Specifying authentication using signal events in CSP

Siraj A. Shaikh *(first and corresponding author)*
Department of Information Systems, Cranfield University, Defence Academy of the United Kingdom, Shrivenham, SN6 8LA, UK, email: siraj.shaikh@gmail.com

Vicky J. Bush
Department of Computing, UGBS, University of Gloucestershire, Park Campus, Cheltenham Spa, GL50 2RH, UK

Steve A. Schneider
Department of Computing, SEPS, University of Surrey, Guilford, Surrey, GU2 7XH, UK

Abstract. The formal analysis of cryptographic protocols has developed into a comprehensive body of knowledge, building on a wide variety of formalisms and treating a diverse range of security properties, foremost of which is authentication. The formal specification of authentication has long been a subject of examination. In this paper, we discuss the use of correspondence to formally specify authentication and focus on Schneider's use of signal events in the process algebra Communicating Sequential Processes (CSP) to specify authentication. The purpose of this effort is to strengthen this formalism further. We develop a formal structure for these events and use them to specify a general authentication property. We then develop specifications for recentness and injectivity as sub-properties, and use them to refine authentication further. Finally, we use signal events to specify a range of authentication definitions and protocol examples to clarify their use and make explicit related theoretical issues. Our work is motivated by the desire to effectively analyse and express security properties in formal terms, so as to make them precise and clear.

Keywords: authentication, security protocols, CSP, formal specification, Kerberos

1. Introduction

Schneider [1] uses Communicating Sequential Processes (CSP) [2] to model cryptographic protocols. The protocol participants are modelled as independent processes, interacting with each other by exchanging messages. Different roles are modelled as different processes, for example, initiator, responder and server. Schneider (1998) attributes CSP to be effective for modelling protocols as "*it is close to the level we think of them*". The use of CSP to model this type of parallel message-passing distributed system captures the precise specification of a cryptographic protocol and is extensible as different aspects of protocol modelling can be included. Schneider [1] takes advantage of this feature and introduces additional control events to help in the analysis. These events, called *signals*, are introduced in the model in terms of protocol participants and messages. Signals are used to express security properties, especially authentication, which is central to our discussion.

The purpose of this paper is to investigate further into this formalism and strengthen it. We develop a structure for these events and use them to specify a general authentication property. We then develop specifications for recentness and injectivity as sub-properties, and use them to refine authentication further. Finally, we demonstrate our use of signal events in modelling a variety of authentication specifications and show how signal events can bring out subtle differences in them, that is, to clarify their use and make explicit related theoretical issues. A preliminary version of this paper has appeared earlier [3]. This paper develops these ideas further by, firstly, making explicit related theoretical issues, particularly with regards to Roscoe's *intensional* and *extensional specifications* [4] and, secondly, clarifying the use of signal events by using them to specify selected authentication properties (including those by Lowe [5] and Gollmann [6]). We also examine some related approaches to formal specification of authentication by Bugliesi *et al* [7], and by Gordon and Jeffrey [8], [9]. Furthermore, as in our earlier work [10], we specify authentication properties for a basic version of the Kerberos protocol [11] as an example to demonstrate the use of signal events.

Our work is motivated by the desire to effectively analyse and express authentication properties in formal terms, so as to make them more precise and clear [12]. Further motivation is provided by Meadows [13], who notes the significance of the specification of requirements for formal cryptographic protocol analysis, and suggests three important characteristics for expressing such requirements: they must be first, expressive enough to specify useful security properties, secondly, unambiguous and finally, "easy to read and write" [13]. Our approach attempts to satisfy these characteristics.

The rest of this paper is organised as follows. In Section 2 we introduce the relevant CSP notation and briefly go over the trace semantics in CSP, along with the message space used for specifying protocols in Section 2.1. In Section 3 we discuss correspondence with respect to authentication. In Section 4 we present our main contribution where we formalise a structure for signal events to specify properties such as authentication, with recentness and injectivity. Section 5 demonstrates the use of signal events in modelling a variety of authentication specifications and Section 6 uses a basic version of the Kerberos protocol [11] as an example. In Section 7 we discuss some related work and we conclude the paper in Section 8.

2. CSP

In this section we introduce the relevant CSP notation and briefly go over the trace semantics in CSP. While we discuss this notation in detail relevant to our usage in this paper, we refer the reader to more in-depth treatments of CSP provided by Hoare [2], Roscoe [14], Schneider [15] and, more relevantly, by Ryan *et al* [16].

A CSP system is modelled in terms of processes and events that these processes can perform, which are essentially instances of communication, usually involving a channel and some data value. Events may be atomic in structure or may consist of distinct components.

The CSP expression $a \rightarrow P$ describes a process that initially performs event a and then behaves as process P. For example, consider a process *PRINT_ONE* that only accepts one document and prints it off, represented by events *accept* and *print*

$$PRINT\_ONE = accept \rightarrow print \rightarrow STOP$$

The above CSP expression describes a process that can perform the event *accept* and then *print*, after which it simply stops. The process *STOP* is the simplest CSP process that can be described; it has no event transitions and does not engage in any events. The process *PRINT_ONE* would simply make no further progress once it reaches *STOP*. We develop the process further to describe *PRINTER* recursively as

$$PRINTER = accept \rightarrow print \rightarrow PRINTER$$

This allows *PRINTER* to *accept* documents, *print* them off and return to the original state to *accept* more documents.

For the purpose of communication, a process may be described using compound events such as $c.m$ which describes a message m being communicated on channel c. Channels are also assigned types, so if T is the type of channel c then $\{c.t \mid t \in T\}$ describes the set of events associated with channel c. The communication of a process can be described as input or output, where $c!v \rightarrow P$ describes a process that will output value v on channel c and then behaves as P. The expression $c?x \rightarrow P(x)$ describes a process P accepting input x on channel c where the behaviour of P after the input is described as $P(x)$, determined by input x.

The *external choice* operator $\Box$ provides the option of running either of the two processes P or Q when put together as $P \Box Q$, where the choice between these two processes is determined by the environment. This operator can also be used in an indexed form where for an indexing set I, $\Box_{i \in I} P_i$ describes the option of running any one of the indexed processes P_i.

The *parallel* operator $\underset{A}{||}$ is used to force P and Q to run in parallel and synchronise on events in the set of events A. This is written as $P \underset{A}{||} Q$. P and Q may perform any events that are not in A independently, without the need for any synchronisation. A process P may be restricted on certain events in A, expressed as $P \underset{A}{||} STOP$ which prevents P from being able to perform any event in A.

The *interleaving* operator $|||$ is used to allow P and Q to run in parallel but not interact with each other at all. This is written as $P|||Q$ and is equivalent to parallel composition over the empty set. For a larger number of processes, an indexed form of the interleaving operator can be used. For a finite indexing set I and process P_i defined for each $i \in I$, $|||_{i \in I} P_i$ describes the interleaving of all processes P_i.

The trace semantics in CSP allows us to capture the sequence of events performed by a communicating process as a trace and then use the trace to model the behaviour of the process.

A trace of a process is a finite sequence of events that it may perform. A sequence of events *tr* is a trace of a process P if some execution of P performs exactly that sequence of events. This is denoted as $tr \in traces(P)$, where $traces(P)$ is the set of all possible traces of P. Every process has an empty trace and therefore the set of its traces is always non-empty. Also, for every process, the set of its traces is prefix-closed, that is to say, if *tr* is a possible trace of P, then so is any prefix of *tr*. The only trace of the simple process *STOP* is an empty sequence $\langle\rangle$

$$traces(STOP) = \{\langle\rangle\}$$

For the process *PRINTER* described above we have

$$traces(PRINTER) = \{\langle\rangle, \langle accept\rangle, \langle accept, print\rangle, \langle accept, print, accept\rangle, \ldots\}$$

A concatenation of two traces tr_1 and tr_2 is written as $tr_1 \mathbin{\char`^} tr_2$, which is the sequence of events in $tr_1$ followed by the sequence of events in $tr_2$. A trace $tr$ of the form $\langle a \rangle \char`^ tr'$ expresses event a followed by the remainder of the trace tr'. A prefix tr' of tr is denoted $tr' \leqslant tr$. A non-contiguous subsequence of a trace is denoted by the symbol $\preccurlyeq$ for example $\langle a,c,e \rangle \preccurlyeq \langle a,b,c,d,e \rangle$. The length $\#tr$ of a trace is the number of elements that it contains so that for example $\#\langle a,b,d \rangle = 3$. The set of events appearing in a trace tr is denoted as $\sigma(tr)$; the operator extends to processes where $\sigma(P) = \bigcup_{tr \in traces(P)} \sigma(tr)$ is the set of all events appearing in all traces of P.

For a set of events A, the projection operation $tr \upharpoonright A$ describes the maximal subsequence of tr, all of whose events are drawn from A. Another form of projection is on the set of channel names where $tr \Downarrow C$ describes the sequence of messages passed on a set of channels C.

