Capturing parallel attacks within the data independence framework

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 [3, 14, 12] on the application of data independence to the model checking of cryptographic protocols using CSP [13] and FDR [5]. In particular, we showed how techniques based on data independence [7, 13] 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 [3], namely by "internalising" all or part of each agent identity within the "intruder" process. The internalisation of agents (initially only server roles) was introduced in [14] 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 agents do introduce fresh values and address the issue of capturing the state of mind of internal agents (for the purposes of analysis).

1. Introduction

We carry forward the work described in our previous papers [3, 14, 12] on the application of data independence to the model checking of cryptographic protocols using CSP [13] and FDR [5], often via extensions to Casper [8]. 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 [9, 11, 15].

In our previous papers we showed how techniques based on *data independence* [7, 13] 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 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 [3], namely by "internalising" all or part of each agent identity within the "intruder" process. The internalisation of agents (initially only server roles) was introduced in [14] 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 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 agents so that the sequentiality problem disappeared.

- The first is that an internalised agent (or server) 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.

- 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 this or included specific signals (to the environment) in the definitions of processes representing trustworthy agents. This is not an issue for server processes, but it is much harder to "get into the minds" of internalised agents, something necessary if their progress on protocol runs plays a part in our specification.

This paper is an extended version of our paper [4] (extended abstract presented at WITS'02 with no formal proceedings). We present the techniques we have evolved for internalising agents, as well as the solutions we have devised for the two problems described above. Our aim is to give the reader an understanding of the most important ideas and definitions behind our work. However there is not space here for many technical details and proofs. These can all be found in the first author's D.Phil. thesis [2].

2. Data independence techniques

The data independence techniques allow 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 when requested. An observation this method relies upon is that a trustworthy agent (or server) will typically 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. We say that a fresh value v is *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 new messages seen across the network. 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. We refer to this mapping process as the *recycling mechanism* of fresh values. Once such a value has been recycled, the corresponding manager process can supply it to the network as fresh again. 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 [12, 14], in the sense that any attack that exists on 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 [14]. We have recently been interested to see that this condition is proving necessary for protocol analysis within the rank functions and the strand spaces framework [6].

3. Internalising agent 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 and the server (if any). Therefore placing either a server (alternatively known as "trusted third party") or an agent we wish to trust within the intruder seems bizarre. However that is not really what we are doing, which is to replace an agent/server with a set of inferences of the style used within our coding of the intruder that reflect 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*.

Deductions performed by the intruder are usually modelled by pairs of the form (X, 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 internal agents that do not introduce any fresh values is captured by this type of deduction within the intruder: we get a deduction (X, f) if, after the agent is told the messages in X, it can be expected to emit f (where f will be functionally dependent on X). The server role in the TMN protocol [16] 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 (and so not introducing any fresh variables into the system). If we modelled this server role internally, then the corresponding deductions would be all valid instantiations of $\{M_1, M_3\} \vdash M_4$.

Internal agents that do introduce fresh values are captured by a special type of deduction, known as a *generation*. A *generation* has the form (t, X, Y), where 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 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 t.

Example 3.1 *Consider the following hypothetical protocol description:*

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

where A is an agent introducing the fresh nonce n_a and S is a server introducing the fresh key k_{ab}. 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, we say that an agent A is internal when A's functionality is captured within the intruder component by a series of representative deductions and generations. We say that an agent A is external otherwise (i.e. when 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. This is done (see [2]) by means of a special class of constraint processes called *Supervisors*. These are designed to ensure that the internal agent'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 agents internally within the intruder. The first is that this approach serves as an effective state space reduction technique (as discussed and illustrated in [2]). The second advantage, and one we will be focusing on in this paper, is that the internal model of an agent A naturally captures a highly parallelised version of A. If A does not introduce any fresh values (for example, the server in the TMN protocol), then the intruder is able to capture any degree of parallelism of A by performing the standard deductions on behalf of A. On the other hand, if A introduces fresh values, then the degree of parallelism of A that the intruder can capture is dependent on the supply of fresh values. In a model where there is an infinite supply, the intruder is able to capture any degree of parallelism of 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 in Example 3.1. If the key manager is given a set of n fresh values to supply the network and the intruder has intercepted the message 1 $\{Bob, N_A\}_{SKey(Alice)}$, then the intruder could legitimately perform the following sequence of generations:

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

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 agents that generate fresh values raises 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 agents? We address these questions in this paper.

