Embedding agents within the intruder to detect parallel attacks

P. J. Broadfoot and A. W. Roscoe
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK
{Philippa.Broadfoot, Bill.Roscoe}@comlab.ox.ac.uk

Abstract

We carry forward the work described in our previous papers [5, 20, 18] on the application of data independence to the model checking of security protocols using CSP [19] and FDR [10]. In particular, we showed how techniques based on *data independence* [12, 19] could be used to justify, by means of a finite FDR check, systems where agents can perform an unbounded number of protocol runs. Whilst this allows for a more complete analysis, there was one significant incompleteness in the results we obtained: While each individual identity could perform an unlimited number of protocol runs sequentially, the degree of parallelism remained bounded (and small to avoid state space explosion). In this paper, we report significant progress towards the solution of this problem, by means anticipated in [5], namely by "internalising" protocol roles within the "intruder" process. The internalisation of protocol roles (initially only server-type roles) was introduced in [20] as a state-space reduction technique (for which it is usually spectacularly successful). It was quickly noticed that this had the beneficial side-effect of making the internalised server arbitrarily parallel, at least in cases where it did not generate any new values of data independent type. We now consider the case where internal roles do introduce fresh values and address the issue of capturing their *state of mind* (for the purposes of analysis).

1 Introduction

We carry forward the work described in our previous papers [5, 20, 18] on the application of data independence to the model checking of cryptographic protocols using CSP [19] and FDR [10], often via extensions to Casper [13]. Since FDR can only check a finite instance of a problem, it was originally only possible to check small instances of security protocols (only involving a few agents and runs). This was excellent for finding attacks, but unsatisfactory as a method of proof of correctness. There has been work on getting round this limitation in a variety of related approaches to protocol modelling, for example [14, 16, 23].

In our previous papers we showed how techniques based on *data independence* [12, 19] could be used to justify, by means of a single finite FDR check, systems where agents could undertake an unbounded number of runs of the protocol. Most of this work was devoted to showing how a finite type could give the illusion (in a way guaranteed to preserve any attack) of being infinite by a careful process of on-the-fly mapping of values of this type (which might be nonces or keys) once they have been forgotten by trustworthy processes (i.e., become *stale*). Since the CSP codings of security protocols, having been rather complex prior to this work, became far worse with these mappings implemented, their creation was automated in Casper.

Aside from restrictions necessary to make our results work (see below), and assumptions common across the whole field arising from the symbolic representation of cryptographic primitives, there was one significant incompleteness in the results we obtained. This was that, while each individual identity could perform an unlimited number of protocol runs, it usually had to do them in sequence. (For small protocols it was possible to run two parallel instances of an agent, but even that was of course far from unbounded!)

We now report significant progress towards the solution of this problem, by means anticipated in [5], namely by "internalising" protocol roles within the "intruder" process. The internalisation of protocol roles (initially only server-type roles) was introduced in [20] as a state-space reduction technique (for which it was usually spectacularly successful). It was quickly noticed that this had the beneficial side-effect of making the internalised server role arbitrarily parallel, at least in cases where it did not generate any new values of data independent type. But there were two problems which prevented us from immediately internalising all roles:

- Firstly, an internalised protocol role which creates a value during a run can, if it has arbitrarily many protocol runs "live" at the same time, require an unbounded number of fresh values. Our existing methods of mapping stale values could not handle this situation, so there was no way of achieving the essential goal of keeping types small and finite.

- Secondly, an essential part of our CSP models is knowing what a given agent believes about the progress of its protocol runs. To this end we have typically either treated specific protocol messages they send or receive as evidence for their *state of mind* or included specific signals (to the environment) in the definitions of CSP processes representing trustworthy agents. When internalising roles whose progress and state of mind plays a part in the specifications, these standard techniques no longer apply. This is not an issue that arises for server-type roles, since they do not usually form part of the specifications.

This paper is an extended and adapted version of our CSFW paper [6] which was in turn related to the extended abstract presented at WITS [7]. We present

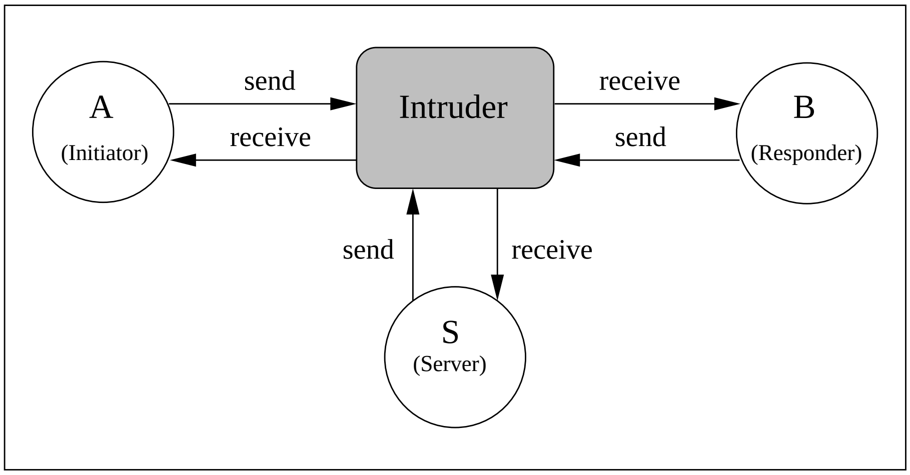

Figure 1: Standard protocol model in CSP.

the techniques we have evolved for internalising agents, as well as the solutions we have devised for the two problems described above. The work this paper reports is based on the later parts on the first author's D.Phil. thesis [3], as well on advances achieved subsequently. In work drawn from [3] we have sometimes abbreviated technical details of examples and proofs, but these can be found in the thesis.

2 Traditional protocol models in CSP

We analyse security protocols using the process algebra CSP [19] and its model checker FDR [10]. In this section, we give a brief and informal overview of the traditional approach, without the application of data independence or the new techniques presented in this paper. Further details can be found in [21].

A security protocol is a sequence of steps taken between two or more parties using cryptography to establish security properties in a potentially hostile environment. In this paper, we will refer to these parties as *protocol roles*. For example, the protocol in Figure 1 comprises the following 3 roles: an initiator A, a responder B and a server S. We refer to instances of protocol roles that are participating in an execution of the protocol as *agents*; each agent is instantiated with an identity. For example, a protocol run could comprise agents *Alice* as role A, *Bob* as role B and *Sam* as role S. An agent can play multiple roles, for example, *Alice* may be running the protocol as roles A and B.

Each honest agent of the protocol is modelled as a CSP process that can *send* and *receive* messages according to the protocol description. These processes are straightforward to derive from standard protocol descriptions in the literature. A *run* of an agent *Alice* is a (not necessarily contiguous) subsequence of messages sent and received by *Alice*. The structure of an initial subsequence of *Alice*'s messages is implied by the protocol and agrees with *Alice*'s a priori knowledge (as far as she is able to determine) on the values of all identities, keys, nonces and so on. A run is complete if it contains every message that *Alice* performs in the protocol.

The intruder is also modelled as a CSP process who can interact with the protocol by overhearing all messages that pass between the honest agents; preventing a message sent by one agent from reaching the intended recipient; generating new messages from those messages held initially or subsequently overheard, subject only to derivation rules that reflect the properties of the crypto-system in use; and sending such messages to any agent, purporting to be from any other. We only allow the intruder to generate messages of size bounded by the message structure of the protocol, so we do not consider messages that do not correspond to part of a genuine protocol message. This is a standard assumption and one which can be justified in many cases, but it should be borne in mind that as with various points of our modelling, all our results are relative to it. The intruder has a deductive system that enables him to deduce information based on his initial knowledge and messages he learns across the network.

As illustrated in Figure 1, the trustworthy and intruder processes are placed in parallel and synchronise upon appropriate events, capturing the assumption that the intruder has control over the network and so can decide what the honest agents receive. We define a specification process that captures the security requirements to be analysed; the model checker FDR is then used to discover whether the specification is satisfied.

In this traditional modelling approach, the number of trustworthy participants running in parallel and the number of runs each one can perform are finite and small (typically 1 or 2 instances of each protocol role).

CSP always models data objects and operations symbolically, so that all messages and their constituents are members of an infinite data type of values. In practice we always limit the size of objects within the data type that are considered by our models to be the maximum size of any message carried within the protocol, so we end up considering a moderately large finite set. For most examples, namely where agents do not do things like sign objects which they do not understand the type of, and where the rules of deduction and algebraic equivalence over data objects are not unusually subtle, this is not a problem and no attacks can be lost. However this limitation should always be remembered, and it applies in the present paper as well as earlier ones.

3 Data independence techniques

In this section, we give a brief summary of the data independence techniques presented in [20] and the early chapters of [3]. Data independence allows us to simulate a system where agents can call upon an unbounded supply of fresh keys even though the actual type remains finite. In turn this enables us to construct models of protocols where agents can perform unbounded sequential runs and verify security properties for them within a finite check. This is achieved in the CSP models by (i) treating the types of the values freshly supplied (such as keys and nonces) as data independent and (ii) implementing a recycling mechanism upon them. We give a brief and informal overview of this approach below.