Trace semantics are used by Schneider [17,1] to specify security properties for protocols as *trace specifications*. This is done by defining a predicate on traces and checking whether every trace of a process satisfies the trace specification. For a process P and a predicate S, P satisfies S if $S(tr)$ holds for every trace tr of P, that is, $P \textbf{ sat } S \Leftrightarrow \forall\, tr \in traces(P) \bullet S(tr)$.

In this paper we use trace specifications to express causal precedence on events. For some events a and b, we describe a predicate 'a **precedes** b' where

$$P \textbf{ sat } a \textbf{ precedes } b \;\Leftrightarrow\; \forall\, tr \in traces(P) \bullet (tr' \char`^ \langle b \rangle \leqslant tr \Rightarrow \langle a \rangle \text{ in } tr')$$

This means that any occurrence of b in any trace tr of P is preceded by an occurrence of a in tr. We also use trace specifications to express properties in terms of the number of occurrences of events. For some events a and b, we describe a predicate 'a **is injective to** b' where

$$P \textbf{ sat } a \textbf{ is injective to } b \;\Leftrightarrow\; \forall\, tr \in traces(P) \bullet \#(tr \upharpoonright a) \geqslant \#(tr \upharpoonright b)$$

This means that the number of occurrences of a in any trace tr of P is equal to or more than the number of occurrences of b in tr.

2.1 Message space

We describe a set of all atoms that could ever appear in a message of a protocol. Consider the set of all participant identities that may take part in a protocol as $\mathcal{U}$, the set of nonces used by participants in protocol runs as $\mathcal{N}$, the set of timestamps used by participants as $\mathcal{T}$ and a set of encryption keys used as $\mathcal{K}$, which may include different types of keys such as public keys and private keys (denoted pkA and skA for a participant A). The set of all such atoms $\mathcal{A}$ is defined as

$$\mathcal{A} ::= \mathcal{U} \mid \mathcal{N} \mid \mathcal{T} \mid \mathcal{K}$$

We use $\mathcal{A}$ to define a message space $\mathcal{M}$ that contains all the messages that may ever appear during a protocol's execution

$\mathcal{M}$ ::= $\mathcal{A}$

| $\{\mathcal{M}\}_k$ encryption of message $\mathcal{M}$ with key k ($\in \mathcal{K}$)

| $\mathcal{M}.\mathcal{M}$ concatenation of messages

3. Authentication by correspondence

This section describes the notion of correspondence and its relationship with the property of authentication. We explore the use of correspondence by Woo and Lam [18] to specify authentication.

We define the notion of correspondence as, for some *A* and *B*, if a participant *A* attempts to communicate with a corresponding participant *B* then the correspondence property requires *B* to have apparently taken part in the communication. While *A* certainly attempts to communicate with *B*, *B* may have attempted to communicate with a participant other than *A*. The underlying idea is used to make explicit a participant's involvement (role) in their run of the protocol with respect to the involvement of a corresponding participant and, therefore, a basis on which the "authenticated-to-authenticator" relationship between the two participants of an authentication protocol is formally expressed.

Authentication is an important security property provided by a family of cryptographic protocols, aptly named authentication protocols. The goal of such protocols is to allow communicating parties to confirm (to varying extent) each other's identities over a public network. There are many attempts at formally defining authentication protocols and expressing their goals in various terms [5,6,19]. Over the years, correspondence has emerged as "the concept of choice" [20] for analysing and verifying such authentication goals.

If the goal of an authentication protocol is for *A* to authenticate *B*, then *B* is required to play its intended role in the protocol. While we use correspondence to formally express *A* and *B*'s involvement in a protocol run, it merely serves as a means to establish the involvement of participants in protocol runs. For the purpose of authentication, however, we need to formally express the often subtle requirements such as a participant's engagement in the correct sequence of events, i.e. message-exchanges, along with an agreement on a set of data values and/or the number of executed runs between the participants.

The use of correspondence to specify authentication was first attempted by Woo and Lam [18]. They describe correspondence in terms of the participants in an authentication protocol as "when an authenticating principal finishes its part of the protocol, the authenticated principal must have been present and participated in its part of the protocol". Woo and Lam [18] introduced the notion of *correspondence assertions* as formal instruments to express authentication, defined with respect to protocol executions. A *correspondence assertion*, using the operator '$\hookrightarrow$' which is read as "is preceded by", is expressed as

$$(B, \mathit{EndRespond}(A)) \hookrightarrow (A, \mathit{BeginInit}(B))$$

The above assertion states the requirement for a protocol with two participants *A* (initiator) and *B* (responder), where the construct (*B*, *EndRespond*(*A*)) represents a successful end of *B*'s response with *A* and (*A*, *BeginInit*(*B*)) represents *A*'s beginning of initiating a protocol run with *B*. The assertion effectively states that *B*'s successful response to *A* has to be preceded by *A*'s corresponding intent to run the protocol with *B*. These constructs can be used to specify further details of a protocol, such as any cryptographic keys or data being used or communicated, providing a very simple basis for specifying authentication goals of a protocol.

4. Signal events in CSP

Schneider [1] makes use of correspondence while formally expressing authentication in terms of trace specifications. The style of expression is very similar to that of Woo and Lam's correspondence assertions above. In this section, we introduce Schneider's [1] use of signal events and demonstrate their use to specify authentication by correspondence. We then proceed to develop the structure of these events and their use within the protocol modelling in CSP.

For an authentication protocol, Schneider [1] uses signal events of the form *Running* and *Commit* to express the progressive stages of the protocol on behalf of the participants. If two protocol participants *A* and *B* are running in parallel, where *A* is trying to authenticate *B*, the signal *Commit.A.B* indicates a stage when *A* is convinced of *B*'s participation in the run and *Running.B.A* indicates *B*'s involvement in the run with *A*. The authentication property is then expressed as whenever *Commit.A.B* appears in a trace *tr* of this system, the corresponding *Running.B.A* signal appears beforehand. We define authentication as a trace specification. We assume a network **NET** that comprises of a protocol with two participants *A* and *B* running in parallel

$$\mathbf{NET} = A \,||\, B$$

where *tr* is some trace of **NET**, $tr \in traces(\mathbf{NET})$. *A*'s authentication of *B* is then defined as follows

Definition 1.0

A authenticates B = *Running.B.A* **precedes** *Commit.A.B*

To express *B*'s authentication of *A*, we simply use *Commit.B.A* to indicate *B*'s authentication of *A* and *Running.A.B* to indicate *A*'s involvement in the run with *B*. The network **NET** is said to satisfy *A authenticates B* if all its traces satisfy the trace specification above. That is to say, **NET sat** *A authenticates B*.

4.1 Developing signal structure

A signal event is introduced within a participant's modelled protocol run; the information articulated by the signal is with respect to that participant. In terms of concurrent processes, a signal is strictly local to a process; so it cannot include any information that a participant has not observed or engaged in up to the point the signal is introduced. We divide a signal event into two distinct parts, *Event* and the *Data*, and express it as

$$Event.Data$$

where *Event* is a member of a set $EVENT = \{Running, Commit, Begin\_Run\}$. Each of these signal types is used to indicate a particular stage in a protocol run

- A *Running* signal indicates that a participant is in a state of execution with perhaps further parameters relevant to that execution defined in the *Data* part of the signal,

- a *Commit* signal indicates the participant's state of agreement regarding some particular execution, details of which are defined in the *Data* part, and

- a *Begin_Run* signal indicates the start of a protocol run.

The set *EVENT* is not limited to these three events and more events could be added if required for the purpose of analysis. While the *Event* part of the signal signifies the type of occurrence of the signal, the *Data* part simply states the information that is strictly relevant to that occurrence. We write the *Data* part as $u_1.u_2.d$ and divide it into two distinct parts. The first part $u_1.u_2$ is the indication of the identities of the participants involved in the protocol run where $u_1, u_2 \in \mathcal{U}$. The order of elements u_1 and u_2 is such that the identity of the process executing the signal is listed first, as u_1, followed by the identity of the process it is attempting to correspond with, as u_2.

The second part d includes data such as nonces or timestamps that distinguish protocol uniquely and provide recentness. We specify d as a set of data of type $\mathcal{N}$ or $\mathcal{T}$, and restricted to only those elements that the process executing the signal can generate (under the '$\vdash$' relation). For a process P executing the signal, $data(P)$ represents all such possible data for P

$$data(P) = \{m \mid m \in \mathcal{N} \cup \mathcal{T} \wedge \sigma(P) \vdash m\}$$

The set d is then written as a concatenation of elements $d_1.d_2\ldots d_n$, where $0 \leqslant n$ and for $\forall\, i$, $1 \leqslant i \leqslant n$, $d_i \in data(P)$. The structure of the entire signal can now be unfolded to $Event.u_1.u_2.d_1.d_2\ldots d_n$.

We may use variables in the data part of a signal to specify values that are used in the protocol run but are not known to the participant executing it. So, for example, we use *Running.B.u* to indicate *B*'s run with a participant whose identity *B* is not aware of. We use u to indicate a variable of type $\mathcal{U}$, n to indicate a variable of type $\mathcal{N}$ and t to indicate a variable of type $\mathcal{T}$. This convention is used throughout the thesis, unless otherwise stated.

