This corresponds to the following attack:

Message α.1.	$A \rightarrow S$	:	$A.S.E(B.Sa.Ka)$
Message α.2.	$S \rightarrow B$	:	$S.B.A$
Message α.3	$B \rightarrow S$	:	$B.S.E(A.Sb.Kb)$
Message α.4.	$S \rightarrow A$	:	$S.A.B.V(Ka, Kb)$
Message β.1.	$A \rightarrow S$	:	$A.S.E(B.Sa.Ka')$
Message β.2.	$S \rightarrow C_B$	:	$S.B.A$
Message β.3.	$C_B \rightarrow S$	:	$B.S.E(A.Sb.Kb)$
Message β.4.	$S \rightarrow A$	:	$S.A.B.V(Ka', Kb)$

In run α, the intruder allows a normal run of the protocol to occur between A and B. Later, when A tries to connect with B again, the intruder replays the Message 3 from the first run, and so A is fooled into thinking that he has established a second session with B, when B might no longer be present. Further, the two runs could be a long time apart, which might mean that the intruder has had time to apply cryptanalysis to the messages from the first run so as to learn the value of the key Kb. □

These attacks could be prevented in a number of ways, such as including extra nonce challenges between the agents. However, we will not consider these changes further, for it is clear that the resulting protocol would be very different from the original TMN protocol.

The authentication specifications we used above were *extensional*, meaning that they were independent of the details of the protocol messages themselves, and instead referred to the states of mind of the agents. In fact, the above specifications represent just one level in a hierarchy of extensional authentication specifications studied in [Low97b]. In [Ros96], *intensional* authentication specifications were studied, that is specifications that assert, in terms of the communications, the way in which a particular state is reached; thus intensional specifications talk about the actual messages passed, and assert that these messages do actually occur in the order described by the abstract protocol description.

8 Conclusions

In this paper we have shown how the FDR refinement checker for CSP can be used to identify errors in a key distribution protocol. We have described nine different attacks on the protocol (with a tenth below), several of them new; we have also discovered several other attacks, not documented here, on slight variants of the protocol. We believe that this is a practical and useful way of analyzing security protocols, allowing for the fast exploration of the state space, and capable of discovering new attacks.

The checks we carried out to find the errors in the protocol are well within FDR's limits: the checks typically took about two minutes and checked about 4000 states—FDR can easily deal with systems with well over a million states.

In [TMN90], an attack on the protocol, due to Simmons, was described. The attack makes use of the fact that if the encryption in steps 1 and 3 is implemented using RSA [RSA78], then the encryption satisfies the following homomorphic property:

$$E(k \times k') = E(k) \times E(k')$$

where "×" represents multiplication modulo the public modulus. In Simmons's attack, two intruders C and D, who have previously agreed a key Kd, conspire together as follows:

Message α.1.	$A \to S$	:	$A.S.B.E(Ka)$
Message α.2.	$S \to B$	:	$S.B.A$
Message α.3.	$B \to S$	:	$B.S.A.E(Kb)$
Message α.4.	$S \to A$	:	$S.A.B.V(Ka, Kb)$
Message β.1.	$C \to S$	:	$C.S.D.E(Ka \times Kc)$
Message β.2.	$S \to D$	:	$S.D.C$
Message β.3.	$D \to S$	:	$D.S.C.E(Kd)$
Message β.4.	$S \to C$	:	$S.C.D.V(Ka \times Kc, Kd)$

In run α, A and B run the protocol to establish a key Kb. Then C uses Message α.1 to calculate $E(Ka) \times E(Kc) = E(Ka \times Kc)$, and sends this in Message β.1. D responds by sending the pre-arranged key Kd in Message β.3, causing the server S to return $V(Ka \times Kc, Kd)$ to C. C can then calculate $Ka \times Kc$, and hence Ka, and can use this to decrypt Message α.4 to obtain the session key Kb, and hence can overhear messages between A and B.