Special processes, known as *manager processes*, are responsible for supplying the network with fresh values upon request. This method relies upon the assumption that a trustworthy agent (or server) will only store these values for a limited duration; for example, in the standard protocols commonly analysed, an agent will typically remember fresh nonces and session keys solely for the duration of a single protocol run. A fresh value v is referred to as *forgotten* precisely when v is no longer known (stored) by any trustworthy participants. The only component that never forgets these values is the intruder, since he stores all messages ever seen across the network in case they become useful in a later subversion. It is upon these fresh values stored in his memory that the collapsing functions are applied. The recycling of a fresh value v involves all instances of v being mapped to some representative stale (or old) value, known as *background* value, throughout the intruder's memory (there can be any finite number of these values for a particular type, most commonly two). Once the value v has been mapped in this way, it can be re-used as fresh – this mapping process known as the *recycling mechanism*. It is this mechanism that enables us to create the illusion of having an infinite supply of fresh values from a small finite source.

This technique is sound [18, 20], in the sense that any attack that exists upon the infinite system has a counterpart in the transformed system.

As in our previous papers, we restrict our attention to protocols where each run involves a fixed number of participants (in our examples invariably two plus perhaps a server). While agents can rely on equality between two values of a given type (e.g. nonces) for progress, they never rely on inequality (except perhaps with the members of a fixed finite set of constants). A similar condition, termed *positive deductive system* applies to the inferences made by the intruder. For more details see [20]. We have recently been interested to see that this condition is proving necessary for protocol analysis within the rank function and the strand space frameworks [11].

An interesting application of our data independence techniques is the Timed Efficient Loss-tolerant Authentication protocol (TESLA), developed by Perrig *et al.* [17]. This protocol differs from the standard class of authentication protocols previously analysed using model checkers in the following way: a continuous stream of messages is broadcast by a sender; the authentication of the n-th message in the stream is achieved on the receipt of the $n+1$-th message. Thus, the receivers use information in later packets to authenticate the earlier ones. Our data independence techniques were used to capture the unbounded stream of cryptographic keys. Details of this work can be found in [4].

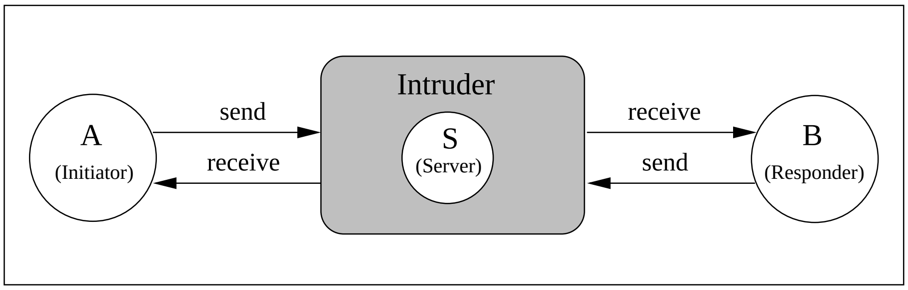

Figure 2: Internal server role in a CSP model.

4 Internalising protocol roles

The natural view of an intruder is of an entity who is trying to break the protocol by manipulating the messages that pass between well-behaved agents. Therefore placing either a server (alternatively known as "trusted third party") or a trustworthy agent within the intruder seems bizarre. However that is not really what we are doing, which is to replace a CSP agent process with a set of inferences of the style used within our coding of the intruder that reflects what the intruder would see if it communicated with the trustworthy agent. The intruder is never given the secrets of a trustworthy process, only a logical picture of what it looks like from the outside when using the other party as an *oracle*. Figure 2 illustrates this model.

Deductions performed by the intruder take the form $X \vdash f$, where X is a finite set of facts and f is a fact that it can construct if it knows the whole of X. The functionality of an internal protocol role A that does not introduce any fresh values is captured by this type of deduction within the intruder: we get a deduction $X \vdash f$ if, after an agent $\mathbf{A}$ taking role A has been told the messages in X in an appropriate order, it can be expected to emit f (where f will, usually, be functionally dependent upon X). In many cases it will also have emitted other outputs f' earlier in the protocol based on subsets X' of the inputs X. The server role in the TMN protocol [24] is such an example, whose function is to receive two messages M_1 and M_3 and construct a corresponding third message M_4, where M_4 only contains variables in M_1 and M_3. Modelled internally, the corresponding deductions would be all valid instantiations of $\{M_1, M_3\} \vdash M_4$.

Internal protocol roles that do introduce fresh values also require a special type of deduction, known as a *generation*. A *generation* has the form $\mathbf{t}, X \vdash Y$, where $\mathbf{t}$ is a non-empty sequence of the fresh objects being created, X is a finite set of input facts, and Y is the resulting set of facts generated containing the fresh values in $\mathbf{t}$. In the CSP implementation, generations are modelled as events over the channel *generate*; the manager processes (responsible for supplying the necessary fresh values) synchronise with the intruder upon these events and determine which values are bound to values in $\mathbf{t}$.

Example 4.1 *Consider the following hypothetical protocol description:*

$$\begin{array}{ll} \textit{Message } 1. & A \to S \ : \ \{B,\, n_a\}_{SKey(A)} \\ \textit{Message } 2. & S \to A \ : \ \{n_a,\, k_{ab}\}_{SKey(A)} \end{array}$$

where A introduces the fresh nonce n_a, S is a server introducing the fresh key k_{ab} and SKey(A) is a symmetric key shared only between A and S. If S is modelled as internal, then its functionality is captured by the following generation:

$$\langle k_{ab} \rangle,\ \{B,\, n_a\}_{SKey(A)} \quad \vdash \quad \{n_a,\, k_{ab}\}_{SKey(A)}$$

Each time such a generation takes place, the key manager synchronises with the intruder and determines which fresh key is bound to k_{ab}.

■

Thus, a role A is *internal* precisely when A's functionality is captured within the intruder component by a series of representative deductions and generations. An instantiation $\mathbf{A}$ of a role A is defined to be *external* otherwise (i.e. when $\mathbf{A}$ is modelled as a CSP process in the standard way and placed in parallel with the rest of the network).

When internalising roles (especially non-server ones) it is often necessary to restrict the patterns of these deductions and generations within the intruder so that they correspond more accurately to the behaviour of real agents. We achieve this (see [3] for details) by means of a special class of constraint processes called *Supervisors*. These are designed to ensure that the internal role's behaviour, after a given generation, follows the protocol sequentially and most particularly does not miraculously "branch" into several continuations of the same run.

There are two main advantages for modelling protocol roles internally within the intruder. The first is that this approach serves as an effective state space reduction technique (as discussed and illustrated in [3]). The second advantage, and one we will be focusing on in this paper, is that the internal model of a protocol role A naturally captures a highly parallelised version within A. If A does not introduce any fresh values (for example, the server role in the TMN protocol), then the intruder is able to capture any degree of parallelism within A by performing the standard deductions on behalf of A. On the other hand, if A introduces fresh values, then the degree of parallelism within A that the intruder can capture is dependent upon the supply of fresh values. In a model where there is an infinite supply, the intruder is able to capture any degree of parallelism within A; however, if this supply is bounded, then the intruder may be restricted to only being able to perform a small number of instances of A in parallel at any one time.

One of the problems that arises from this new modelling approach is that if the intruder is unrestricted, then he can perform any number of these generations he wishes, each time requesting a fresh value; this will result in the corresponding manager running out of fresh values (since there is only a finite source). The intruder can do this, for example, by using the same message 1 to generate many different message 2's, each characterised by a distinct fresh value. Furthermore, he can build up a store of these values and later use them one at a time with the honest agents. For this reason, the recycling mechanism used elsewhere cannot necessarily be applied to these multiple message 2's held within the intruder. As an example, consider the generation of the internal server role in Example 4.1. If the key manager is given a set of n fresh values and the intruder has intercepted the message 1 $\{Bob, N_A\}_{SKey(Alice)}$, then the intruder could legitimately perform the following sequence of generations:

$$\begin{array}{rcl}
\langle K_1 \rangle,\ \{Bob, N_A\}_{SKey(Alice)} & \vdash & \{N_A, K_1\}_{SKey(Alice)} \\
\langle K_2 \rangle,\ \{Bob, N_A\}_{SKey(Alice)} & \vdash & \{N_A, K_2\}_{SKey(Alice)} \\
 & \vdots & \\
\langle K_n \rangle,\ \{Bob, N_A\}_{SKey(Alice)} & \vdash & \{N_A, K_n\}_{SKey(Alice)}
\end{array}$$

thereby always being able to cause the key manager to run out of values, irrespective of the value bound to n. The only way to keep the number of fresh values manageable (or even bounded) is to prevent the intruder storing many fresh values for later use.

Internal protocol roles that generate fresh values raise the following problems: How can we reasonably limit the intruder's appetite for fresh values when it has the capability of requesting any number it wishes on behalf of the internal agents? Furthermore, can we restrict the intruder and still be able to capture attacks for any degree of parallelism within the internal roles? We address these questions in this paper.

5 Just-in-time principle

In this section, we introduce a protocol model property, referred to as *just-in-time* (abbreviated JIT). This property allows us to derive and justify finite bounds upon the intruder that prevent him from requesting an unbounded supply of fresh values, without weakening our analysis (by losing attacks).

We start by introducing a simple definition of equivalence.

Definition 5.1 (External equivalence) *Two traces ω and ω' are defined to be "externally equivalent" precisely when all behaviour involving external agents is identical in both traces, namely:*

$$\omega \upharpoonright \{|send,\ receive|\} \quad = \quad \omega' \upharpoonright \{|send,\ receive|\}$$

where, for a given trace γ and set X, $\gamma \upharpoonright X$ returns the trace of events in γ that are members of X. $\{|send,\ receive|\}$ is the set of all send and receive events.