4.2 Signals to specify recentness

Recentness is an important property often discussed in the context of authentication (and key establishment) protocols. This property is critical to cryptographic protocols as the use of cryptography serves as means of providing some form of trust between protocol participants. This trust, however, may be valid for some limited time after which it may not hold. Consider the one-step protocol shown in Figure 1.0 for example. The protocol involves no nonce challenges and allows a participant *B* to authenticate itself to *A* by signing *A*'s identity and sending it to *A*. The signal *Running.B.A* indicates *B*'s intention of running the protocol with *A* prior to the message sent to *A*. The *Commit.A.B* signal indicates *A*'s authentication of *B* after it has received the message. Figure 1.0 shows such a run of the protocol followed by an intruder replaying the message to *A* at a later stage to convince it of *B*'s authenticity. *A* is misled into authenticating *B* since the protocol provides no indication of recentness of *B*'s run. The intruder can continue to use this message several times defeating the goal of the protocol each time. Due to the very nature of such a trust between protocol participants (subject to deception and manipulation by an intruder in an open environment), it needs to be inextricably tied with some notion of time in the sense of being old or recent. In this section, we use signal events to specify recentness and express recentness in the context of authentication.

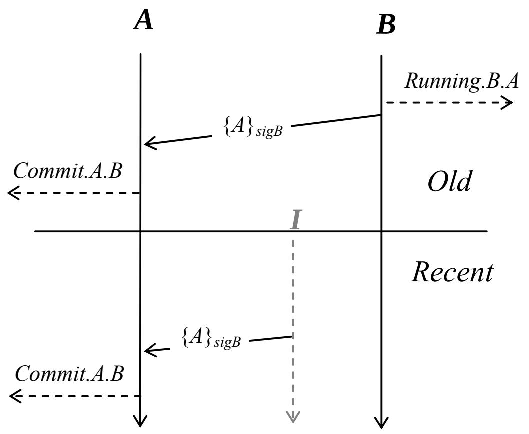

Figure 1.0 The problem of recentness

For a protocol to satisfy the property of recentness, we assume that it already satisfies Definition 1.0, that is to say, for every *Commit* event that occurs a corresponding *Running* event precedes it. We consider the entire protocol sequence from the perspective of the current run (a complete single run as intended by the protocol designer) and any signals that are modelled are only meaningful for this run; any *Running* or *Commit* events for example occur in this current run. We consider a run to be previous to a current run if it has started before the current run has started. It may or may not have finished before the current has started (so they may overlap).

We introduce an extra signal event *Begin_Run* to mark the start of a current run and use it differentiate between the previous and current runs of a protocol. The data part of this signal includes the names of the participants such as *A*, *B*, etc. and data values used during the run such as a nonce or a timestamp that act as indicators of recentness.

We build on the general definition of authentication in Definition 1.0 and define *A*'s recent authentication of *B* using a recentness indicator *N* in Definition 1.1 as

Definition 1.1

A recently_authenticates B = *Running.B.A.N* **precedes** *Commit.A.B.N*

∧ *Begin_Run.A.B.N* **precedes** *Running.B.A.N*

In Definition 1.1, the corresponding *Running.B.A.N* event precedes every *Commit.A.B.N* event and a corresponding *Begin_Run.A.B.N* event precedes every *Running.B.A.N* event, indicating that the *Running.B.A.N* event occurs after the start of the current run. The recentness indicator *N* is generated by the authenticating participant, in this case by *A*. In this paper, we will denote the identity of the participant generating the indicator by adding it as a subscript. So, for example, N_A and T_A denote a nonce and a timestamp generated by *A* respectively.

To demonstrate our modelling of recentness, consider the simple protocol shown in Figure 2 below. The protocol allows an initiator *A* to send a fresh nonce N_A to *B*, who responds by signing the nonce along with *A*'s identity and sending it back to *A*. *A* is assured that *B* has only responded after *A* has sent N_A out to *B*. We place a *Begin_Run.A.B.*N_A signal to indicate the start of this run by *A*. For *A*, it acts as a recentness indicator for *B*'s response, as *B* could not have possibly replied before this event.

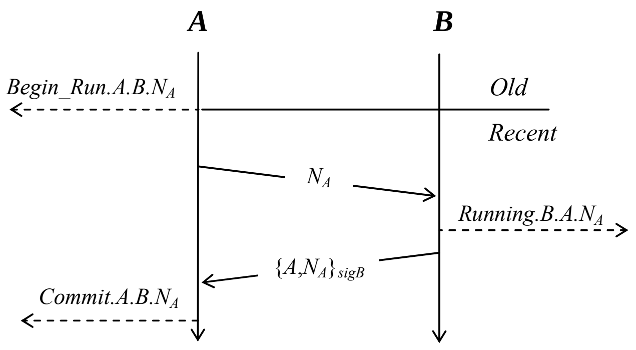

Figure 2 Modelling recentness using the *Begin_Run* event

Note that Definition 2 can also be expressed for protocols using timestamps to provide recent authentication. In this case the data part of the signal can specify a timestamp instead of a nonce; an example of such a property could be found in Section 6.3. The use of timestamps, however, do not restrict the *Commit* event to appear only once for every *Running* event prior to it. So it is possible for *A* to authenticate *B* more than once while *B* has only run the protocol once (we deal with this issue in Section 4.3). Recentness provided by using timestamps is still useful. Due to the use of a *Begin_Run* event, if *A* does authenticate *B* more than once, *A* is still assured that *B* has run the protocol recently. This models precisely the nature of authentication provided by protocols that use timestamps to serve as a measure of recentness. Such protocols are designed so that an authenticator accepts a message (containing a timestamp) within an acceptable window (length of time during which the message is deemed to be recent). If so, then it is possible that an authenticating message from the authenticatee is replayed to the authenticator several times during the acceptable window. As soon as the window expires, however, the authenticator declines any such message to be recent.

Our approach to recentness here is comparable to Lowe's earlier work [5] on specifying recentness. Our modelling of recentness is straightforward in that we focus on a single protocol run and use a signal event to mark the start of a recent run. Lowe [5], on the other hand, uses a unique signal *Begin.ds* to mark the start of every run between two participants, where *ds* indicates the involved participants such as *A* and *B*, along with an extra field for a *run identifier*. The *run identifier* (assigned α, β, etc.) is used to associate the corresponding *Begin* and *Commit* signals with every run. Lowe uses this signal to specify the runs of the involved participants. If the runs of the participants overlap, then the recentness of a participant's run is verified. This approach of overlapping runs however may not always hold true; a participant may finish its run of the protocol while the other participant may not even have started by that stage, which is possible in protocols that use trusted servers. Lowe gets around this by suggesting that a server's run may overlap both participants' run, one followed by another; there may still be limitations where more than one server is involved as Lowe highlights [5].

4.3 Signals to specify injectivity

The notion of injectivity, as described by Lowe [5], requires every run of an authenticator to correspond to a unique run of the authenticatee. This implies a one-to-one relationship between the runs of the protocol participants. This property is particularly useful for key distribution and establishment as fresh session keys are attributed to each run of the protocol.

Observe that Definitions 1.0 and 1.1 require a *Commit* event to be preceded by a *Running* event but they do not require every *Commit* event to be preceded by a unique *Running* event.

Consequently, we may have more than one *Commit* event for a single *Running* event. An authenticatee may run the protocol only once but the authenticator may authenticate that run more than once. This is true for the example protocol shown in Figure 1. The use of a nonce, however, addresses this problem and provides injectivity due to its uniqueness, as in Figure 2. To specify injectivity in trace specifications, we require each *Commit* signal to be preceded by a unique *Running* signal. The number of *Running* signals is strictly equal to or more than the number of corresponding *Commit* signals. We build further on Definition 1.0 and formalise injective authentication in Definition 1.2 as

Definition 1.2

$$A\ injectively\_authenticates\ B\ =\ Running.B.A.N_A\ \textbf{ precedes }\ Commit.A.B.N_A$$
$$\wedge\ Running.B.A.N_A\ \textbf{ is injective to }\ Commit.A.B.N_A$$

where the second clause implies a unique *Running* event preceding every *Commit* event. The property '$Running.B.A.N_A$ **is injective to** $Commit.A.B.N_A$' (as described in Section 2) ensures that every time *A* authenticates *B*, *B* has taken part in a run with *A*. The uniqueness of runs is ensured by the use of nonces (in this case N_A). Note from the definition that the converse is not true, that is to say, *B* may have attempted to run the protocol with *A* more times than *A* has successfully authenticated *B*.