The fact that the above attack is described in terms of two intruders is not important. As we have seen, the protocol fails to authenticate messages, so D's part in the above attack could be taken by C imitating D, or indeed C imitating B.

The use of the homomorphic property is, to a certain extent, a red herring. There is a very similar attack where C simply replays $E(Ka)$ in Message β.1 (the resulting attack is very similar to our Attack 5.4). Simmons seems to be assuming that the server is able to spot replays of previously used keys (although this is never mentioned explicitly in [TMN90]), in which case the multiplication by Kc serves to disguise Ka. Further, even if the server does have this property, several of our attacks still work (Attacks 4.1, 4.2, 5.1, 5.2 and 7.1). And where our attacks do not still work, if we adapt the CSP model of the server to keep track of the keys seen so far and to only accept new ones, then FDR simply finds slightly different attacks; for example, the overhearing attack against the second protocol (from Section 5) becomes:

Attack 8.1.

Message α.1.	$A \to C_S$	:	$A.S.B.E(Sa.Ka)$
Message α.2.	$C_S \to B$	:	$S.B.A$
Message α.3.	$B \to C_S$	:	$B.S.A.E(Sb.Kb)$
Message β.1.	$C \to S$	:	$C.S.B.E(Sc.Kc)$
Message β.2.	$S \to C_B$	:	$S.B.C$
Message β.3.	$C_B \to S$	:	$B.S.C.E(Sb.Kb)$
Message β.4.	$S \to C$	:	$S.C.B.V(Kc, Kb)$
Message γ.1.	$C_A \to S$	:	$A.S.C.E(Sa.Ka)$
Message γ.2.	$S \to C$	:	$S.C.A$
Message γ.3.	$C \to S$	:	$C.S.A.E(Sc.Kc')$
Message γ.4.	$S \to C_A$	:	$S.A.C.V(Ka, Kc')$
Message α.4.	$C_S \to A$	:	$S.A.B.V(Ka, Kb)$

In run α, the intruder imitates the server to help A and B establish a key; he uses the server as an oracle in run β to learn Kb; he uses the server as an oracle in run γ to learn Ka. □

In Appendix C we briefly discuss how we would go about modelling the homomorphic property of RSA encryption.

The TMN protocol has been the subject of two previous case studies using protocol analysis tools. In [KMM94], Kemmerer, Meadows and Millen use the TMN protocol as the subject of a comparative case study, using three different tools. In the Interrogator [MCF87], a small system running the protocol is described symbolically in Prolog, and the Interrogator searches the state space, possibly being guided by the user, to discover if an insecure state is reachable. The NRL Protocol Analyzer [Mea96] is more geared towards verifying protocols than discovering attacks, and also requires human guidance; an arbitrary system running the protocol is modelled, and the Protocol Analyzer is used to prove that portions of the state space are unreachable. Inatest [EK85] is a symbolic execution tool, which can be used to walk through a protocol execution and demonstrate vulnerabilities. The Interrogator and the NRL Protocol Analyzer are both used to discover the same attack on the protocol, namely our Attack 4.2, while Inatest is used to reproduce Simmons's attack.

Mitchell, Mitchell and Stern [MMS97] use Murϕ, which is a general purpose state enumeration tool, to analyze the TMN protocol (among others). They also discover our Attack 4.2 and Simmons's attack; further, they discover an overhearing attack similar to our Attack 5.3.

Our approach has been the most successful at discovering new attacks. Further, it has the advantage over the NRL Protocol Analyzer, the Interrogator and Inatest of not requiring user guidance once the system has been modelled. Also, we do not have to constrain the intruder (in order to limit the state space) as much as the Interrogator or Murϕ approaches.

We have not considered attacks on the encryption methods used, only on the protocol itself: we have assumed that the intruder could not decrypt instances of Message 1 and Message 3—because he does not know the correct decrypting key—and could decrypt Message 4s only if he held one of the keys. However, Coppersmith et al. [CFPR96] show that if low-exponent RSA is used (as is suggested in [TMN90]) then a passive attack is possible against the TMN protocol.