■

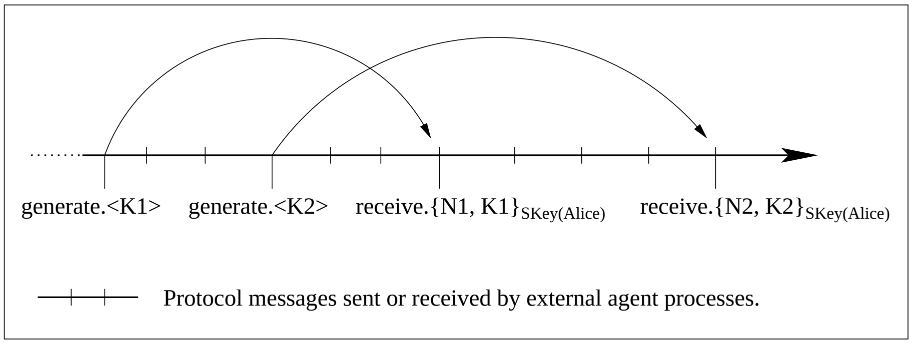

Figure 3: Satisfying JIT

Definition 5.2 (Just-in-time) *Consider a CSP protocol model with a number of externally modelled agents, together with an internal role S, where S introduces fresh values of some type T.*

A fresh value t of type T, received by an external agent, is generated "just-in-time" (JIT) in a trace ω precisely when t is freshly introduced (via the corresponding generation of S) after the occurrence of all the protocol messages that precede the receipt of t (in some message M) by the external agent.

S satisfies the "just-in-time" property with respect to type T precisely when, for every trace ω in the system, either (i) all values of type T are generated just-in-time in ω, or (ii) there exists another trace ω' in the system such that ω' is externally equivalent to ω and all values of type T received by an external agent are generated just-in-time in ω'.

■

Notice that this property is concerned only with those fresh values that *are* eventually passed on to external agent processes and the point at which *they* are generated; the fact that the intruder can store fresh values that he never passes on to external agent processes is an issue we discuss later on. Intuitively, if JIT holds, then there is no advantage to be gained by the intruder to store this type of fresh values, unknown to any external agent processes, that will only be introduced into the network later on.

On the other hand, if a CSP protocol model does not satisfy JIT, then there exists some trace ω that relies on the intruder being able to store fresh values before the point at which they are passed on to an external agent process. By doing this, the intruder is able to construct and send out messages using these values in a way that cannot be reproduced just-in-time. Clearly, this type of behaviour cannot be discarded or ignored, since it might be crucial towards an attack upon the protocol being modelled.

Example 5.1 (Satisfying JIT) *Consider the (hypothetical) protocol description presented in Example 4.1, where the server role S is modelled as internal.*

Suppose there are 2 instances of role A declared as external agent processes, both given the identity Alice. Consider the following valid sequence of events:

1. *Message* 1. $Alice_1 \rightarrow I_S \; : \; \{Bob, N_1\}_{SKey(Alice)}$
2. *Message* 1. $Alice_2 \rightarrow I_S \; : \; \{Bob, N_2\}_{SKey(Alice)}$
3. *(Generation 1)* $\langle K_1 \rangle, \; \{Bob, N_1\}_{SKey(Alice)} \; \vdash \; \{N_1, K_1\}_{SKey(Alice)}$
4. *(Generation 2)* $\langle K_2 \rangle, \; \{Bob, N_2\}_{SKey(Alice)} \; \vdash \; \{N_2, K_2\}_{SKey(Alice)}$
5. *Message* 2. $I_S \rightarrow Alice_1 \; : \; \{N_1, K_1\}_{SKey(Alice)}$
6. *Message* 2. $I_S \rightarrow Alice_2 \; : \; \{N_2, K_2\}_{SKey(Alice)}$

where I_S, Generation 1 and Generation 2 represent the intruder acting on behalf of S.

In this trace, the intruder is not generating the fresh value K_2 just-in-time. However, there exists an equally valid trace of the system that does and is externally equivalent to the trace above, where steps 4 and 5 are swapped; this is illustrated in Figure 3.

■

Example 5.1 illustrates how the intruder has the ability legitimately to generate many messages on behalf of the server S without necessarily passing them on immediately; the number of fresh values he can request is dependent on the number available. In this particular case, there is no advantage to be gained by the intruder from performing generations early and storing the fresh values; it does not enable him to perform any deductions or further generations towards constructing new messages that he otherwise would not be capable of. The protocol only introduces a single fresh key (encrypted under a public key) per run on behalf of S. At any point, the intruder can only ever generate messages of that form and pass at most one fresh value onto an external agent per run. Furthermore, any deductions that he was able to perform earlier, he is always able to perform in the future (since deductions are never disabled).

However, for larger protocol examples, determining whether a given protocol satisfies JIT is much less intuitive and often very complex. Example 5.2 gives an example of a protocol that does not satisfy our property.

Example 5.2 (Violating JIT) *Consider the following hypothetical protocol, where one of the security requirements is that the fresh nonce n_{sec} remains a secret shared only between B and S:*

$$\begin{array}{lll}
\textit{Message } 1. & A \rightarrow B & : \; \{k_1, \; A\}_{PK(B)} \\
\textit{Message } 2. & A \rightarrow S & : \; \{B, \; i_a\}_{SKey(A)} \\
\textit{Message } 3. & S \rightarrow B & : \; \{k_2, \; k_3, \; i_a\}_{SK(S)} \\
\textit{Message } 4. & B \rightarrow S & : \; \{n_{sec}, \; i_a\}_{k_2} \\
\textit{Message } 5. & A \rightarrow B & : \; \{n_{pub}, \; i_a\}_{k_1} \\
\textit{Message } 6. & B \rightarrow A & : \; n_{pub}
\end{array}$$

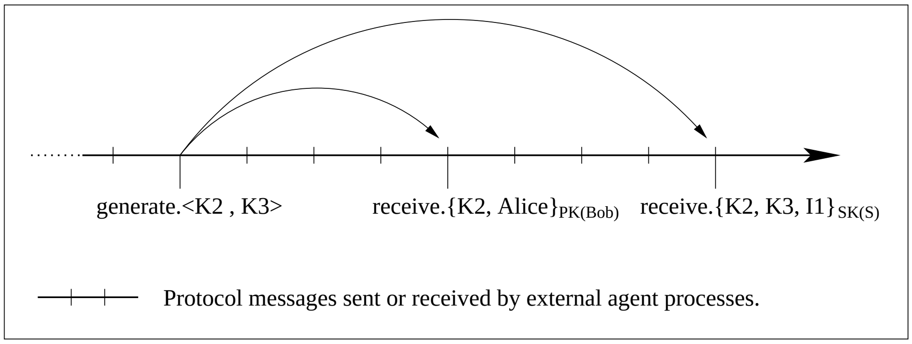

Figure 4: Violating JIT

where k_1, k_2 and k_3 are keys introduced freshly by A and S respectively; i_a is an index value introduced freshly by A; and finally, n_{sec} and n_{pub} are fresh nonces introduced by B and A respectively. $PK(X)$ and $SK(X)$ are the public and private keys for a given agent/server X respectively, and $SKey(A)$ is a symmetric key shared only between A and the server S. Suppose S is modelled as internal and captured by the generation:

$$\langle k_2,\, k_3\rangle,\ \{B,\, i_a\}_{SKey(A)} \ \vdash\ \{k_2,\, k_3,\, i_a\}_{SK(S)}$$

Suppose further that the system is composed of one external instance of each roles A and B with identities Alice and Bob respectively. Consider the following valid sequence of events:

1. *Message* 1. $Alice \rightarrow I_{Bob}\ :\ \{K_1, Alice\}_{PK(Bob)}$
2. *Message* 2. $Alice \rightarrow I_S\ :\ \{Bob, I_1\}_{SKey(Alice)}$
3. *(Generation)* $\langle K_2, K_3\rangle,\ \{Bob, I_1\}_{SKey(Alice)} \vdash \{K_2, K_3, I_1\}_{SK(S)}$
4. *Intruder deduces $\{K_2, Alice\}_{PK(Bob)}$ from step 3.*
5. *Message* 1. $I_{Alice} \rightarrow Bob\ :\ \{K_2, Alice\}_{PK(Bob)}$
6. *Message* 3. $I_S \rightarrow Bob\ :\ \{K_2, K_3, I_1\}_{SK(S)}$

The message 3 *generated on behalf of the server (in step 3) does not satisfy JIT in this trace, since he does not immediately pass K_3 on to an external agent. Instead, he deliberately chooses to gather it first and use the fresh key K_2 to construct a new message* 1. *The intruder then sends this message to Bob in step 5, pretending to be from Alice. It is only at this point that he sends the server-message generated earlier, to Bob in step 6.*

If there exists an externally equivalent trace ω that satisfies JIT, then ω would have to capture the fact that the generation of the server-message is delayed until *immediately before it is sent on to Bob; in other words, after step 3 and before step 6. However, this is not possible since the intruder specifically wants the fresh value bound to k_1 in step 5 (message 1) to be the same as that bound to k_2 in step 6 (message 3); in this trace he uses the K_2 generated on behalf of the server, as illustrated in Figure 4.*

■

Example 5.2 illustrates how there are cases where it is advantageous for the intruder to store fresh values, unknown to any external agents, that he will only pass on to the network later in the trace (thereby violating JIT). The mere fact that storing fresh values gives the intruder an advantage means that it is impossible to find an *externally equivalent* trace where all the fresh values are generated just-in-time.