4.4 Placing signals

Constructing and placing a signal within a protocol model depends on the nature of the relationship that is being analysed. Authentication protocols are built on top of challenge-response exercises, which is central to the idea of confirming a participant's identity. Most such protocols are designed so that an authenticator is allowed to issue a challenge to an authenticatee. The stage at which the authenticatee receives the challenge is marked by a *Running* signal to indicate its participation. The authenticatee is then expected to respond to the challenge as per the rules of the protocol, which may involve sending the challenge back (in the desired form) directly to the authenticator, or further interaction with a third party who relay the response to the authenticator. At the satisfactory receipt of this response, the authenticator is assured of the authenticatee's identity and the stage is marked by a *Commit* signal. The authentication property is formulated for this *Commit* event to be preceded by the corresponding *Running* event, where the data parts specify the challenge used in the run. We demonstrate this by using a challenge-response exercise in Figure 3 below.

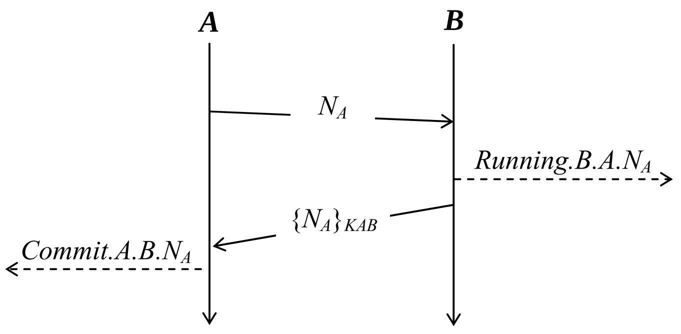

Figure 3 Placing signals in a challenge-response protocol

A *Running* signal is placed in *B*'s run after it receives the challenge N_A from *A*. The data part of the signal shows *B.A* as the participant identifiers where *B* followed by *A* indicates *B*'s involvement in a run with *A*. The data value N_A refers to the specific nonce as the challenge received. This comes together to form *Running.B.A.*N_A. A *Commit* signal is placed in *A*'s run of the protocol right after it receives the response from *B*. The participant identifiers show *A.B* indicating *A*'s satisfactory completion of the run with *B*, and the data value shows N_A as the nonce used in this run. The entire signal is formed as *Commit.A.B.*N_A.

We introduce rules that guide the placing of signals in protocols using nonce-based challenge and response

1) The *Running* signal is executed by the authenticatee and placed immediately after the nonce challenge is received, and

2) the corresponding *Commit* signal is executed by the authenticator and placed immediately after the desired response (containing the nonce) is received.

We incorporate a third rule for the *Begin_Run* event, introduced in Section 4.2, and note that it is always placed at the start of a protocol run to indicate recentness of the run. Since (the uniqueness of) nonces used in challenge-response exercises also provide recentness we state the rule

3) The *Begin_Run* signal is executed by the authenticator and placed immediately before the nonce challenge is issued.

We describe three further rules to place corresponding signals in protocols that use timestamps. Such protocols use a single message containing the timestamp to provide both authentication and recentness. The message, referred to as the authenticating message, allows the authenticator to authenticate the authenticatee

4) The *Running* signal is executed by the authenticatee and placed immediately before the authenticating message is sent,

5) the corresponding *Commit* signal is executed by the authenticator and placed immediately after the authenticating message is received, and

6) the corresponding *Begin_Run* signal is executed by the authenticatee and placed immediately before the *Running* signal is executed.

There are also other mechanisms that provide guarantees about the prior occurrence of an event during a protocol execution. If a participant receives a message encrypted with a secret key that it shares with another participant then it is guaranteed that the other participant has sent the message. A digitally signed message is also an example of such an occurrence, where the public key of a participant could be used to check whether the participant has used its private key to sign the received message. Such mechanisms, however, do not provide any guarantees about the recentness of messages, as the key could have been used at any time in the past. We describe two further rules to guide the placement of *Running* and *Commit* signals for such mechanisms. We focus on the message that is sent by the authenticatee and received by the authenticator allowing it to derive the guarantee about its sender. The message received by the authenticator may not be in the same form as sent by the authenticatee, as it may pass through a third participant and may also go through a transformation

7) The *Running* signal is executed by the authenticatee and placed immediately before the concerned message is sent, and

8) the corresponding *Commit* signal is executed by the authenticator and placed immediately after the received message which allows it to guarantee the transmission of the authenticatee's message.

We leave the data part of the two signals blank as such mechanisms do not use any recentness indicator relevant to the authentication property. Any data values agreed as a result of such mechanisms are outside the scope of signals. An example property that demonstrates Rules 7 and 8 is given in Section 6.2.

We now proceed to use our rules to place signal events in protocol models and use the Needham-Schroeder-Lowe (NSL) protocol as an example to demonstrate some of these rules. It is a good example as it provides authentication along with both recentness and injectivity to both participants. The NSL protocol originally presented by Needham and Schroeder [21], was later found to have a flaw by Lowe [22] who suggested an amendment of including *B*'s name in the second message, hence the name. A shortened version of the amended protocol is shown in Figure 4 below

(1) $\mathrm{A} \rightarrow \mathrm{B}$: $\{A{,}N_A\}_{pkB}$
(2) $\mathrm{B} \rightarrow \mathrm{A}$: $\{N_A{,}N_B{,}B\}_{pkA}$
(3) $\mathrm{A} \rightarrow \mathrm{B}$: $\{N_B\}_{pkB}$

Figure 4 Needham-Schroeder-Lowe protocol

The goal of the protocol is to authenticate *A* to *B* and *B* to *A*. *A* initiates the protocol by sending to *B* its own identity concatenated with a nonce N_A and encrypted with *B*'s public key. Once *B* receives the message, it is aware of *A* and its nonce N_A. *B* responds by generating a nonce N_B, concatenating it with N_A and its own identity, encrypting it under *A*'s public key and sending it back to *A*. Once *A* finds the nonce N_A in *B*'s response, it successfully authenticates *B*. *A* finally sends back *B*'s nonce N_B encrypted under *B*'s public key. Upon receipt of this, *B* successfully authenticates *A*. The complete execution of the protocol is shown in Figure 5 below

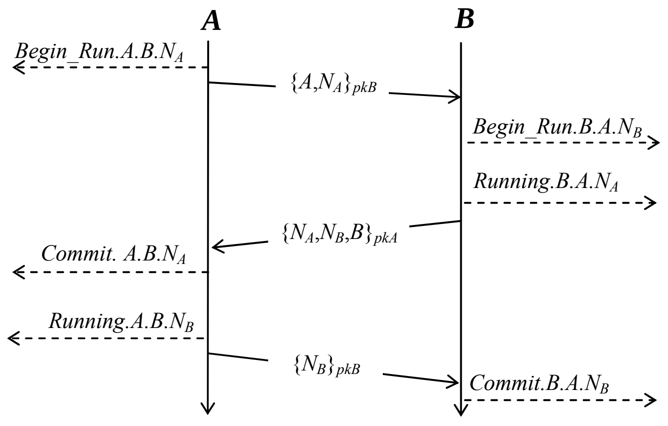

Figure 5 Complete execution of the NSL protocol

We follow Rules 1, 2 and 3 that are to do with protocols using nonces, and use six signal events each of which is explained in detail below in the order they appear. Figure 5 above illustrates the signals for a complete execution of the protocol.

Begin_Run.A.B.N_A
The signal indicates the initiation of the protocol on behalf of *A* with *B*. We use nonce N_A to identify this particular run on *A*'s behalf and also to act as a recentness indicator for *A*'s

authentication of *B*.

Begin_Run.B.A.N_B
The signal indicates the start of the response from *B* to *A*, after it receives the first message from *A*. The nonce N_B is not only mentioned as a unique identifier for this run of *B*, but also as a recentness indicator for *B*'s authentication of *A*.

Running.B.A.N_A
This signal is important for *A*'s authentication of *B*. At this stage, *B* is in possession of all the information that it needs to respond to *A*: *A*'s identity and *A*'s nonce N_A. The data part of the signal includes N_A to represent the unique run *B* is taking part in.

Commit.A.B.N_A
B's response also allows *A* to authenticate *B*: this signal indicates the successful authentication event. *A* is assured that *B* has received its nonce N_A and responded.

Running.A.B.N_B
This signal corresponds to the *Running* signal in *B*'s run. It is only after receiving *B*'s response to the first message that *A* reaches a stage where it has all the information it needs for this protocol, particularly *B*'s nonce N_B.

Note that the *Commit.A.B.*N_B signal is placed before the *Running.A.B.*N_B signal in Figure 3.4. The order of these two signals in this instance is not important. Once *A* receives the response from *B*, it both authenticates *B* to execute *Commit.A.B.*N_B and also reaches a stage where it can execute the *Running.A.B.*N_B event.

Commit.B.A.N_B
The protocol run is completed by *A* sending the last message to *B*. Once *B* receives the last message, it is assured that *A* is in possession of the nonce N_B and has agreed upon it, allowing *B* to authenticate *A*: this signal indicates the successful authentication event and corresponds to *A*'s *Commit* event.