4. Just-in-time principle

In this section, we introduce a protocol model property, referred to as *just-in-time*. This property allows us to derive

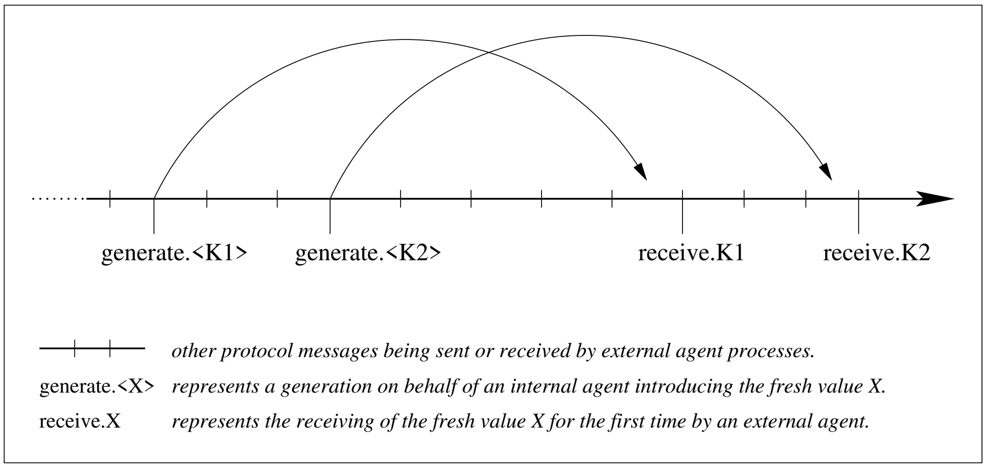

Figure 1. Satisfying the just-in-time property.

and justify finite bounds upon the intruder that prevent him from requesting an unbounded supply of fresh values, without weakening our analysis (namely, not losing any attacks). We firstly introduce a simple definition of equivalence.

Definition 4.1 (External equivalence) *We say that two traces ω and ω' are "externally equivalent" precisely when all behaviour involving external agents is identical in both traces, namely:*

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

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

■

Definition 4.2 (Just-in-time) *Suppose we have a CSP protocol model with a number of externally modelled agents, together with an internal agent S, where S introduces fresh values of some type T.*

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

We say that 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 the just-in-time property 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 the just-in-time property, 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 4.1 (Satisfying just-in-time) *Consider the hypothetical protocol description presented in Example 3.1, where the server S is modelled as internal. Suppose there are 2 instances of A declared as external agent processes, all given the identity Alice. Consider the following valid sequence of events:*

- *Message* 1. $\mathit{Alice}_1 \rightarrow I_S \; : \; \{\mathit{Bob}, N_1\}_{\mathit{SKey}(\mathit{Alice})}$
- *Message* 1. $\mathit{Alice}_2 \rightarrow I_S \; : \; \{\mathit{Bob}, N_2\}_{\mathit{SKey}(\mathit{Alice})}$
- *Generation 1:* $\langle K_1 \rangle, \; \{\mathit{Bob}, N_1\}_{\mathit{SKey}(\mathit{Alice})} \; \vdash \; \{N_1, K_1\}_{\mathit{SKey}(\mathit{Alice})}$
- *Generation 2:* $\langle K_2 \rangle, \; \{\mathit{Bob}, N_2\}_{\mathit{SKey}(\mathit{Alice})} \; \vdash \; \{N_2, K_2\}_{\mathit{SKey}(\mathit{Alice})}$
- *Message* 2. $I_S \rightarrow \mathit{Alice}_1 \; : \; \{N_1, K_1\}_{\mathit{SKey}(\mathit{Alice})}$