Such cases arise when the intruder can exploit dependencies between fresh values and the ways in which they are used within the protocol description. In our example above, the intruder exploits the fact that he can bind the same fresh value to the variables k_1 in message 1 and k_2 in message 3, by generating a fresh server-message 3 early on and using the fresh value K_2 in a new message 1. The fact that the values bound to these two variables are the same, allows him to attack the protocol by the sequence of steps taken above, together with the following subsequent behaviour:

7. Message 4. $Bob \rightarrow I_{Server} : \{N_1, I_1\}_{K_2}$
8. Message 5. $I_{Alice} \rightarrow Bob : \{N_1, I_1\}_{K_2}$
9. Message 6. $Bob \rightarrow I_{Alice} : N_1$

By sending the final message 6, *Bob* believes that N_1 is a secret nonce shared only between himself and the server. As we see in step 9, the intruder learns this secret.

In order to perform this attack, the intruder needs to be able to replay message 4 received from *Bob* in step 7 as a message 5 in step 8. The intruder does this by ensuring that the fresh keys bound to k_1 and k_2 are the same. However, the only way he can achieve this, is by generating the server-message 3 *before* sending the corresponding message 1 to *Bob*. In turn, this means that while K_2 is generated just-in-time (since it is immediately passed on to *Bob* as part of the message 1), the fresh value K_3 bound to k_3 is not. There is no way that the intruder can generate K_3 just-in-time and bind the same fresh value to the variables k_1 and k_2.

Determining whether a protocol model satisfies JIT is not straightforward; we discuss how we achieve this later on in this paper. However, once we have established that this property is satisfied by a given protocol model, we are able to derive bounds upon the intruder that prevent him from requesting an unbounded number of fresh values through generations of internal roles and justify that no behaviour of the system is lost.

6 Constructing a reduced protocol model

In this section, we present an extension to our CSP protocol models that involves introducing special values, referred to as *dummy* values, into the data independent types being generated. Together with the JIT property we introduced in the previous section, this extension often enables us to map a protocol model with an infinite supply of fresh values to a reduced system with only a finite set of fresh values which simulates the original one in a similar sense to that achieved in our earlier data independence work with only external agents. The nature of the simulation will be developed so that it means that no attacks are lost through the mapping; if no attack is found upon the reduced system, then none exists upon the original infinite version of the system. This preservation of attacks is, as explained earlier, a general principle of our work.

6.1 Dummy values

The JIT property is concerned only with the fresh values that are generated by the intruder and passed on to external agents; it says nothing about any other fresh values he generates and never sends out. We will refer to this latter class of values as *internal fresh values*. It may initially seem rather odd that the intruder would want to generate fresh values and then never pass them on to any external agent processes. However, without placing any restrictions upon the number of fresh values he can request and having an infinite supply of them, the intruder is free to do and behave how he pleases. There are two main reasons why the intruder may want to store internal fresh values. The first is simply because he is able to do so and therefore stores them with no particular gain. The intruder can do this, for example, by using the same antecedent to generate many different resulting messages, each characterised by distinct fresh values.

The second motivation for holding data containing internal fresh values is that it might enable further generations and deductions for the intruder to take advantage of and construct new messages that he could not have built otherwise. This ability to internally manipulate messages and construct every possible valid message is crucial when working towards developing a complete analysis of protocols, since it considers the full range of the intruder's abilities.

It is of course possible that a single generation $\langle t\rangle, X \vdash Y$ may be used at first as though it was internal – in other words members of Y being used to create other things not involving t – with a message containing t being delivered to an external agent somewhat later. However no use like this is ever essential since the same externally visible effects can in many cases be achieved by performing two separate generations $\langle t\rangle, X \vdash Y$ and $\langle t'\rangle, X \vdash Y'$ (i.e. with the same set of prerequisites X) and delivering two result sets identical except for the name of the fresh value it creates. The first would be performed at the same time as the original generation and its results used in the creation of the objects not using t, and the second as late as possible prior to the delivery of t – now actually t' – to an external agent. Demonstrating that a role is JIT essentially comes down to showing that the second one always can be delayed until the moment before the fresh value is delivered.

Identifying these two *classes* of fresh values in any given trace is the key to how we extend our CSP models and justify finite bounds upon the intruder. The observation we make is that it does not actually matter *which* internal fresh values are supplied; what is important is that they exist in some form for the purposes of allowing the intruder to perform whatever manipulations he needs in order to construct the necessary messages. It is not even necessary for these values to be *fresh*, since they are never passed on to external agents. The intruder could just as easily perform subsequent deductions and generations with any values that were strictly for internal use only.

Based on this observation, we introduce a new class of values, referred to as *dummy* values, that will be added to the data independent types being generated. These extra values have the special characteristic that they are not accepted as genuine by any externally modelled honest process (so the latter will never accept any message involving one). The intruder can use these values itself like any others, in particular doing deductions involving them. The trick is that we allow the intruder to perform, at any time, a "generation" based on a valid input set X, but unless the number of fresh values he is currently storing (unknown to any external agents) is less than the given bound, the result will always be based on a dummy value; otherwise, the result may be either a fresh or dummy value. Hence, for a given bound N upon the intruder, we allow the intruder to perform a generation $\langle t \rangle, X \vdash Y$ for a given input set X and a *fresh* value t precisely when the intruder stores fewer than N fresh values unknown to any external agent processes. In Section 8, we discuss what these bounds should be for various classes of protocols.

In practice, we typically declare one dummy value per generated data independent type. However, there is the possibility that this technique introduces false attacks. An example would be where the value being introduced is a key K, and one of the messages contains something encrypted under K that the intruder would not otherwise learn; representing K by the dummy value, which the intruder could learn from elsewhere, would allow him to deduce the contents of the message, as a false attack. A solution (also applied to the background values [20] to avoid false attacks), is to use two dummy values: one that is created in circumstances where we would expect the intruder to learn it legitimately, and one that is created in other cases. However, a single value appears to suffice more frequently than in the analogous case of background values.

In the rest of this paper, we will refer to this implementation of dummy values within our CSP models as the *dummy-value strategy*.

6.2 Constructing a reduced model

Given a protocol model that satisfies JIT and has an infinite supply of fresh values of the data independent types generated, there exists a reduced model, simulating the original, where there is only a finite source of fresh values, together with the dummy values for the relevant types. By simulation, we mean that for every trace in the original infinite model, there exists an externally equivalent trace in our reduced model. Such a trace is constructed by mapping all the internal fresh values to the dummy values, leaving only those fresh values in the trace that are passed on to external agent processes (and therefore generated just-in-time). Proposition 6.1 captures this more formally.

Proposition 6.1 *Suppose System(AS) is a protocol model with the set of roles AS, where a role A in AS is modelled as internal and introduces fresh values of some data independent type T. Suppose further that System(AS) is provided with an unbounded supply of fresh values of type T. If System(AS) satisfies JIT, then there exists a reduced (finite) system $System_R(AS)$ such that, for every trace ω in System(AS), there exists a trace ω' in $System_R(AS)$ where ω and ω' are externally equivalent. $System_R(AS)$ is constructed as follows:*

1. *The dummy-value strategy is implemented for type T.*
2. *The maximum number of fresh values of type T the intruder can store (unknown to any external agents) is equal to the maximum number of them he can pass on to an external agent in a protocol message.*

Proof We need to show that every trace ω in *System*(AS) can be mapped to an externally equivalent trace ω' in $System_R(AS)$, where $System_R(AS)$ is subjected to the conditions presented above. This is quite straightforward by the definition of JIT, as follows.

By definition, the fact that *System*(AS) satisfies JIT (with regards to the internal role A and type T) means that, for every trace ω in the system, there exists a trace ω' in the system such that ω' is externally equivalent to ω and all values of type T are generated just-in-time in ω'. We can, therefore, consider solely these just-in-time traces and still capture all possible behaviour (by the definition of external equivalence). This is the first step of the reduction. The supply and storage of fresh values though are still infinite. We now need to prove that every trace in this subset of traces of *System*(AS) can be mapped to an externally equivalent trace in $System_R(AS)$.

Suppose ω is a trace of the reduced (but still infinite) *System*(AS), where all the values of type T are generated just-in-time (by definition of the reduction step above). There exists an externally equivalent trace ω' in $System_R(AS)$, where all values of type T that have not been generated JIT are mapped to the dummy value D_T. Since the remaining fresh values are generated JIT, the maximum number of them the intruder will ever need to store (unknown to any external agents) is the maximum number of them he can pass on to an external agent for the first time in a single protocol message. This justifies the bound placed upon the intruder in condition 2 of $System_R(AS)$. Furthermore, the fact that there are only ever a finite number of external agents declared in our CSP models, means that $System_R(AS)$ will only need a finite source of fresh values of type T (by the mechanics of the recycling mechanism in our models, together with the assumption that agent processes never remember fresh values for more than a single run).