Note that the *Commit* events mention the corresponding nonces N_A and N_B used by *A* and *B* to authenticate each other. This is significant for injectivity as we assume both participants use fresh nonces and therefore both *A* and *B* would always take part in a unique run to authenticate each other every time they do so. The protocol has two authentication goals and we will now use these signals to specify each of the goals as separate trace specifications.

Definition 1.3

$$\begin{aligned} \mathit{NSL\_A\_authenticates\_B} = \; & \mathit{Running.B.A.N_A} \;\textbf{precedes}\; \mathit{Commit.A.B.N_A} \\ & \wedge\; \mathit{Begin\_Run.A.B.N_A} \;\textbf{precedes}\; \mathit{Running.B.A.N_A} \\ & \wedge\; \mathit{Running.B.A.N_A} \;\textbf{is injective to}\; \mathit{Commit.A.B.N_A} \end{aligned}$$

Definition 1.3 specifies *A*'s authentication of *B*. The first line specifies the actual authentication of *B* to *A*, where the signals provide useful detail in terms of the data agreed by both. Note that the *Running* signal shows *B*'s awareness of the identity of the initiator *A*. The following line specifies the recentness of this authentication where *B* could have only used the nonce N_A after *A* had sent out the nonce to *B*, hence *B*'s response could not possibly have been sent out any earlier. The final line insists on the one-to-one relationship between *A* and *B*'s protocol runs such that the number of *Running* events have to be at least as many as the number of *Commit* events – the uniqueness of each of the nonces used by *A* and *B*, provide this one-to-one relationship.

Definition 1.4

$$\begin{aligned} NSL\_B\_authenticates\_A \;=\; & Running.A.B.N_B \;\textbf{precedes}\; Commit.B.A.N_B \\ & \wedge\; Begin\_Run.B.A.N_B \;\textbf{precedes}\; Running.A.B.N_B \\ & \wedge\; Running.A.B.N_B \;\textbf{is injective to}\; Commit.B.A.N_B \end{aligned}$$

Definition 1.4 specifies *B*'s authentication of *A*. The first line specifies the actual authentication of *A* to *B*, followed by the recentness and injectivity conditions. This property is entirely symmetrical to *A*'s authentication of *B*.

The NSL protocol is a good example where authentication is achieved in a strong form, that is, it provides recentness and injectivity to both participants. Note that the use of a fresh nonce by each participant to provide these properties inherently provides agreement on data (nonces) as well, making this protocol feasible for key derivation.

In terms of authentication goals, observe how signals demonstrate the progressive nature of such a protocol. Every step of a protocol is crucial as it allows participants to develop trust in each other. This also demonstrates the vulnerability of protocol design to manipulation and deception by an attacker on a network.

4.5 Intensional and extensional specifications

Observe that describing the use of signal events occurring in a specific order as a property, requires protocols not only to execute and arrive at a final authentication goal but also behave in a specific causal order. It is possible for a protocol to satisfy a goal set out by a certain specification, but may not necessarily run as per the specification. Roscoe [4] clarifies this distinction by introducing the notion of *intensional* and *extensional specifications* and highlights "*extensional specifications*, which seek to establish what is achieved by the protocol without making detailed analyses of the actual communications nodes have sent, and *intensional specifications* which concentrate on checking that the designer's expectations about these communications are justified" [4].

The CSP trace specifications (in Definitions 1.0 and so forth) combine both aspects of *intensional* and *extensional specifications*. Whereas the use of a *Commit* event in a trace specification allows an *extensional* goal to be expressed, the ordering of two or more signal events is essentially *intensional* nature. We demonstrate this by examining Lowe's [5] *weak agreement* property as an example.

Lowe [5] introduces *weak agreement* and is defined for a protocol that "guarantees to an initiator *A* weak agreement with another agent *B* if, whenever *A* (acting as initiator) completes a run of the protocol, apparently with responder *B*, then *B* has previously been running the protocol, apparently with *A*. *B* may not necessarily have been acting as responder". An example protocol with just such a property (and no more) is described in Figure 6.

(1)	A → B	:	$\{A,B\}_{skA}$
(2)	B → A	:	$\{B,A\}_{skB}$

Figure 6 An example weak agreement protocol

A sends out a message with its own identity concatenated with *B*'s identity who it wishes to authenticate, to *B*. The message is signed by *A*'s private key. At this stage *B* is assured that *A* wishes to talk to *B*. *B* responds with sending its own identity, concatenated with *A*'s identity in a similar fashion. Upon receipt of a response from *B*, *A* is assured that it is talking to *B* and also that *B* is aware that it is talking to *A*. Hence *A* is in *weak agreement* with *B*.

When *B* receives the first message from *A*, it is assured that *A* indeed intended to talk to *B*. When *B* replies to *A*, *A* checks for the identities of *B* and *A*, and verifies *B*'s digital signature. Once the response is verified, *A* is assured that *B* is aware of *A*'s presence, indicated by *Running.B.A* in *B*'s run, and forms *weak agreement* with *B*, indicated by *Commit.A.B* in *A*'s run. We formalise this in Definition 1.5 and illustrate a run of the protocol in Figure 7 below

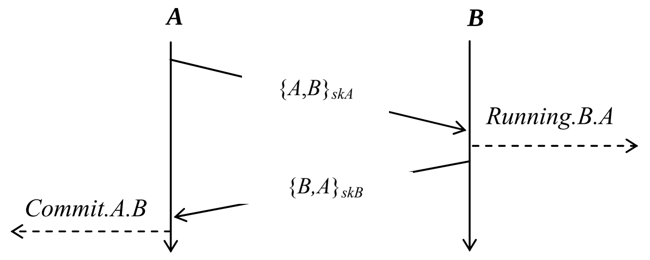

Figure 7 Complete execution of the example weak agreement protocol

Definition 1.5

A in_weak_agreement_with B = *Running.B.A* **precedes** *Commit.A.B*

Note that Definition 1.5 provides the same authentication guarantee as our original property of authentication in Definition 1.0. Both *A* and *B* authenticate each other, and both are aware that they are being authenticated as such. Both do not require any recentness indicators, nor do they require any agreement on protocol runs.

The example protocol in Figure 7 provides no guarantees on the recentness of either of the messages. This allows an intruder to replay either of the old messages to *A* or *B* thereby misleading them to commit to runs with each other, or a situation where *A* and *B* may both initiate a run with each other allowing *A* to believe that it has successfully authenticated *B*, an example of such a run is shown in Figure 8 below

(1.1)	A → B	:	$\{A,B\}_{skA}$
(2.1)	B → A	:	$\{B,A\}_{skB}$

Figure 8 A possible run of the example weak agreement protocol

*B'*s initiating attempt to *A* is identical to *B*'s response to *A*'s initiating attempt. *A* mistakes *B*'s initiating attempt as the response to *A*'s initiating attempt earlier, allowing it to commit to a run with *B*. This allows *A* to *Commit.A.B* without a preceding *Running.B.A* on *B*'s behalf and therefore means that the protocol does not satisfy Definition 1.5. Observe how Definition 1.5 encapsulates both *intensional* and *extensional specifications*. In terms of signals, the possible run shown in Figure 8 implies that for every *Commit.A.B* there may not be a *Running.B.A* event. This would satisfy the *extensional specification* of *weak agreement* but not the *intensional specification*, which requires a corresponding *Running.B.A* to precede every *Commit.A.B* event; the possible run shown above does not satisfy Definition 1.5.

Gollmann [19] notes that such a violation does not necessarily constitute an attack on a protocol. This is true in this case as *weak agreement* allows this by definition, that is, "*B* may not necessarily have been acting as responder". This means that the violation of the *intensional specification* does not lead to the violation of *extensional specification* for this protocol.

5 Authentication Definitions

Over the years there have been numerous attempts at defining and expounding authentication, particularly in the context of designing authentication protocols. It is interesting that many of these definitions have added to the ambiguity of the notion rather than clarifying it, as Gollmann [12] discusses. Our effort intends not to add to these definitions but present a notation that can capture the precise terms of these definitions and highlight subtle differences between them. In this section, we examine some selected definitions of authentication that have appeared in the literature and specify them using signal events. We highlight any differences between them and discuss relevant modelling issues.

5.1 Lowe's Aliveness

Lowe [5] presents a range of authentication specifications with varying degrees of strictness. We examine the weakest of these definitions called *aliveness*. Lowe [5] defines *aliveness* for a protocol that "guarantees to an initiator *A aliveness* of another agent *B* if, whenever *A* acting as initiator completes a run of the protocol, apparently with responder *B*, then *B* has previously been running the protocol. *B* may not necessarily have believed that he was running the protocol with *A*. Also, *B* may not have been running the protocol recently". We formalise *aliveness* in Definition 1.6.

Definition 1.6

A confirms_aliveness_of B = *Running.B.u* **precedes** *Commit.A.B*

Observe that *aliveness* in this form only guarantees that agent *B* was once alive. Since there is no nonce or timestamp defined for recentness, it is possible that *A* may authenticate *B* for *aliveness* from an old run. This definition is not very strict and does not even require *B* to be aware of *A*. We compare this with a similar definition by Gollmann and highlight some interesting differences.