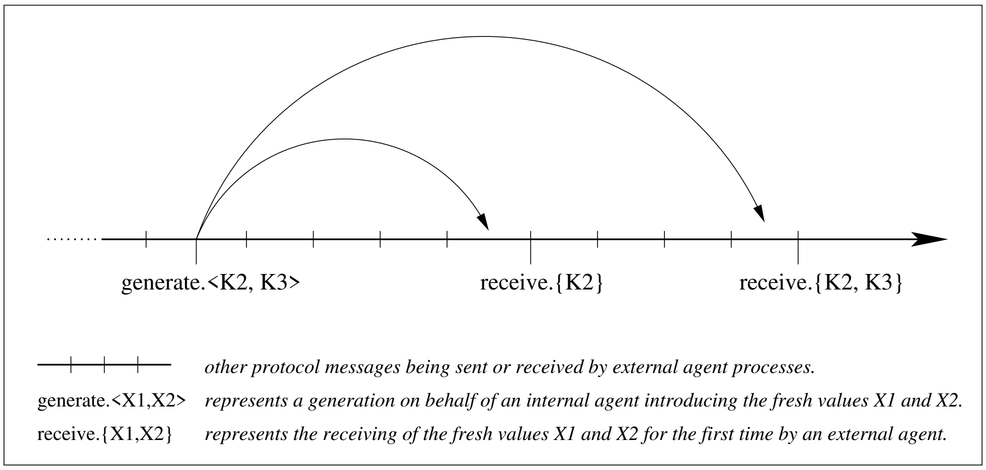

Figure 2. Violating the just-in-time property.

- *Message* 2. $I_S \rightarrow Alice_2 \; : \; \{N_2, K_2\}_{SKey(Alice)}$

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

In this trace, the intruder is not generating the fresh value K_2 just-in-time. However, we can construct an equally valid trace of the system that does and is externally equivalent to the trace above (as illustrated in Figure 1):

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

■

Example 4.1 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 by the internal agent. At any point, he 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, as soon as we start looking at larger protocol examples, determining whether a given protocol satisfies the just-in-time principle is much less intuitive and often very complex. Example 4.2 gives an example of a protocol that does not satisfy our property.

Example 4.2 (Violating just-in-time) *Consider a protocol defined by the following sequence of messages:*

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

where k_1, k_2 and k_3 are keys introduced freshly by the agent A and the server 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. Suppose S is modelled as internal and captured by the generation:

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

Suppose further that the system is composed of one external instance each of 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: (on behalf of S)* $\langle K_2, K_3\rangle, \{Bob, I_1\}_{SKey(Alice)} \vdash \{K_2, K_3, I_1\}_{SK(S)}$

4. *Intruder deduces $\{K_2, Alice\}_{PK(Bob)}$ using the message generated in the previous step.*

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 the just-in-time property 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.*

In order to construct an externally equivalent trace that satisfies the just-in-time property, we need to be able to construct a trace such 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 message 1 to be the same as that bound to k_2; in this trace he uses the K_2 generated on behalf of the server, as illustrated in Figure 2.

■

Example 4.2 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 the just-in-time principle). 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 messages 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.

Being able to determine whether a protocol model satisfies our just-in-time property 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 internal agents' generations and justify that no behaviour of the system is lost.

5. 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 just-in-time 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 an *equivalent* reduced system with a finite source of fresh values. By equivalent, we mean that no attacks are lost through the mapping; thus, if no attack is found upon the reduced system, then we can conclude that none exists upon the original infinite version of the system.

5.1. Dummy values

The just-in-time 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 (in terms of enabling deductions). 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 storing 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 towards attacking them.

Distinguishing between 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 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 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 (t, X, 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 7, 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 [14] 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 the implementation of dummy values within our CSP protocol models (as described above) as the *dummy-value strategy*.

5.2. Constructing an equivalent reduced model

Given a protocol model that satisfies the just-in-time property and has an infinite supply of fresh values of the data independent types generated, we can construct an equivalent reduced model where there is only a finite source of fresh values and that incorporates the dummy values for the relevant types. By equivalent, we mean that for every trace in the original infinite model, we can find 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 5.1 captures this more formally.

Proposition 5.1 *Suppose System(AS) is a protocol model with the set of roles AS, where some 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 the just-in-time property, then we can construct 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.*

A formal proof is presented in [2].

■

We illustrate this mapping more intuitively in the following example.