■

Example 6.1 *Consider a simple protocol defined as follows:*

$$\begin{array}{lll} \textit{Message } 1. & A \rightarrow S & : \{n_a\}_{SKey(A)} \\ \textit{Message } 2. & S \rightarrow A & : \{k_1,\ n_a\}_{SKey(A)} \\ \textit{Message } 3. & S \rightarrow A & : \{k_2,\ n_a\}_{SKey(A)} \end{array}$$

where n_a is a fresh nonce introduced by A, k_1 and k_2 are keys supplied freshly by the server S and $SKey(A)$ is a symmetric key known only by S and A. Consider the following trace, where the server is modelled as internal, there is one instance of role A declared externally with identity Alice and no dummy values are implemented:

- *Message* 1. $Alice \rightarrow I_S \ : \ \{N_A\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_1 \rangle,\ \{N_A\}_{SKey(Alice)} \ \vdash \ \{K_1, N_A\}_{SKey(Alice)}$
- *(Generation 2)* $\langle K_2 \rangle,\ \left\{ \begin{array}{l} \{N_A\}_{SKey(Alice)}, \\ \{K_1, N_A\}_{SKey(Alice)} \end{array} \right\} \ \vdash \ \{K_2, N_A\}_{SKey(Alice)}$
- *Message* 2. $I_S \rightarrow Alice \ : \ \{K_2,\ N_A\}_{SKey(Alice)}$
- *Message* 3. $I_S \rightarrow Alice \ : \ \{K_1,\ N_A\}_{SKey(Alice)}$

By storing the two server-generated messages 2 and 3, the intruder is able to replay them in reverse order. This trace does not conform to JIT, since K_1 is not generated JIT. However, there exists an externally equivalent trace in this model that does, for example:

- *Message* 1. $Alice \ \rightarrow \ I_S \ : \ \{N_A\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_3 \rangle,\ \{N_A\}_{SKey(Alice)} \ \vdash \ \{K_3, N_A\}_{SKey(Alice)}$
- *(Generation 2)* $\langle K_2 \rangle,\ \left\{ \begin{array}{l} \{N_A\}_{SKey(Alice)}, \\ \{K_3, N_A\}_{SKey(Alice)} \end{array} \right\} \ \vdash \ \{K_2, N_A\}_{SKey(Alice)}$
- *Message* 2. $I_S \rightarrow Alice \ : \ \{K_2,\ N_A\}_{SKey(Alice)}$
- *(Generation 3)* $\langle K_1 \rangle,\ \{N_A\}_{SKey(Alice)} \ \vdash \ \{K_1, N_A\}_{SKey(Alice)}$
- *Message* 3. $I_S \rightarrow Alice \ : \ \{K_1,\ N_A\}_{SKey(Alice)}$

where the fresh key K_3 is not subsequently used, but exists for the sole purpose of allowing the intruder to gain access to a message 3 from the server to replay as a message 2 to Alice. The generation of a message 2 can be performed again with the same antecedents and another fresh key (K_1) just-in-time for supplying it as a message 3 to Alice. Thus, if we implement the notion of dummy values, we can map this trace to the following corresponding one in the reduced model:

- *Message* 1. $Alice \rightarrow I_S \ : \ \{N_A\}_{SKey(Alice)}$

- *(Generation 1)* $\langle K_D \rangle,\ \{N_A\}_{SKey(Alice)} \ \vdash\ \{K_D, N_A\}_{SKey(Alice)}$

- *(Generation 2)* $\langle K_2 \rangle,\ \left\{ \begin{array}{l} \{N_A\}_{SKey(Alice)}, \\ \{K_D, N_A\}_{SKey(Alice)} \end{array} \right\} \ \vdash\ \{K_2, N_A\}_{SKey(Alice)}$

- *Message* 2. $I_S \ \rightarrow\ Alice \ :\ \{K_2,\ N_A\}_{SKey(Alice)}$

- *(Generation 3)* $\langle K_1 \rangle,\ \{N_A\}_{SKey(Alice)} \ \vdash\ \{K_1, N_A\}_{SKey(Alice)}$

- *Message* 3. $I_S \rightarrow Alice \ :\ \{K_1,\ N_A\}_{SKey(Alice)}$

■

Example 6.1 provides a simple example of how a trace in the original protocol model (with potentially infinite supply of fresh values) can be mapped to an externally equivalent one in the reduced model. In this particular example, the intruder needed to perform a generation with a dummy value in order to gain access to the second server-message, before passing on the first. Since intruder deductions and generations are never disabled, he can simply use the same antecedents (in this case, $\{N_A\}_{SKey(Alice)}$) to generate another server-message 3 just-in-time. Which actual fresh value gets supplied does not matter (since the type is data independent), as long as it is fresh. Thus, this has the same effect as gathering the two messages and playing them in reverse order.

As discussed in the introduction, a protocol model with an infinite supply of fresh values enables the intruder to perform attacks for any degree of parallelism among the internal protocol roles. By being able to map an infinite model with an internal role A to an equivalent reduced one (for protocols that satisfy JIT), means that we are able to capture attacks upon protocols for any degree of parallelism within A by performing a finite refinement check.

The main question that we still need to consider is how we determine whether a given protocol model satisfies JIT and so falls within the scope of Proposition 6.1.

7 Factorisability of internal protocol roles

We now introduce a new property, namely the *factorisability* of internal protocol roles, and show that when satisfied by an internal role A within a protocol model it makes JIT, and hence the justification of bounds on the intruder, easy to check.

Establishing JIT turns out to be straightforward for roles that generate at most one value of type T per run. The following definition captures the essence of why this is true and allows us to deal with certain protocols generating more than one. Factorisability says that each run of the role can be *factored* into ones where each generates only one non-dummy value. This definition has proved central to our work on justifying finite bounds on intruders' memory.

Definition 7.1 (Factorisability) *An internal role A is "factorisable" with respect to some data independent type T precisely when, for each run R of an agent* $\mathbf{A}$ *taking role A that generates fresh values $v_1, \ldots, v_k$ of type T, the following conditions are satisfied:*

1. *There exist runs $R_1, .., R_k$ of* $\mathbf{A}$*, where each run R_i contains the fresh value v_i and the dummy value (and possibly values of type T generated by agents other than* $\mathbf{A}$*) only.*

2. *For each output message M in R, there exists at least one R_i that contains M, where $i \in \{1, \ldots, k\}$.*

3. *For all v_i and v_j of type T generated on behalf of* $\mathbf{A}$*, where $v_i \neq v_j$: if* $\mathbf{A}$ *receives v_i back in some protocol message, then neither this message, nor any subsequent message sent or received by* $\mathbf{A}$*, contains v_j.*

■

We emphasise that this definition only constrains the use of those fresh values that are generated by agent **A**. A message can contain as many other values (presumably introduced by other agents) as it likes. Values above are identified by their symbolic representations in the protocol definitions, ignoring any further actual equalities there may be.

An intruder with an internalised factorisable role A is equivalent to one in which agents taking role A are constrained to deliver at most one fresh value (plus a dummy value) per run.

Thus, the ability for the intruder to store messages that contain fresh values (generated on behalf of some internal agent **A** taking role A) in an infinite model for the purposes of replaying them in a different order, is simulated here by the intruder being able to perform an independent run with **A** for each message (and fresh value) required. By the definition of factorisability, he can achieve this by replaying external agent messages as input to the various internal agent runs he is interested in, since there exists a run for each internally generated message and fresh value (where all the other fresh values generated are dummy values).

The fact that the other values generated are dummy values means that the generation of any message containing a fresh value on behalf of **A** is not dependent on **A** being able to distinguish the runs of the protocol; otherwise one could not replay the same messages as input to **A**. Hence the need for the 3^{rd} condition in the definition.

Example 7.1 (Factorisable internal role) *Consider the following protocol:*

$$\begin{array}{lll} Message\ 1. & A \rightarrow S & : \{n_a,\, n_b\}_{SKey(A)} \\ Message\ 2. & S \rightarrow A & : \{k_1,\, n_a\}_{SKey(A)} \\ Message\ 3. & S \rightarrow A & : \{k_2,\, n_b\}_{SKey(A)} \end{array}$$

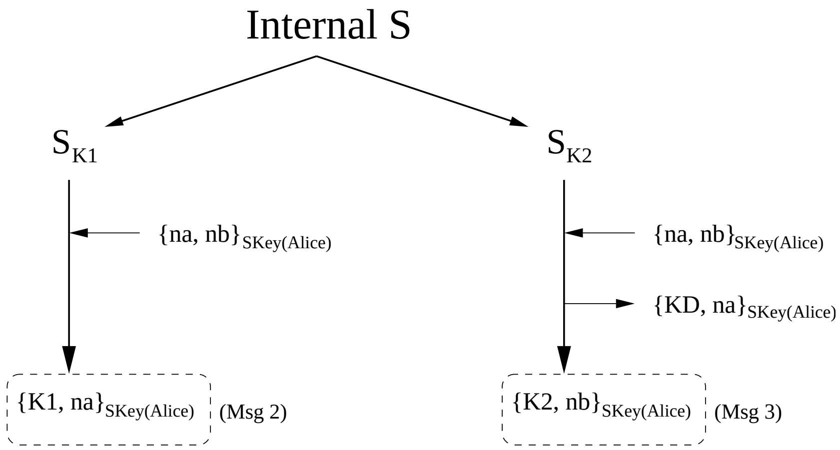

Figure 5: Factorisable internal server role S in Example 7.1.

where n_a and n_b are nonces freshly introduced by A, and k_1 and k_2 are keys freshly introduced by S.

S modelled as internal is factorisable, since it satisfies the conditions required in Definition 7.1. As illustrated in Figure 5, it is straightforward to see that we can factor the runs of S such that each run only ever produces a single fresh value (the rest being dummy values). Since there are only the two messages generated on behalf of S where a single fresh value is introduced in each, every possible output message of S can clearly be generated in one of the independent runs of S. Furthermore, S does not depend on receiving any fresh values previously introduced, thereby satisfying the 3^{rd} condition.