As a result of this case study, and others, we now understand fairly well the process of analyzing security protocols using FDR. As a result, we have been able to produce a prototype tool, `Casper` [Low97a], for automatically producing the CSP code from a more abstract description of the protocol. This greatly facilitates the process of modelling protocols in CSP.

References

[CFPR96] Don Coppersmith, Matthew Frankin, Jacques Patarin, and Michael Reiter. Low-exponent RSA with related messages. In Ueli Maurer, editor, *Proceedings of Eurocrypt '96*, number 1070 in Lecture Notes in Computer Science. Springer Verlag, 1996.

[EK85] Steven T. Eckmann and Richard A. Kemmerer. Inatest: An interactive environment for testing formal specifications. *ACM—Software Engineering Notes*, 10(4), 1985.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2—User Manual*, 1997. Available via URL `ftp://ftp.comlab.ox.ac.uk/pub/Packages/FDR`.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[KMM94] Richard Kemmerer, Catherine Meadows, and Jonathan Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2), 1994.

[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.

[Low97a] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997. World Wide Web home page at URL `http://www.mcs.le.ac.uk/~glowe/Security/Casper/index.html`.

[Low97b] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997.

[MCF87] J. K. Millen, S. C. Clark, and S. B. Freedman. The interrogator: Protocol security analysis. *IEEE Transactions on software Engineering*, 13(2), 1987.

[Mea96] Catherine Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, 1997.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare.* Prentice-Hall, 1994.

[Ros96] A. W. Roscoe. Intensional specification of security protocols. In *9th IEEE Computer Security Foundations Workshop*, pages 28–38, 1996.

[Ros97] A. W. Roscoe. The theory and practice of concurrency. Forthcoming, 1997.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sim94] Gustavus J. Simmons. Cryptanalysis and protocol failures. *Communications of the ACM*, 37(11):56–65, 1994.

[TMN90] Makoto Tatebayashi, Natsume Matsuzaki, and David B. Newman, Jr. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology: Proceedings of Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 324–333. Springer-Verlag, 1990.

A A brief overview of CSP

In this section we give a brief overview of CSP. More details can be obtained from [Hoa85, Ros97].

An event represents an atomic communication; this might either be between two processes or between a process and the environment. Channels carry sets of events; for example, $comm.Msg1.A.S.B.Encrypt.Ka$ is an event of channel $comm$. Σ represents the set of all events. The notation $\{|a, b|\}$ represents the set of all events over channels a and b.

In this paper we use processes defined using the following syntax:

$STOP$	process that can perform no events.
$a \to P$	process that can perform the event a, and then act like P.
$P \,\Box\, Q$	external choice; the process can act like either P or Q; the choice is made by the environment.
$P[[a \leftarrow b]]$	process that acts like P, except the event a is renamed to b; this operator can also be used for multiple renamings, and has a comprehension form.
$P \setminus A$	process that acts like P, except all events from the set A are hidden, i.e. made internal.
$CHAOS(A)$	the most nondeterministic, nondivergent process with alphabet A; the process can perform any sequence of events from A.
$P\ [\vert A\vert]\ Q$	parallel composition of P and Q, synchronizing on events from A.

The traces model of CSP represents a process by the set of traces it can perform, where a trace is a sequence of events. We say that process P is (trace) refined by process Q if the traces of P are a superset of the traces of Q. FDR can be used for testing refinement between two finite state processes.

B Representing an intruder in CSP

In this appendix we give a CSP representation of the intruder. We consider an intruder who interacts with the first protocol; the intruders for other protocols are very similar.