Example 5.1 *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 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:* $\langle K_2\rangle,\ \{\{N_A\}_{SKey(Alice)}, \{K_1, N_A\}_{SKey(Alice)}\} \vdash \{K_2, N_A\}_{SKey(Alice)}$
- *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 the just-in-time property, since K_1 is not generated just-in-time. 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:*
  $\langle K_2 \rangle,\ \{\{N_A\}_{SKey(Alice)},\ \{K_3, N_A\}_{SKey(Alice)}\} \vdash \{K_2, N_A\}_{SKey(Alice)}$

- *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:*
  $\langle K_2 \rangle,\ \{\{N_A\}_{SKey(Alice)}, \{K_D, N_A\}_{SKey(Alice)}\} \vdash \{K_2, N_A\}_{SKey(Alice)}$

- *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 5.1 provides a useful and simple example of how we can map a trace in the original protocol model (with potentially infinite supply of fresh values) 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 agents. By being able to map an infinite model with an internal agent A to an equivalent reduced one (for protocols that satisfy the just-in-time property), means that we are able to capture attacks upon protocols for any degree of parallelism within A by performing a finite refinement check.

There are two main questions that we still need to consider. The first is how we determine whether a given protocol model satisfies the just-in-time property. The second is, given a protocol model that satisfies this property and makes use of the dummy values, what should the finite bound upon the intruder be to prevent him from using fresh values for internal purposes only, while still ensuring that he can perform all possible interactions with the external agent processes? We address these questions in the remaining part of this paper.

6. Factorisability of internal agents

We now introduce a new property, referred to as the *factorisability* of internal agents, and show that when satisfied by an internal agent A within a protocol model, we can apply the just-in-time principle to justify the bounds placed upon the intruder with regards to the fresh values introduced on behalf of A (via the corresponding generations).

The factorisability property is quite a restrictive one and there are many protocol examples that do not satisfy it. However, it has proved to be extremely useful towards determining and justifying finite bounds upon the intruder with regards to the number of fresh values he can request through internal agent generations.

Definition 6.1 (Factorisability) *We say that an internal agent A is factorisable with respect to some data independent type T precisely when, for each run R of A that generates fresh values $v_1, \ldots, v_k$ of type T, the following conditions are satisfied:*

1. *We can construct runs $R_1, \ldots, R_k$ of A, where each run R_i contains the fresh value v_i and the dummy value 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 A, where $v_i \neq v_j$: if A receives v_i back in some protocol message, then no subsequent message sent or received by A contains v_j.*

■

An intruder with an internalised factorisable agent A is equivalent to one in which A has been replaced by A', where A' can only deliver one fresh value per independent run.

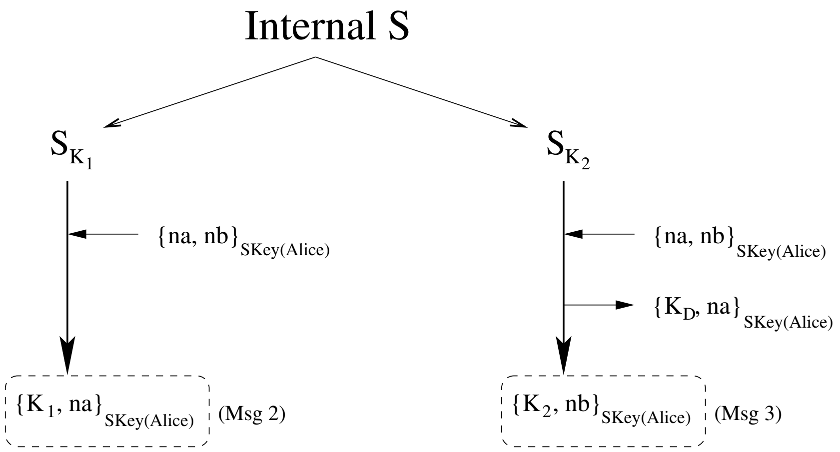

Figure 3. Factorisable internal server S in Example 6.1.

Thus, the ability for the intruder to store messages that contain fresh values (generated on behalf of some internal agent 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 an internal agent 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.

To illustrate the factorisability property more clearly, consider the following example of a simple protocol whose internal server is factorisable.

Example 6.1 (Factorisable internal server) *Consider the following protocol:*

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

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

S modelled as internal is factorisable, since it satisfies the conditions required in Definition 6.1. As illustrated in Figure 3, 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} requirement of the factorisability definition.

■

Example 6.1 provides a simple example of a factorisable internal server 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 using the dummy values (as illustrated in Figure 3) and the fact that, by definition, he is able to simply replay the external agent A's message 1 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.