■

Example 7.1 provides a simple example of a factorisable internal server role S and as a consequence, how the intruder can gain access to the fresh values bound to k_1 and k_2 by performing independent runs with S. He achieves this (i) using the dummy values (as illustrated in Figure 5) and (ii) replaying message 1's that he receives from the externally modelled instance (for example, with identity *Alice*) of role A, to S for each run. Without the use of dummy values, the intruder could achieve the same result by performing the same independent runs, where the dummy values are replaced by freshly supplied ones.

Example 7.2 (Non-factorisable internal role) *Consider the following sequence of messages within a protocol:*

$$\begin{array}{lll} \textit{Message } 1. & S \rightarrow \ldots & : \{k_1\}_E \\ \textit{Message } 2. & S \rightarrow \ldots & : \{k_2\}_E \\ \vdots & & \\ \textit{Message } n. & S \rightarrow \ldots & : \{k_1,\ k_2\}_E \end{array}$$

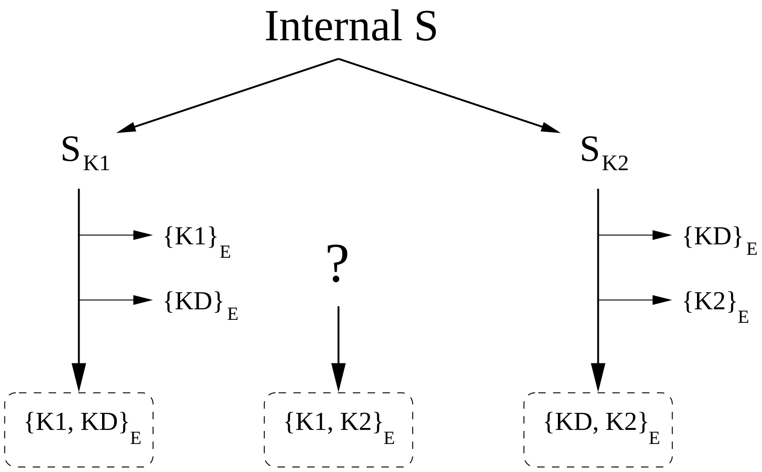

Figure 6: *Non-factorisable* internal server S in Example 7.2.

where S is modelled as internal introducing the fresh keys k_1 and k_2. E is some encryption key known to S and whomever the messages are intended for; the type of encryption is not relevant for the purposes of this example.

If S is factorisable, then there exists independent runs for each fresh value introduced by S. Furthermore, every output message from S must be present in one of these runs. In this example, two factored runs of S are needed, namely one where k_1 was bound to a fresh value and k_2 was bound to the dummy value K_D, and the other vice versa. If the fresh values K_1 and K_2 are bound to k_1 and k_2 respectively in separate runs (while the other is bound to the dummy value K_D), then S will either generate a message n of the form $\{K_1, K_D\}_E$ or $\{K_D, K_2\}_E$; the message $\{K_1, K_2\}_E$ is not generated in either of the factored runs. Figure 6 illustrates this problem.

■

Example 7.2 highlights the problem that can arise as a result of having multiple fresh values in a single message generated by an internal role, even though these values were both introduced in previous messages. In order to satisfy the factorisability property, there cannot be any *dependencies* among fresh values introduced by an internal role S of this nature, since they could result in S being able to distinguish between his runs.

Dependencies between fresh values such as these are not the only source of problems when it comes to the factorisability of internal agents. A more subtle way an internal role S can distinguish between runs is if we allow S to generate a fresh value that depends on S receiving a previously introduced fresh value. The effect this can have is illustrated in Example 7.3.

Example 7.3 (Non-factorisable internal role) *Consider the following protocol:*

$$\begin{array}{lll} \textit{Message } 1. & S \rightarrow A & : \{k_1\}_{SKey(A)} \\ \textit{Message } 2. & A \rightarrow S & : \{k_1,\, n_a\}_{SKey(A)} \\ \textit{Message } 3. & S \rightarrow A & : \{k_2,\, n_a\}_{SKey(A)} \end{array}$$

where k_1 and k_2 are fresh keys introduced by S, n_a is a fresh nonce introduced by A, SKey(A) is a previously established symmetric key known only to A and S. S modelled as internal is not factorisable, since message 3 relies on S receiving the previously introduced fresh value k_1. This violates the 3^{rd} condition of the factorisability definition.

■

Example 7.3 illustrates why we need the 3^{rd} condition of the factorisability definition. If S is factorisable, then there exists an independent run R_i of S for each fresh value v_i generated by S, where all other fresh values (generated by S in R_i) are dummy ones. This is not possible for the following reason. An externally modelled agent *Alice* playing role A will only generate message 2 in response to message 1, which must contain a genuine fresh value to be accepted (recall that external agents do not accept *dummy*). It follows that S cannot be factored into separate runs, each of which contains only one of k_1 and k_2.

The motivation for introducing the factorisability property is to help determine which protocols satisfy JIT. The relationship between these two properties is captured by Proposition 7.1.

Proposition 7.1 *Consider a CSP protocol model comprising a number of external agents and an internal role A, where A introduces fresh values of some data independent type T. If A is factorisable, then A satisfies JIT.*

Proof For role A to satisfy JIT, it must be the case that, for every trace ω in the system, there exists an equally valid trace ω' of the system such that ω' is externally equivalent to ω and all the values of type T are generated just-in-time in ω'.

Suppose ω is a trace of our system (with the factorisable internal role A). If all the values of type T are generated just-in-time in ω, then clearly our implication is satisfied. However, if this is not the case, then we can construct another trace ω' (valid within the system) that is externally equivalent to ω and satisfies JIT with respect to T, as follows.

When constructing ω', we need to consider every fresh value of type T in ω that is received by an external agent (in some message M) for the first time and is not generated JIT. Suppose t is such a value, M_t is the message that passes t on to an external agent $\mathbf{B}$ for the first time, and *MS* is the sequence of protocol messages, sent and received by external agents, that occur after the generation of t and before the receipt of M_t by $\mathbf{B}$. Furthermore, let G_t be the generation of an agent $\mathbf{A}$ taking role A that introduces the sequence $\mathbf{t}$ of fresh values which includes t and takes the form $\mathbf{t},\, X \,\vdash\, Y$, where X is the antecedent (set of input messages), known by the intruder (otherwise he would not be able to perform this generation!), and Y is the set of messages generated as a result.

By definition, no messages in MS contain t (since M_t is the first) and therefore none of them rely on the fact that G_t is introducing a fresh value t (as opposed to a dummy one). Their existence may, nevertheless, rely on G_t taking place; for example, the intruder may use other components in the set Y (resulting from G_t) to construct some of them (either directly or through further deductions being enabled), or prompt deductions or generations of $\mathbf{A}$ that follow on from G_t. Thus, when constructing ω' from ω, we cannot necessarily move this generation forward on the trace to satisfy JIT. We can, however, construct this sub-trace of ω' (with respect to t satisfying our property) as follows. Firstly, we replace the generation G_t with the generation G_{D_T}, whose only difference is that the dummy value D_T (for type T) is supplied instead of the fresh value t; G_{D_T} uses the same antecedent as G_t and therefore takes the form $\langle D_T \rangle, X \vdash Y'$ (Y' differs from Y above only in that all instances of t are replaced by D_T). Secondly, all instances of t in subsequent deductions and further generations within the intruder, that take place *before* the receipt of M_t by $\mathbf{B}$, are replaced by D_T. Thirdly and finally, by the definition of factorisability, we extend the trace with a new independent run R_t of $\mathbf{A}$ after the last message in MS and before the receipt of M_t by $\mathbf{B}$, such that the fresh value required in M_t, namely t, is generated on behalf of $\mathbf{A}$; any other values generated in R_t are bound to the dummy value. The intruder can simply replay the same messages he used earlier on in the trace, to prompt $\mathbf{A}$ (through the corresponding deductions and generations of $\mathbf{A}$) to perform R_t. The fact that S is factorisable means that the intruder is always able to achieve this, and therefore generate these fresh values just-in-time.

To construct ω', we simply repeat this process for every fresh value t of type T that is not generated just-in-time in ω.

$\blacksquare$

8 Bounding the intruder's appetite

The results of Sections 3–6 are the key to our approach to constructing CSP models within the scope of FDR that address the existence or otherwise of attacks that require a high degree of parallelism within agents. Furthermore, they allow a variety of sets of structural results implying factorisability and hence the capture of attacks requiring any degree of parallelism amongst internal protocol roles. This involves deriving bounds upon the number of fresh values the intruder may store at any one time (unknown to any external agents) and justifying them using Propositions 6.1 and 7.1.

The most obvious and trivial one of these is the case where (i) an internal role A generates 0 or 1 fresh value (of some data independent type T) per protocol run and (ii) any protocol message (within the system) contains at most 1 value of type T. It follows immediately from Definition 7.1 that A is factorisable and therefore by Proposition 7.1, A satisfies JIT. By Proposition 6.1, there exists a reduced (finite) system $System_R$, where the dummy-value strategy is implemented (condition 1) and the intruder is only allowed to store at most 1 value of type T (derived from (ii) above) at any given time, unknown to any external agent (condition 2). Since this reduced system is externally equivalent to the same system with an unbounded supply of fresh values of type T and an unrestricted intruder, $System_R$ will capture attacks for any degree of parallelism within A. This simple class would include, for example, server roles whose function is to supply agents with a fresh session key for every run or to simply re-compose messages (like the server in the TMN protocol).

An example of a more complex class is defined by the following proposition.