We begin by defining the set of facts that the intruder might learn; this consists of the atomic datatypes, encrypted keys, and Vernam encryptions:

$$\begin{aligned} Fact \;\widehat{=}\; & Initiator \cup Responder \cup Server \cup Key \\ & \cup \{Encrypt.k \mid k \in Key\} \\ & \cup \{Vernam.k.k' \mid k \in Key, k' \in Key\}. \end{aligned}$$

We now define the ways in which the intruder can derive new facts from ones he has already learnt. We write $S \vdash f$ if fact f can be derived from the set of facts S. The relation $\vdash$ is defined by the following rules, where k and k' range over keys:

$$\begin{aligned} \{k\} &\vdash Encrypt.k, \\ \{k, k'\} &\vdash Vernam.k.k', \\ \{Vernam.k.k'\} &\vdash Vernam.k'.k, \\ \{Vernam.k.k', k\} &\vdash k', \\ S \vdash f \wedge S \subseteq S' &\Rightarrow S' \vdash f. \end{aligned}$$

The first two rules represent encryption and Vernam encryption; the third rules models the commutativity property of Vernam encryption; the fourth represents Vernam decryption; the final rule is a structural rule, representing that if the intruder can deduce fact f from some set S, then he can also deduce f from any larger set. We recognize that these rules do not capture all the properties of Vernam encryption, but they are enough to uncover all the attacks in this paper.

We now define the submessages of a message; these are the facts that an intruder will learn by seeing a message (without doing any further deductions); they are also the facts that the intruder needs to know in order to send a fake message (this definition could be simplified by assuming that the intruder always knows all the agents' identities):

$$\begin{aligned} & submsgs(Msg1.a.s.b.Encrypt.k) \;\widehat{=}\; \{a, s, b, Encrypt.k\}, \\ & submsgs(Msg2.s.b.a) \;\widehat{=}\; \{s, b, a\}, \\ & submsgs(Msg3.b.s.a.Encrypt.k) \;\widehat{=}\; \{b, s, a, Encrypt.k\}, \\ & submsgs(Msg4.s.a.b.Vernam.k.k') \;\widehat{=}\; \\ & \qquad \{s, a, b, Vernam.k.k'\}, \end{aligned}$$

where a and b range over agents' identities, s over servers, k and k' over keys.

We now define the intruder. We declare a channel $deduce$, which will be used for deducing new facts, and a channel use_key, which will represent the intruder using a key in a fake session:

$$\begin{aligned} \mathsf{channel}\quad deduce &: Fact.\mathbf{P}(Fact), \\ \mathsf{channel}\quad use\_key &: Key. \end{aligned}$$

The definition of the intruder is parameterized by the set of facts that he knows. The intruder can overhear or intercept a message so as to learn all its submessages; he can fake a message if he knows all the submessages; he can deduce a new fact from ones he already knows; he can use a key that he knows in fake sessions[4]:

$$
\begin{aligned}
&INTRUDER_0(S) \mathrel{\hat=} \\
&\quad \Box_{m \in MSG}\ comm.m \to \\
&\qquad INTRUDER_0(S \cup submsgs(m)) \\
&\quad \Box \\
&\quad \Box_{m \in MSG}\ intercept.m \to \\
&\qquad INTRUDER_0(S \cup submsgs(m)) \\
&\quad \Box \\
&\quad \Box_{m \in MSG,\ submsgs(m) \subseteq S}\ fake.m \to INTRUDER_0(S) \\
&\quad \Box \\
&\quad \Box_{f \in Fact,\ f \notin S,\ S \vdash f}\ deduce.f.S \to \\
&\qquad INTRUDER(S \cup \{f\}) \\
&\quad \Box \\
&\quad \Box_{k \in S \cap Key}\ use\_key.k \to INTRUDER_0(S).
\end{aligned}
$$