The motivation for introducing the factorisability property is to help determine which protocols satisfy the just-in-time property. The relationship between these two properties is captured by Proposition 6.1.

Proposition 6.1 *Consider a CSP protocol model comprising a number of external agents and an internal agent A, where A introduces fresh values of some data independent type T. If A is factorisable, then A satisfies the just-in-time property.*

Proof For A to satisfy the just-in-time property, it must be the case that, for every trace ω in the system, we can construct 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 agent 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' that is externally equivalent to ω and satisfies the just-in-time property 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 B (in some message M) for the first time, but arose in a generation of A before the last protocol message preceding the receipt of M by B. Suppose t is such a value, M_t is the message that passes t on to any external agent 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 B. Furthermore, let G_t be the generation of A that introduces the fresh value t and takes the form $\langle t \rangle,\ X\ \vdash\ Y$, 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, we know that 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 construction 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 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 the just-in-time property. 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 $\langle G_{D_T} \rangle,\ X\ \vdash\ Y'$ (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 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 A after the last message in MS and before the receipt of M_t by B, such that the fresh value required in M_t, namely t, is generated on behalf of 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 A (through the corresponding deductions and generations of 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 not generated just-in-time in ω.

■

7. Bounding the intruder's appetite

The results of Sections 3–5 are the key to constructing CSP models within the scope of FDR that address the existence or otherwise of attacks that require a high degree of parallelism in 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 agents. The 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 Proposition 5.1.

The most obvious and trivial one of these is the case where (i) an internal agent 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 6.1 that A is factorisable and therefore by Proposition 6.1, A satisfies the just-in-time property. By Proposition 5.1, we can construct a reduced (finite) system System_R by implementing the dummy-value strategy (Condition 1) and only allowing the intruder 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 recompose messages (like the server in the TMN protocol).

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

Proposition 7.1 *Suppose System*(AS) *represents the CSP model for some protocol* P*, where AS is the set of roles in* P*. Suppose further that we model the role* A *in AS as internal, where* A *introduces fresh values of some type* T*. We claim that if the following conditions are satisfied:*

1. *A introduces at most 1 value of type T per message.*
2. *The progress of A never depends upon the receipt of previously introduced values of type T on behalf of A.*
3. *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.*

then no attack found upon System(AS) *implies that 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 within the internal agent 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 prove this by firstly proving that A satisfies the just-in-time property and then justifying how this reduced system is externally equivalent to the corresponding infinite version. To prove the just-in-time property, we make use of our factorisability argument and Proposition 6.1.

Suppose R is a run of A represented by a sequence of deductions and generations DG_R performed by the intruder on behalf of A, 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 A respectively, in R. R can be factored into k independent runs of 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 A, the intruder can construct an independent run R_i on behalf of 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 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 A will be mapped to D_T. The intruder is able to perform these independent runs on behalf of A by simply re-using the same antecedents each time; this reflects the intruder replaying the same input messages to A, k times. He can choose to perform these runs in any order, either sequentially or interleaved. We know, from condition *2* above, that A never relies on the receipt of previously introduced fresh values (by A) in order to generate fresh values; therefore, 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. This is the case, since by definition (condition *1* above), all protocol messages generated on behalf of A contain at most one value of type T. 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 6.1, A therefore also satisfies the just-in-time property.

By Proposition 5.1, we can conclude that the version of *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 the intruder, since (i) *System*(AS) satisfies the just-in-time property, (ii) the dummy-value strategy is implemented and (iii) the number of values of type T that the intruder is able to store (unknown to any external agents) is equal to the maximum number of them an external agent can learn for the first time. Therefore, any attack that exists upon the protocol model that supplies A with an infinite number of fresh values of type T (reflecting any degree of parallelism of 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 3 above.

■

8. Basing specifications on internal agents

When an agent A is modelled as a standard external process, signal events (capturing the state of mind of A for specification purposes) are constructed through the appropriate renaming of messages sent and received by A. An internal agent 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 A is relatively straightforward, as this corresponds to the deduction or generation of A resulting in M. On the other hand, constructing signal events for an internal agent A that are bound to the *receiving* of some message M by A 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*.

It is thus possible to verify authentication specifications where those roles satisfying the just-in-time principle are arbitrarily parallel. Such proofs do not at present exclude one-to-many attacks in which A and B can think they are having different non-zero numbers of runs with each other.