Proposition 8.1 *Consider a CSP model System(AS) for some protocol P, where AS is the set of roles in P. Suppose the role A in AS is modelled as internal, where A introduces fresh values of some type T. Suppose further that:*

1. *Each message M that can be sent on behalf of an agent $\mathbf{A}$ taking role A contains at most 1 value v of type T that was either freshly generated in M or previously freshly generated in a message on behalf of $\mathbf{A}$.*
2. *If $\mathbf{A}$ receives v of type T in some message M, where v was freshly introduced in a message sent on behalf of $\mathbf{A}$ earlier, then no subsequent message received or sent on behalf of $\mathbf{A}$ may contain v', where v' is a value of type T freshly introduced in some message generated on behalf of $\mathbf{A}$ and $v \neq v'$.*
3. *If $\mathbf{A}$ generates more than one fresh value of type T per run, then it checks that no value of type T it receives, apparently introduced by another agent, was in fact generated by $\mathbf{A}$ previously in the run.*[1]
4. *The intruder can store N fresh values of type T, unknown to any external agents, where N is the maximum number of values of type T in any single protocol message.*

If no attack is found upon System(AS) then no attack exists upon P for any degree of parallelism within A.

Proof As discussed earlier, if *System*(AS) is given an infinite supply of fresh values of type T, the intruder would be able to capture any degree of parallelism

[1] This assumption is required to ensure that non-factorisability of runs involving interventions by the intruder cannot slip in subtly. Such a case can arise when an internal agent $\mathbf{A}$ (as role A) expects to receive a value t_b of type T for the first time in some message M (from some agent $\mathbf{B}$), but instead $\mathbf{B}$ replays a value t_i that was previously freshly generated by $\mathbf{A}$. If $\mathbf{A}$ unknowingly receives its own value back, this could lead to a non-factorisable run of $\mathbf{A}$. In some protocols, in which $\mathbf{A}$ can neither decrypt nor reconstruct a relevant portion of a message, the implementation of this assumption may be impossible and so excludes the given protocol role from our result. Though this assumption violates the requirement for a positive deductive system, the distinctions required do not in fact invalidate our result because of the carefully controlled circumstances: We only implement this condition in our model for fresh values, which is where it is required. These values are never identified by our collapsing mappings with anything other than stale values, where the condition is not enforced. If, in a given protocol, this requirement is not implemented for whatever reason, this result does not exclude attacks in which $\mathbf{A}$'s own values are replayed back at it.

within the internal role A and therefore perform attacks, irrespective of the number of instances of A required. For this proposition to hold, it must be the case that every trace of such an infinite version of this system can be mapped to an externally equivalent trace in the reduced model, defined by the conditions presented above. We start by proving that A satisfies JIT and then justify how this reduced system is externally equivalent to the corresponding infinite version. To prove JIT, we make use of our factorisability property defined in Definition 7.1 and Proposition 7.1.

Suppose R is a run of an internal agent $\mathbf{A}$ taking role A represented by a sequence of deductions and generations DG_R within the intruder and resulting in the set of output messages MS_R. Furthermore, suppose that $t_1, \ldots, t_k$ are fresh values generated by the generations $G_1, \ldots, G_k$ of $\mathbf{A}$ respectively, in R. R can be factored into k independent runs of $\mathbf{A}$, according to the factorisability definition, as follows.

For each fresh value t_i ($\in \{t_1, \ldots, t_k\}$) generated by the generation G_i of $\mathbf{A}$, the intruder can construct an independent run R_i on behalf of $\mathbf{A}$, by performing the same sequence of deductions and generations in DG_R, where the fresh value t_i is indeed generated as fresh in G_i and all other generations of $\mathbf{A}$ are supplied with the dummy value D_T for type T. Consequently, all instances of t_i in the resulting output messages MS_{R_i} from run R_i will remain the same, whereas all instances of the other fresh values introduced in R on behalf of $\mathbf{A}$ will be mapped to D_T. In the case where $\mathbf{A}$ receives a message M that contains t_j ($i \neq j$), the run R_i is terminated just before M since by construction an external agent will never generate M with D_T substituted for t_j, meaning that the intruder may never be in possession of M. The intruder is able to perform these independent runs on behalf of $\mathbf{A}$ by simply re-using the same antecedents each time; this reflects the intruder replaying the same input messages to $\mathbf{A}$, k times. He can choose to perform these runs in any order, either sequentially or interleaved. We know, from condition 2 above, that $\mathbf{A}$ never relies on the receipt of previously introduced fresh values (by $\mathbf{A}$) in order to generate fresh values; therefore, $\mathbf{A}$ is not able to distinguish the runs with regards to the input stimuli given.

We must ensure that each output message in MS_R is present in one of the factored runs. By conditions 1 and 3 above, every protocol message M generated on behalf of $\mathbf{A}$ contain at most one value t_i of type T, where t_i was generated freshly on behalf of $\mathbf{A}$ (either in M or in a previous message). Whenever a run R_j is terminated just before message M, condition 2 ensures that M and every subsequent message is present in R_i. Therefore, for each fresh value t_i ($\in \{t_1, \ldots, t_k\}$), the messages in MS_R containing t_i will be generated in run R_i; the fact that the other values are mapped to dummy values does not affect these output messages (since they only contain one fresh value, namely t_i). Hence A is factorisable. By Proposition 7.1, A also satisfies JIT.

By Proposition 6.1, *System*(AS) with an infinite supply of fresh values of type T can be reduced to an equivalent one (traces are externally equivalent) with a finite source under the condition placed upon the intruder. Therefore, any attack that exists upon the protocol model that supplies agents taking role A with an infinite number of fresh values of type T (reflecting any degree of parallelism within A) and allows the intruder to store any number of them, can be mapped to an equivalent attack upon the reduced version of the system, where the intruder is bounded by condition 4 above.

■

9 Basing specifications on internal roles

When an agent **A** is modelled as a standard external process, signal events (capturing the state of mind of **A** for specification purposes) can be constructed through the appropriate renaming of messages sent and received by **A**. However, an agent **B** playing an internal role no longer performs *send* and *receive* events, since its functionality is solely captured within the intruder's deductive system.

Capturing the sending of a message M by an internal agent **B** is relatively straightforward, as this corresponds to the deduction or generation of **B** resulting in M. On the other hand, constructing signal events for **B** that are bound to the *receiving* of some message M by **B** is more complicated, since the intruder's deductive system does not directly capture this information. A solution to this is to ensure (artificially if necessary) that such receipts are immediately followed by the same agent performing some *send*.

We cannot, however, simply adopt these signals into the framework we have been using to date in which all signals come from external agents' actions. This is because the JIT property has been based on *external* equivalence, which preserves the events relevant to external-agent specifications, but not the ones used for internal-agent ones. The process of transforming a trace to be JIT has to be examined so that we understand what happens to the events relevant to signals. Exactly the same transformations we have performed generally suffice, observing that

- Each generation or deduction that happened in the original trace t has at least one analogue in the transformed trace t', each of which is the same except that perhaps data independent values are dummy. One of them is at the original point in the trace, all others follow this, and the last one creates the value from t.

These transformations are unlikely to create false attacks, since (with the dummy values being replaced by fresh values) the trace t' will usually be one of a system with infinitely many fresh values. But the real concern is that they might eliminate an attack. While the discussions below will show that no significant problems arise there from internally-based signals, care will be needed in future if specifications involving more complex patterns of events are considered.

10 Towards a more complete analysis

Being able to construct signal events for internal agents, strengthens our work even further, since it allows our models to capture parallelism amongst agents that are part of a specification. Further still, it opens potential avenues for capturing more complete results upon protocol analysis, than simply parallelism of internal agents. This involves carefully selecting which protocol roles are internalised and which are modelled as external; this is dependent on the specification being verified. In this section, we outline how this can probably be achieved for secrecy and authentication properties in turn, though we emphasise that the establishment of exact criteria for these claims to apply is work in progress.

10.1 Secrecy specifications

A specification of the form $\mathit{Secret}(A, s, [B_1, \ldots, B_n])$ specifies that in any completed run, an agent **A** playing role A believes that the value bound to s is a secret shared only with agents playing the roles $B_1, \ldots, B_n$. We require only one type of signal event within our protocol model when verifying these specifications; it is constructed through the standard renaming of the last message *received* by **A**. (The same applies to other types of secrecy specifications defined by Lowe [13]. For the purposes of illustration in this paper we will only consider the standard one.)

Suppose we model a protocol P with the set of roles $AS = \{A, B_1, \ldots, B_n\}$ and verify specifications of the form $\mathit{Secret}(A, s, [B_1, \ldots, B_n])$ for some secret s, as follows:

1. All roles in AS are modelled as internal.
2. One instance of role A is modelled as an external agent process.
3. The necessary signal events are constructed for the external instance of A; none are constructed for any internal instances of A.

We believe that, under appropriate bounds upon the intruder (such as that derived in Proposition 8.1), if no secrecy attack is found in this model (in which the intruder can exploit an additional identity as is normal), then none exists upon P for any degree of parallelism within AS.

We justify this claim as follows. The functionalities of all the agents in AS are internalised within the intruder and therefore, given that we have calculated an appropriate bound upon him, this represents an unbounded degree of parallelism within them (for the same reasons described in the earlier propositions). By definition, a secrecy specification $\mathit{Secret}(A,\ s,\ [B_1, \ldots, B_n])$ is broken precisely when there exists an instance of A who believes that the value v bound to s is shared only with $B_1, \ldots, B_n$, while in fact the intruder has been able to deduce it v. It does not matter which instance of A this is; the only requirement is that there exists one of them. Our model has one external instance **A** of A with the appropriate signal event linked to it. The point is that since in the real world all instances of A are symmetric, if there is an attack on the protocol then there is one in which the "complaining" instance of A is the one which is mapped to **A** under the reduction process.