5.2 Gollmann's G3

Gollmann [6] presents four related authentication goals, one of which is the *authenticated recipient* goal, *G3*. Assuming *A* and *B* are participants in a protocol, *G3* states "A cryptographic key associated with *B* was used during the protocol run. It is not necessary for *A* to receive a message where this key had been used. The protocol run is defined by *A*'s challenge or a current timestamp".

Compared with Lowe's *aliveness*, *G3* provides an assurance of *B*'s involvement in the protocol run but in a different way. The *aliveness* property requires *B* to be simply running the protocol while *G3* specifically states the use of a cryptographic key associated with *B*, hence *B*'s involvement in the run. *G3* also requires that the key is used "during" the protocol run, requiring a *recent* involvement on behalf of *B*, as opposed to *aliveness*, an observation also made by Lowe [5]. Both properties, however, do not require *B* to be involved in a run necessarily with *A*, which makes them similar.

Consider K_B to represent a key associated with *B* and a challenge C_A, where $C_A \in \mathcal{N} \cup \mathcal{T}$, to represent *A*'s challenge. The $Running.B.u.C_A$ signal would then indicate *B* is running the protocol run, uniquely defined by C_A. Since C_A represents *A*'s challenge, we include it in the data part of the $Commit.A.B.C_A$ signal. We define *G3* in Definition 1.7 as

Definition 1.7

$$A\ confirms\_G3\_with\ B \ = \ Running.B.u.C_A \ \textbf{precedes} \ Commit.A.B.C_A$$
$$\wedge \ Begin\_Run.A.B.C_A \ \textbf{precedes} \ Running.B.u.C_A$$

Interestingly, the *G3* definition is not explicit about precisely what assurances *A* derives from this property such as whether *A* is assured of *B*'s use of the key K_B.

5.3 Dead Peer Detection in IKEv2

More recently Gollmann [20] has compared Lowe's *aliveness* with a high-level definition of entity authentication from the international standard ISO/IEC 9798-1 [23] which states "entity authentication mechanisms allow the verification, of an entity's claimed identity, by another identity. The authenticity of the entity can be ascertained only for the instance of the authentication exchange."

Gollmann [20] observes that the ISO definition is weaker than *aliveness*, as it provides no hints of the entities being connected or even wanting to establish a connection and only provides a guarantee that the "authenticated entity was alive during the protocol run" whereas *aliveness* actually requires the responder to run the protocol – a stricter requirement for the responder. The ISO definition, as opposed to *aliveness*, insists on recentness – confined to "the instance of the authentication exchange" – of the authenticity.

Gollmann [20] nevertheless finds the ISO definition useful as it serve a similar purpose to the *dead peer detection* (DPD) scheme Huang *et al* [24] of the IKEv2 protocol [25]. The IKE protocol allows two communicating parties to establish Security Associations (SAs) for subsequent secure communication and thus become IKE *peers* for the duration of the session. In a situation where the connectivity between the two established peers goes down (for example due to one of the hosts rebooting) and/or there is no communication between the two, it becomes imperative for the parties to detect a dead peer so as to perform failover. We consider one of the proposals that deal with this problem, known as *heartbeat* Huang *et al* [24], and examine it further. Consider *A* and *B* to be some IKE peers both of which desire to confirm each other's online status. The *heartbeat* scheme requires both peers to send periodic `HELLO` messages to each other as shown in Figure 9 below

10 seconds → A's timer fires

A → B : `HELLO(Seq_Num)`

10 seconds → B's timer fires

B → A : `HELLO(Seq_Num)`

Figure 9 The *heartbeat* scheme

The scheme requires both peers to maintain a timer. When *A*'s timer fires (after an agreed interval of, say, 10 seconds) it sends a `HELLO` to *B* and also expects a reciprocal `HELLO` from *B*. Once *B* receives *A*'s `HELLO` – sent as an authenticated payload – it is assured that *A* is not dead and waits a similar interval before it sends a `HELLO` to *A*; assuring *A* that *B* is not dead. The `HELLO` message from *A* contains a sequence number which *B* uses in the response `HELLO` message to *A*, preventing replay of old responses from *B*. In case of *B* failing to respond, *A* retransmits the message after the interval and, after a certain number of retries, *A* assumes *B* is dead and deletes any SAs formed with the peer. Both peers carry out this routine periodically irrespective of whether any peer is interested in their online status.

Note that the purpose of the scheme is not to authenticate the participants but to provide them with an assurance that their IKE peer is not dead. This certainly brings it closer to the ISO definition and justifies why Gollmann [20] sets it apart from Lowe's *aliveness*. One may maintain, however, that the periodic nature of the DPD scheme – the exchange of `HELLO`s – locks *A* and *B* in a sequence similar to any other *protocol run* and would therefore imply that they are *running* a protocol. We specify the goal of the DPD scheme in terms of signal events in Figure 10 and formalise it in Definition 1.8.

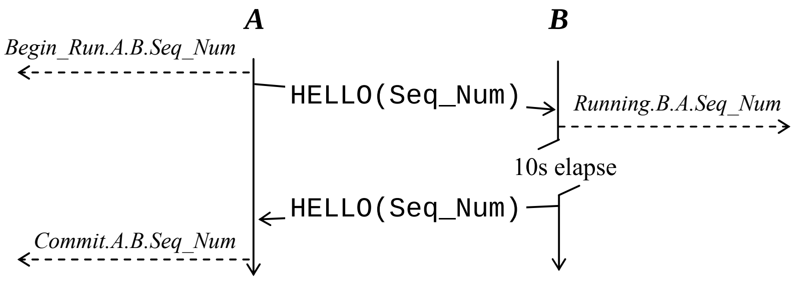

Figure 10 Complete execution of *heartbeat* scheme

Definition 1.8

A dead_peer_detects B =

$$Running.B.A.Seq\_Num \ \textbf{precedes} \ Commit.A.B.Seq\_Num$$

$$\wedge \ Begin\_Run.A.B.Seq\_Num \ \textbf{precedes} \ Running.B.A.Seq\_Num$$

We use *Begin_Run.A.B.Seq_Num* at the start of this run, where sequence number *Seq_Num* identifies this unique run and provides recentness to *A*. We use the signal event *Running.B.A.Seq_Num* to indicate that *B* has responded to *A*'s `HELLO` with the same sequence number and use the *Commit.A.B.Seq_Num* event to indicate the assurance that *A* receives regarding *B*. Note that for the purpose of placing signals in the above definition, we treat all sequence numbers as nonces with the only exception that they are predictable.

6 Kerberos Authentication protocol

The Kerberos system provides a distributed authentication service developed as part of a project Athena [26] at MIT. The system is based on a timestamp version of the Needham-Schroeder protocol [21] and the current version, known as Version 5 [27], of the protocol is part of the Microsoft Windows 2000 network authentication service. For the sake of simplicity, we use a basic version of the Kerberos protocol [11] which essentially provides entity authentication along with key transport using symmetric encryption.

(1)	A → S	:	A,B,N_A
(2)	S → A	:	$ticket_B,\{k_{AB},N_A,L,B\}_{KAS}$
(3)	A → B	:	$ticket_B,\{A,T_A\}_{kAB}$
(3*)	A → B	:	$ticket_B,\{A,T_A'\}_{kAB}$

where $ticket_B = \{k_{AB},A,L\}_{KBS}$

Figure 11 The basic Kerberos protocol

The protocol, shown in Figure 11 above, involves a client, *A*, a network service, *B* and a trusted authentication server, *S*, which shares a long-term secret key with all clients and services. The purpose of the protocol is two-fold: first, to provide recent authentication of *A* to *B* and secondly, to allow *S* to establish a session key k_{AB} between *A* and *B* with a lifetime *L*; *S* is also authenticated to both *A* and *B* in the process. The protocol step marked 3* shows *A*'s repeated authentication to *B* and establishment of key k_{AB} between the two, as long as the lifetime *L* is valid and the timestamp T_A is recent. We do not consider here an optional fourth protocol step that allows *B* to be authenticated to *A* [11]. The complete execution therefore of the protocol is shown in Figure 12 below