9. 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 agent roles are internalised and which are modelled as external; this is dependent on the specification being verified. In this section, we suggest how this could potentially be achieved for secrecy and authentication properties in turn. This work is part of future research we are interested in pursuing.

9.1. Secrecy specifications

A specification of the form $Secret(A, s, [B_1, \ldots, B_n])$ represents the following property: A believes that the value bound to variable s is a secret shared only with the agents $B_1, \ldots, B_n$. We require only one type of signal event within our protocol model when verifying these specifications; it is identified with the last message *received* by A. (The same applies to other types of secrecy specifications defined by Lowe [8]. For the purposes of illustration in this paper we will only consider the standard one.)

Suppose we are interested in modelling a protocol P with the set of roles *AS*, and verifying secrecy specifications of the form $Secret(A, s, [B_1, \ldots, B_n])$, where $A, B_1, \ldots, B_n$ are members of *AS*. Suppose further that we construct this CSP model as follows:

1. All agents in *AS* that are modelled as internal.
2. One instance of A is modelled as an external agent process.
3. The necessary signal events (required for A only) 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 7.1), if no secrecy attack found in this model, then none exists upon P for any degree of parallelism amongst *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 amongst them (for the same reasons described in the earlier propositions). By definition, a secrecy specification $Secret(A, s, [B_1, \ldots, B_n])$ 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 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_E of A with the appropriate signal event linked to it, and so any attack the intruder can perform upon an internalised instance of A, he can also perform upon A_E. Therefore, it suffices to have signal events for the secrecy specifications linked only to A_E, and none associated to the internal instances of A.

Our current limitation with this result is that we have not derived bounds upon the intruder for all general cases; our current propositions are somewhat limited when internalising multiple agent roles. 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.

9.2. Authentication specifications

We can apply a similar approach for capturing authentication properties for any degree of parallelism within the models. The class of authentication specifications we consider are those that do not demand a one-to-one relationship between the runs of the agents in question (the reason for this becomes apparent later on).

The authentication specification $Auth(A, B, xs)$ represents the property: A must be authenticated to B and agree on the values bound to the variables in xs. 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 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 authentication specifications of the form $Auth(A, B, [x_1, \ldots, x_n])$, where A and B are members of *AS*. Suppose further that we construct this CSP model as follows:

1. All agents in AS that are modelled as internal.
2. One instance of B is modelled as an external agent process.
3. The *Signal.Commit* events (required for B) are constructed for the external instance of B only; none are constructed for any internal instances of B.
4. The *Signal.Running* events (required for A) are constructed for all 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 found in this model, then none exists upon P for any degree of parallelism amongst *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 appropriate bounds upon him, this represents an unbounded degree of parallelism amongst 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 if some instance of B (and we don't care which!) commits himself to a run of the protocol believing he has done so with A, when in fact A has not been running the protocol (according to the definition of one of the three possible specification we are considering). For each specification, it does not matter which instance of 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 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 B, he can also perform upon the one external instance B_E of B. It therefore suffices to have *Signal.Commit* events for the linked only to B_E, and none associated to the internal instances of B.

We expect that similar arguments to those used earlier for a single internal agent will allow the restriction of the intruder's appetite for fresh values to manageable proportions in many cases, but this, together with widening the range of specifications, remains work in progress.

10. Conclusions

As well as proving to be a highly effective state space reduction strategy (by 2 orders of magnitude [2]), we have shown that the internal agent model frequently permits protocols to be analysed with some agents 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 [10], where the secrecy attack requires three instances of an initiator agent, 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 [2].

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 agents (referring to the early stages of this development in [3]). 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 failsafe 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. Blanchet currently only considers secrecy specifications.

Acknowledgements

This work has benefited enormously from discussions with Gavin Lowe. It was supported by funding from US ONR, DERA and EPSRC. We also thank the anonymous referees for their useful comments and suggestions.

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 Press, 2001.

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

[3] P. J. Broadfoot, G. Lowe, and A. W. Roscoe. Automating data independence. In *Computer Security - ESORICS 2000*, volume 1895 of *LNCS*, pages 175–190. Springer, 2000.

[4] 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.

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

[6] J. A. Heather and S. A. Schneider. Equal to the task? *Submitted for publication*, 2002.

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

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

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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.