The transformations to make a trace JIT cannot affect the truth or falsity of the *Secret* specification as **A**'s actions are unchanged and if the intruder knows the value bound to s in the original, he will eventually know it in the transformed trace.

Our current limitation with this result is that we have not derived bounds upon the intruder for all general cases. However, deriving such bounds is a current area of our future research, thereby making a more complete analysis of protocols as presented above, a feasible achievement. Furthermore, we have currently only considered internalising one role at a time; extending this to internalise multiple roles is part of future research.

10.2 Authentication specifications

We can apply a similar approach for capturing non-injective authentication properties for any degree of parallelism within the models.

The authentication specification $Auth(A, B, [x_1, \ldots, x_n])$ specifies that if an agent **B** playing role B thinks he has successfully completed a run of the protocol with some agent **A** as role A, then **A** has previously been running the protocol, apparently with **B** and both agents agree as to which roles they took and the values bound to $x_1, \ldots, x_n$. Two types of signal events are required: *Signal.Running* and *Signal.Commit* events. The former of these is identified with the last message sent by **A** which affects **B** either directly or indirectly and the latter is bound to the last message participated by **B**.

Suppose we model a protocol P with the set of roles AS and verify specifications of the form $Auth(A, B, [x_1, \ldots, x_n])$, where A and B are members of AS, as follows:

1. All roles in AS are modelled as internal.

2. One instance of role B is modelled as an external agent process.

3. The *Signal.Commit* events are constructed for the external instance of B only; none are constructed for any internal instances of B.

4. The *Signal.Running* events are constructed for the internal instances of A.

We believe that, under appropriate bounds upon the intruder (such as those derived in the earlier propositions), if no authentication attack is found in this model, then none exists upon P for any degree of parallelism within AS.

We justify this claim as follows. The functionalities of all the roles in AS are internalised within the intruder and therefore, given that we have calculated appropriate bounds upon him, this represents an unbounded degree of parallelism within them. We know that the specifications in question do not demand a one-to-one relationship between the runs of the two agents being verified. An attack will be found upon these specifications if there exists at least one case where some instance of role B (and we don't care which!) commits himself to a run of the protocol believing he has done so with an agent **A** as role A, when in fact **A** has not been running the protocol. For each specification, it does not matter which instance of role A performs the necessary *Signal.Running* as long as there exists one of them that is in that state. Therefore, we can internalise all instances of A within the intruder and capture the *Signal.Running* events that we are interested in through the corresponding deductions and generations. Furthermore, it does not matter which instance of B participates in the run that leads to the authentication attack; any authentication attack the intruder can perform upon an internalised instance of B, he can also perform upon the one external instance $\mathbf{B}$ of B. It therefore suffices to have *Signal.Commit* events for $\mathbf{B}$ only and none associated to any internal instances of B.

Again, the use of internal deductions or generations as the *Signal.Running* event is not damaged by the transformations to achieve JIT, since the occurrence of such an event may be duplicated and/or moved to a later time but is never removed or moved earlier. These are the only things that could remove a failure of the authentication specification.

11 Conclusions

As well as proving to be a highly effective state space reduction strategy (by 2 orders of magnitude [3]), we have shown that the internal protocol role model frequently permits protocols to be analysed with some roles having an arbitrary degree of parallelism. An example protocol we used for testing purposes was an extended version of the hypothetical *ffgg* protocol by Millen [15], where the secrecy attack requires three instances of an initiator agent running in parallel, together with a single instance of a responder. Using old modelling techniques, this model is infeasible to run; using our new techniques and internalising the initiator role, this attack was found very easily. Details concerning this model and other examples can be found in [3].

Proposition 8.1 defines the class of protocol roles that we are currently able to internalise and verify for an arbitrary degree of parallelism within a finite refinement check. To give an intuitive overview of the protocol roles that satisfy this proposition, we consider some example protocols in the survey by Clark and Jacob [8]. All protocol roles are factorisable and satisfy the conditions presented in Proposition 8.1 in the following protocols:

- All versions of the Andrew Secure RPC, Kao Chow Authentication (v.1 and v.2), Needham Schroeder Public Key, Needham Schroeder Symmetric Key, Yahalom, Woo and Lam Mutual Authentication, SK3 and TMN.
- The Otway Rees protocol, provided that the index value introduced by the initiating role is treated as a distinct type from the type of the nonces.
- All versions of the Wide Mouthed Frog and Denning Sacco shared key[2].

[2] In our previous work, we did not address the issue of modelling timestamps in our CSP models when applying the data independence techniques, since timestamps are not of data independent type. We have since established that timestamps can be modelled within this framework, but this work is beyond the scope of this paper.

There are a few protocol examples in this survey with protocol roles that do not satisfy the conditions in Proposition 8.1, for example, the KSL and Neumann Stubblebine protocols. In both cases, the initiator A and responder B generate fresh values of type *number* in the latter part of the protocol that depend upon the receipt of values of type *number* that they previously generated as fresh. This violates condition 2 of Proposition 8.1.

Further work planned includes broadening classes of conditions and specifications where we can use these techniques, wherever possible on all the identities present in a given protocol.

Blanchet [1] uses a similar idea to ours, especially with regards to the internalisation strategy of protocol roles (referring to the early stages of this development in [5]). The author presents a prolog based framework with the following two abstractions: (i) fresh values are represented as functions over the possible pairs of participants and (ii) a protocol step can be executed several times, instead of only once per session. With these abstractions, the author is able to capture unbounded agent runs and degrees of parallelism within their identities. Similar to ours, his analysis is fail-safe in the sense that no attacks are lost, but the potential for false attacks does arise. However, by modelling values that are expected to be fresh for every run as functions over the participants, it is no longer possible to distinguish between old values used in previous ones and new. This means that one cannot verify properties that depend upon freshness. In his initial work, Blanchet only considered secrecy specifications; more recently, this has been extended to handle authentication properties [2].

In the later sections of this paper we have envisaged protocol models in which all roles are internalised (though perhaps additionally having external copies of some). The concept of an intruder with internalised copies of all roles, thereby using them all as oracles within the rules of the protocol, is very close to the concept of the kernel of a rank function in the sense used in [22]. More precisely, the comparison is between such kernels and the sets which our intruder can generate at various points in its execution, such as the set of messages and sub-terms that can be generated without external intervention; the internalised roles provide similar closure to that required for one of these kernels. The only real exceptions to this relate to values introduced for or by external agents, and the additional constraints imposed by our supervisor processes. We are sure there is some interesting comparative work to be done here, and perhaps also with the concept of an *ideal* in a Strand Space [9].

Acknowledgements

This work has benefited enormously from discussions with Gavin Lowe. We also thank the anonymous referees for their useful comments and suggestions, which have led to significant improvements in this paper. This research was supported by funding from US ONR, DERA/QinetiQ and EPSRC.

References

[1] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE Computer Society, 2001.

[2] B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium*, pages 242–259. Springer-Verlag, 2002.

[3] P. J. Broadfoot. *Data independence in the model checking of security protocols*. D.Phil, Oxford University, 2001.

[4] P. J. Broadfoot and G. Lowe. Analysing a stream authentication protocol using model checking. In *7^{th} European Symposium on Research in Computer Security*, LNCS. Springer, 2002.

[5] P. J. Broadfoot, G. Lowe, and A. W. Roscoe. Automating data independence. In *6^{th} European Symposium on Research in Computer Security*, volume 1895 of *LNCS*, pages 175–190. Springer, 2000.

[6] P. J. Broadfoot and A. W. Roscoe. Capturing parallel attacks within the data independence framework. In *15^{th} IEEE Computer Security Foundations Workshop*, pages 147–159. IEEE Computer Society, 2002.

[7] P. J. Broadfoot and A. W. Roscoe. Internalising agents in CSP protocol models (extended abstract). *Workshop on Issues in the Theory of Security (WITS '02)*, 2002.

[8] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature: Version 1.0. Technical report, University of York, 1997.

[9] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand Spaces: Why is a security protocol correct? In *Proceedings of IEEE Symposium on Security and Privacy*, pages 160–171, 1998.

[10] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR2 User Manual*, 2000.

[11] J. A. Heather and S. A. Schneider. Equal to the task? In *7^{th} European Symposium on Research in Computer Security*, LNCS. Springer, 2002.

[12] R. S. Lazić. *Theorems for mechanical verification of data-independent CSP*. D.Phil, Oxford University, 1999.

[13] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

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

[15] J. Millen. A necessarily parallel attack. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[16] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6, 1998.

[17] A. Perrig, R. Canetti, J. D. Tygar, and D. X. Song. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Symposium on Security and Privacy*, pages 56–73, 2000.

[18] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *11th IEEE Computer Security Foundations Workshop*, pages 84–95, 1998.

[19] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[20] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.

[21] P. Y. A. Ryan, S. A. Schneider, M. H. Goldsmith, G. Lowe, and A. W. Roscoe. *Modelling and Analysis of Security Protocols.* Pearson Education, 2001.

[22] S. A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 1998.

[23] D. X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[24] M. Tatebayashi, N. Matsuzaki, and D. B. Newman. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology: Proceedings of Crypto '89*, volume 435 of *LNCS*, pages 324–333. Springer-Verlag, 1990.