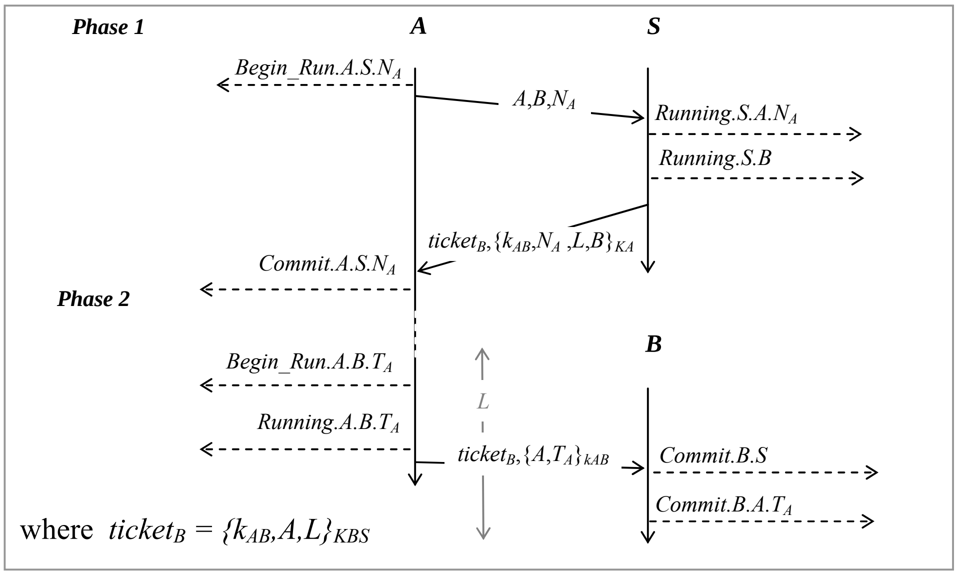

Figure 12 Complete execution of the basic Kerberos protocol

6.1 *A*'s authentication of *S*

The Kerberos protocol, as shown in Figure 12, is divided in two phases. In the first phase, *A* initiates the protocol and sends a request to *S* to establish a session key with *B* along with a nonce N_A. *S* responds by generating a fresh session key k_{AB}, concatenated with the nonce N_A, the lifetime *L* of key k_{AB} and the identity of *B*. It encrypts all this with K_{AS} and sends it to *A*. This message provides *A* with the key k_{AB} and also ensures *A* of *S*'s recent involvement in the protocol by the presence of N_A. It also provides A with $ticket_B$ that allows S and A to be authenticated to B and establish the key k_{AB} with B. We formalise A's authentication of S in Definition 1.9.

Definition 1.9

$$\begin{aligned} Kerberos\_A\_authenticates\_S = {} & Running.S.A.N_A \ \textbf{precedes} \ Commit.A.S.N_A \\ & \wedge \ Begin\_Run.A.S.N_A \ \textbf{precedes} \ Running.S.A.N_A \\ & \wedge \ Running.S.A.N_A \ \textbf{is injective to} \ Commit.A.S.N_A \end{aligned}$$

To indicate A's authentication of S, we follow Rule 2 (from Section 4.4) and use a $Commit.A.S.N_A$ event. The data part of the event specifies A's runs with S, along with N_A. We follow Rule 1 (from Section 4.4) and use a $Running.S.A.N_A$ event to show the corresponding event for S. This signal indicates S's involvement with A using the nonce N_A. We also use a $Begin\_Run.A.S.N_A$ signal as per Rule 3 (from Section 4.4) to mark the beginning of this run by A, using the nonce N_A. The nonce N_A is important as it allows A to judge the recentness of S's response with respect to this nonce; the use of N_A in this case provides A with recent authentication of S. Moreover, the final condition of Definition 1.9 specifies injectivity between S's and A's run. The use of a unique nonce in every run by A allows this phase to provide injectivity between A and S.

6.2 *B*'s authentication of *S*

The use of $ticket_B$ in the second phase of the protocol is important as it provides B with the key k_{AB} to be used between A and B. We use Rules 7 and 8 (from Section 4.4) to place appropriate signals to specify B's authentication of S.

B is assured of S's participation in the protocol every time it receives $ticket_B$ from A. Although S does not engage with B directly, B is assured that the ticket is issued by S as it is encrypted with K_{BS} and also of the validity of k_{AB} by the lifetime L in $ticket_B$. We indicate S's participation in the run by using Rule 7 and place a $Running.S.B$ signal just before S distributes $ticket_B$ to A in Phase 1. The data part of the signal is empty as there is no agreement on recentness between S and B; the signal merely indicates that S took part in the run and issued a ticket for B.

To show B's authentication of S we use Rule 8 and use a $Commit.B.S$ signal. The signal appears every time B receives A's message in Phase 2. We formalise this property in Definition 1.10.

Definition 1.10

$$Kerberos\_B\_authenticates\_S = Running.S.B \ \textbf{precedes} \ Commit.B.S$$

Observe that Definition 1.10 does not allow B to authenticate a recent run of the server S, as $ticket_B$ provides no such guarantees to B. Indeed S could generate $ticket_B$ well in advance of the time that the key k_{AB} could be used. So although B is guaranteed that $ticket_B$ $(=\{k_{AB},A,L\}_{KBS})$ comes from S, it does not authenticate S recently, neither does it have an injective relationship with S, as B could accept $ticket_B$ $(=\{k_{AB},A,L\}_{KBS})$ many times from A, only subject to the validity of k_{AB} as defined by lifetime L.

Compared to A's authentication of S, therefore, B's authentication of S is relatively weaker, as Definitions 1.9 and 1.10 show. On a technical note, the use of signal events and the corresponding Rules 7 and 8 prove to be effective here in refining (and distinguishing between) the degrees of authentication.

6.3 *B*'s repeated authentication of *A*

Along with $ticket_B$, A sends a message, known as the *authenticator*, encrypted with k_{AB}, which contains A's identity along with a fresh timestamp, T_A. The $ticket_B$ allows A to authenticate itself repeatedly to B as long as the lifetime L of k_{AB} is valid. A simply regenerates a fresh *authenticator*, and sends it to B along with $ticket_B$ (indicated by 3* in Figure 11). The use of the *authenticator* therefore provides recent authentication of A to B as it includes a fresh timestamp T_A every time A sends it to B; in case of replay of an authenticator, B would reject it. This feature provides an injective relationship between A and B; every time B accepts an *authenticator*, it is assured of A's recent and distinct involvement. Moreover, A combines a fresh *authenticator* together with a $ticket_B$ ($=\{k_{AB},A,L\}_{KBS}$) to send to B, providing B with *authenticated key establishment*, as it is assured of A's possession (and therefore validity) of k_{AB} as well.

To specify the property between A and B, we use three different signals as per Rules 4, 5 and 6 (from Section 4.4). We use the $Commit.B.A.T_A$ signal to indicate B's authentication of A. The use of timestamp T_A by A adds to this to provide recentness to B. We use a $Begin\_Run.A.B.T_A$ signal as per Rule 6 to mark the beginning of this run, where T_A serves as the measure of recentness for B.

To indicate A's involvement in this phase of the protocol, we use a $Running.A.B.T_A$ signal which specifies A's intention to run with B and its use of the timestamp T_A marking in time the precise moment of A's participation. We formalise B's authentication of A in Definition 1.11.

Definition 1.11

$$\begin{aligned} Kerberos\_A\_authenticates\_B = \; & Begin\_Run.A.B.T_A \textbf{ precedes } Running.A.B.T_A \\ & \wedge\ Running.A.B.T_A \textbf{ precedes } Commit.B.A.T_A \\ & \wedge\ Running.A.B.T_A \textbf{ is injective to } Commit.B.A.T_A \end{aligned}$$

Observe that our specification of recentness in this case is different to the one in Definition 3.9. Due to the use of timestamp T_A, B's judgement of A's recentness is not relative (with respect to a nonce), but rather in absolute terms of clock time. This allows B to be sure of both A's participation and the recentness of this instance of participation. This fine peculiarity is clearly distinguished in our specifications.

7 Related work

Our use of signal events in CSP trace specifications to specify authentication is not the first. Earlier work by Lowe presents a hierarchy of authentication properties using signal events in such specifications [5]. Lowe's work [5] is important in relating different properties to each other, particularly in terms of trace refinements. Our main contribution is to rigorously define the actual structure of a signal event. We specify a *Event* and *Data* part for a signal and emphasise on the relationship between these values and the authentication property being modelled. We hope such an effort would allow an expressive yet consistent framework to specify authentication properties for protocols.

In hindsight, the use of signals within CSP to express authentication is not very different from the use of *correspondence assertions* in the Woo-Lam [18] model introduced in Section 3. Both *correspondence assertions* and the CSP signals represent similar notions of rendering the state of protocol involvement from a participant's point of view. These notions are brought closer as both models use them to express authentication. A good example is the type-checking approach by Gordon and Jeffrey [9], who use Woo and Lam's *correspondence assertions* to specify authentication properties for cryptographic protocols [8]. The assertions are introduced in protocol runs which are modelled using spi-calculus [28]. They are modelled as events labelled with messages, where a *begin L* is a *begin-event* labelled with message *L* representing the beginning of a *correspondence*, while an *end L* is an *end-event* labelled with message *L* representing the end of a *correspondence*. The labels are used to specify the precise terms of *correspondence*, that is, the identity of participants involved and the use of data such as nonces or keys. A type-checking approach [9] is then used to identify the types of messages that are used in the protocol; as a way to prove whether a protocol satisfies the *correspondence assertions*. Protocol participants are modelled as processes in spi-calculus, where the transmission and reception of messages is modelled as events that the process may take part in. Similar to CSP, a sequence of events which a process may perform is captured by a *trace*. A *trace* of a process *P* is defined using a trace-labelled transition system, $P \xrightarrow{s} P'$, meaning *P* performs trace *s* and becomes *P'*. The *correspondence assertions* are used within these *traces* of a process to characterise it. A trace of process(es) is defined as a *correspondence* if its beginnings dominate its endings, that is, for each *end-event* labelled *L*, there is a corresponding *begin-event* labelled *L*.