We rename the channel *use_key* in the above definition[5], and hide the deductions:

$$
\begin{aligned}
INTRUDER \mathrel{\hat=} \ & INTRUDER_0(IK_0) \\
& [[use\_key \leftarrow init\_fake\_session.a.b, \\
& \quad use\_key \leftarrow resp\_fake\_session.a.b, \\
& \quad use\_key \leftarrow leak\_session.a.b \mid \\
& \qquad a \in Initiator, b \in Responder]] \\
& \setminus \{\!|deduce|\!\},
\end{aligned}
$$

where IK_0 represents the initial knowledge of the intruder:

$$IK_0 \mathrel{\hat=} \{A, B, C, S, Kc\}.$$

The technique described above works well in theory, but in practice it runs into difficulties. FDR operates by building explicit state machines for each of the sequential processes in the system. If there are n facts, then the $INTRUDER$ process above will have 2^n states; this number is too large, even for quite simple systems.

We therefore use a slightly different representation of the intruder, which is CSP-equivalent to the above process. Instead of representing the intruder as a single process, which can learn n facts, we represent him as n processes, each of which can learn a single fact; each process has two states, corresponding to knowing or not knowing the corresponding fact. These processes synchronize on events representing overhearing or intercepting messages, deducing new facts, or faking new messages. Using this technique, FDR can build the state machines representing the intruder in time $O(n)$. This representation is discussed in more detail in [Ros97].

[4]Note that we use external choices where nondeterministic ones might seem more natural: this is because we test for refinement in the traces model, where the two operators are equivalent; this makes checking using FDR somewhat faster.

[5]We use here a renaming comprehension; the process $INTRUDER$ can perform any $init\_fake\_session.a.b$, $resp\_fake\_session.a.b$ or $leak\_session.a.b$ event (for any a and b) whenever $INTRUDER_0(IK_0)$ can perform the corresponding *use_key* event.

C Modelling the homomorphism property

In this appendix we briefly discuss how we go about modelling the homomorphic property of RSA encryption within CSP and FDR. We give a high level overview, without going into the mechanics in detail.

We model multiplication of keys symbolically, writing $Product.(k, k')$ for the product of k and k'. We define the set Key of (atomic) keys as before, and consider the set formed by performing at most one multiplication:

$$Key' \mathrel{\hat=} Key \cup \{Product.(k, k') \mid k \in Key, k' \in Key\}.$$

(We could also allow products of more than two keys, but we need to impose some limit in order to keep the state space finite, and the above is enough for our purposes.) We then define the sets of messages in terms of Key', for example:

$$
\begin{aligned}
MSG1 \mathrel{\hat=} \ & \{Msg1.a.S.b.Encrypt.k \mid \\
& \quad a \in Initiator, b \in Responder, k \in Key'\}.
\end{aligned}
$$

We extend the set of facts that the intruder can learn to include products of keys, products of encrypted keys, and encryptions of products of keys:

$$
\begin{aligned}
Fact \mathrel{\hat=} \ & \ldots \cup Key' \\
& \cup \{Encrypt.Product(k, k') \mid k \in Key, k' \in Key\} \\
& \cup \{Product.(Encrypt.k, Encrypt.k') \mid \\
& \qquad k \in Key, k' \in Key\},
\end{aligned}
$$

and extend the intruder's derivation relation to model that he can perform multiplications and divisions (where the derivations are restricted to members of $Fact$):

$$
\begin{aligned}
\{m, m'\} &\vdash Product.(m, m'), \\
\{m, Product.(m, m')\} &\vdash m'.
\end{aligned}
$$

Finally, we model the commutativity property of multiplication and the homomorphic property of RSA:

$$
\begin{aligned}
& Product.(m, m') = Product.(m', m) \\
& Product.(Encrypt.m, Encrypt.m') = \\
& \qquad Encrypt.Product.(m, m').
\end{aligned}
$$

(We could also include the commutativity property of Vernam encryption here.) In order to model these equivalences we:

- represent the equivalences as a set of pairs of messages;
- for each equivalence class, select a representative element (FDR has a built in function to perform this process);
- for each sequential process, rename every event to the representative element of its equivalence class, before combining the processes in parallel.