Gordon and Jeffrey introduce their notion of *safety* such that a process is considered *safe* if for every *end-event* there is a distinct corresponding *begin-event*. Gordon and Jeffrey [9] go further to present a *safety theorem* to prove authentication for protocols, such that, a process *P* is *safe* if and only if for all traces *s* and processes *P'*, if $P \xrightarrow{s} P'$ then *s* is a *correspondence*. A process *P* is described as *robustly safe* if and only if for all intruder processes *O*, $P \mid O$ is safe; $P \mid O$ implies *P* and *O* running in parallel where *O* represents the intruder process that cannot assert any events and follows no typing rules. If the processes involved in a protocol are *robustly safe*, then the protocol is deemed correct.

While the labelled events are used to specify the terms of *correspondence* (similar to the data part in signal events), it is interesting to observe that Gordon and Jeffrey's notion of *safety* entails a corresponding *begin-event* for every *end-event*. This provides a one-to-one relationship between the runs of the protocol participants, which makes their notion of *safety* rather stricter than our Definition 1.0; Gordon and Jeffrey [29] distinguish between *injective* and *non-injective* correspondences by using type-checking.

More recently, Bugliesi *et al* [7] adopt a similar approach of using *correspondence* to specify authentication for protocols. Their approach uses a trace-based model, similar to that of CSP, and introduces *marker* actions described in terms of protocol participants: the marker *run*(*A*,*B*), similar to *signal Running.A.B*, indicates the start of a run by *A* to authenticate itself to *B* whereas the *marker commit*(*B*,*A*), similar to signal *Commit.B.A*, indicates the completion of an authentication session where *B* authenticates *A*. Bugliesi *et al* [7] then proceed to present their notion of *safety* that effectively uses *correspondence* to express authentication in formal terms. Their notion of *safety* is similar to that of Gordon and Jeffrey [9] in terms of checking for *correspondence* and insisting on *injectivity*. A trace is deemed to be *safe* if for every *commit*(*B*,*A*) marker there is a corresponding *run*(*A*,*B*) marker that precedes it. It is interesting to note that, unlike Schneider's signal events, these marker actions do not assert any nonces used in protocol runs; nonces are expressed separately.

8. Conclusion

Our main contribution is the development of a precise and expressive notation that is capable of embracing variations of authentication and makes our understanding of the property clearer, while satisfying the three characteristics set out by Meadows [13] for such an approach (see Section 1). We have developed the structure of the signal events in the trace semantics of CSP so they are flexible to accommodate finer notions of authentication in protocols. We have given rules to guide the precise placement of signals in protocol specifications. These are important in determining what signals are to be placed where (in relation to the challenge and response). We also demonstrate the usefulness of signal events as instruments to express the different flavours of authentication in precise terms, including recentness and injectivity. It is interesting to see how the expressive notation brings to the surface the subtle variations that exist between related concepts of authentication. We have attempted to highlight some of these diversities using signal events by various examples and comparing with other definitions of authentication. This work is important in helping protocol designers to precisely model authentication and specify the subtle variations that exists within the concept.

References

[1] Schneider, S.A. (1998) Verifying Authentication Protocols in CSP. *IEEE Transactions on Software Engineering*, 24 (9), pp.741-758

[2] Hoare, C. A. R. (1985) *Communicating Sequential Processes*, Prentice-Hall International

[3] Shaikh, S.A., Bush, V.J. and Schneider, S.A. (2005) Specifying authentication using signals events in CSP. *In*: D. Feng, D. Lin, and M. Yung, (eds.) *Conference on Information Security and Cryptology (CISC'2005), Beijing, China, volume 3822 of LNCS*, Berlin: Springer-Verlag, pp.63-74

[4] Roscoe, A.W. (1996) Intensional specification of security protocols. *Proceedings of the Ninth IEEE Computer Security Foundations Workshop*, Washington: IEEE Computer Society Press, pp.28-38

[5] Lowe, G. (1997c) A hierarchy of authentication specifications. *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, Washington: IEEE Computer Society Press, pp.31-44

[6] Gollmann, D. (1996) What do we mean by entity authentication? *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington: IEEE Computer Society Press, 1996, pp.46-54

[7] Bugliesi, M., Focardi, R. and Maffei, M. (2003) Principles for Entity Authentication. *In*: M. Broy and A.V. Zamulin (eds.) *Perspectives of System Informatics*, volume 2890 of LNCS, Berlin: Springer-Verlag, pp.294-306

[8] Gordon, A.D. and Jeffrey, A. (2001a) Typing correspondence assertions for communication protocols. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 45, pp.1-22

[9] Gordon, A.D. and Jeffrey, A. (2001b) Authenticity by typing for security protocols. *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, Washington: IEEE Computer Society Press, pp.145-159

[10] Shaikh, S.A., Bush, V.J. and Schneider, S.A. (2005) Kerberos – specifying authenticity properties using signal events. *Proceedings of Indonesia Cryptology and Information Security Conference 2005, (INA-CISC2005), Jakarta, Indonesia*, Jakarta: INFORMATIKA Press, pp.87-93

[11] Menezes, A.J., van Oorschot, P.C. and Vanstone, S.A. (1997) *Handbook of Applied Cryptography*. Boca Raton: CRC Press

[12] Gollmann, D. (2002) Analysing Security Protocols. *In*: A.E. Abdallah, P. Ryan and S.A. Schneider (eds.) *BCS-FACS Formal Aspects of Security, volume 2629 of LNCS*, Berlin: Springer-Verlag, pp.71-80

[13] Meadows, C. (2004) Ordering from Satan's menu: a survey of requirements specification for formal analysis of cryptographic protocols. *Science of Computer Programming*, 50 (1-3), pp.3-22

[14] Roscoe, A.W. (1997) *The Theory and Practice of Concurrency*. Prentice-Hall International

[15] Schneider, S.A. (1999) *Concurrent and Real-time Systems: the CSP Approach*. London: Addison-Wesley

[16] Ryan, P., Schneider, S.A., Goldsmith, M., Lowe, G. and Roscoe, A.W. (2001) *Modelling and Analysis of Security Protocols*. London: Addison-Wesley

[17] Schneider, S.A. (1996) Security Properties and CSP. *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington: IEEE Computer Society Press, pp.174-189

[18] Woo, T.Y.C. and Lam, S.S. (1993) A semantic model for Authentication Protocols. *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, Washington: IEEE Computer Society Press, pp.178-194

[19] Gollmann, D. (2000) On the Verification of Cryptographic Protocols - A Tale of Two Committees. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 32, pp.42-58

[20] Gollmann, D. (2003) Authentication by Correspondence. *IEEE Journal on Selected Areas in Communication, Special Issue on Formal Methods for Security*, 21 (1), pp.88-95

[21] Needham, R. and Schroeder, M. (1978) Using encryption for Authentication in Large Networks. *Communications of the ACM*, 21 (12), pp.993-999

[22] Lowe, G. (1995) An attack on the Needham-Schroeder public key protocol. *Information Processing Letters*, 56 (3), pp.131-133

[23] ISO/IEC - International Organisation for Standards (1991) *Information technology – Security techniques – Entity Authentication Mechanisms; Part 1: General Model*. ISO/IEC 9798-1, Second Edition, September 1991. International Standard

[24] Huang, G., Beaulieu, S. and Rochefort, D. (2004) A Traffic-Based Method of Detecting Dead Internet Key Exchange (IKE) Peers [online], *Internet Engineering Task Force (IETF)*, Available from: http://www.ietf.org/rfc/rfc3706.txt [Accessed on 21/06/2006]

[25] Kaufman, C. (2005) Internet Key Exchange (IKEv2) Protocol [online], *Internet Engineering Task Force (IETF)*, Available from: http://www.ietf.org/rfc/rfc4306.txt [Accessed on 19/06/2006]

[26] Miller, S.P., Neuman, C., Schiller, J.I. and Saltzer, J.H. (1987) *Kerberos authentication and authorization system*. Project Athena Technical Plan Section E.2.1, MIT Project Athena, Cambridge, Massachusetts, USA

[27] Neuman, B.C. and Ts'o, T. (1994) Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9), pp.33-38

[28] Abadi, M. and Gordon, A.D. (1999) A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148 (1), pp.1-70

[29] Gordon, A.D. and Jeffrey, A. (2002) Typing one-to-one and one-to-many correspondences in security protocols. *In*: *Software Security - Theories and Systems: Mext-NSF-JSPS International Symposium, ISSS 2002 Tokyo, Japan, November 8-10, 2002 Revised Papers, volume 2609 of LNCS*, Berlin: Springer-Verlag, pp.263-282