A Logical Language for Specifying Cryptographic Protocol Requirements

Paul Syverson and Catherine Meadows
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375

Abstract

In this paper we present a formal language for specifying and reasoning about cryptographic protocol requirements. We give examples of simple sets of requirements in that language. We look at two versions of a protocol that might meet those requirements and show how to specify them in the language of the NRL Protocol Analyzer. [Mea91] [Mea92] We also show how to map one of our sets of formal requirements to the language of the NRL Protocol Analyzer and use the Analyzer to show that one version of the protocol meets those requirements. In other words, we use the Analyzer as a model checker to assess the validity of the formulae that make up the requirements.

Introduction

The past few years have seen a proliferation of formal techniques for the specification and analysis of cryptographic protocols. That these techniques can be useful has been shown by the fact that several (including BAN logic [BAN89], the NRL Protocol Analyzer [Mea91] [Mea92], and the Stubblebine-Gligor model [SG92]) have been used to find flaws in open literature protocols that were previously believed to have been secure. Thus the use of formal methods for the analysis of cryptographic protocols has begun to attract attention as a promising way of guaranteeing their correctness.

Less attention, however, has been paid to the question of what exactly constitutes the correctness of a cryptographic protocol. Yet, we see that what constitutes correctness can vary widely with the application. In a key distribution protocol guarantee of secrecy and guarantee against replay attacks and impersonation are of the most importance. For a protocol used to guarantee the security of banking deposits, secrecy may or may not be important, although guarantee against replay attacks and impersonation definitely will be. Guarantee of timeliness may also be important, as well as the guarantee that messages are processed in the order that they are sent. (For example, a malicious intruder could cause somebody to overdraw his account by causing a deposit message and a withdrawal message to processed out of order.) For a protocol used to distribute rights by proxy, not only is it necessary to guarantee against impersonation, but also to guarantee the entire pedigree of a message.

Protocols may also differ in the amount of trust that is placed in each individual. For example, Burrows, Abadi, and Needham, in their logic of authentication, make the assumption that the parties trying to authenticate each other are honest and will follow the rules of the protocol.[1] For other protocols, this may not necessarily be the case. In the Burns-Mitchell resource sharing protocol [BM90], it is assumed that the party attempting to obtain the resource may be trying to cheat the resource supplier into giving him a resource that he has not paid for at the same time he is trying to guarantee the the resource supplier is not cheating him. In a voting protocol, we make the assumption that individuals may try to find out other individuals' votes, that they may try to cast their votes more than once, and that they may be willing to divulge their votes to a small group of individuals if this will help them subvert the goals of the protocol.

Even when we restrict ourselves to the analysis of key distribution protocols, it is not always clear what constitutes the appropriate requirements. For example, in [BAN90], Burrows, Abadi, and Needham describe the various orders of belief that a protocol can achieve, but make no recommendations. For example, a protocol may achieve first order belief, in which A believes that K is a good key for communication with B, and vice versa, but neither has any belief about the beliefs of the other, or it may achieve second order belief, in which not only does each believe in the key, but each believes the other believes in the key, or it may achieve some yet higher order of belief. In [Syv91] Syverson discusses the various orders of belief and where each would be appropriate.

Confusion and vagueness about requirements and assumptions has also contributed to much of the controversy about the various techniques. For example, in [Nes90], Nessett points out an alleged flaw in the Burrows-Abadi-Needham logic by using it to prove that a protocol in which keys are distributed in an obviously unsafe way is secure. The response of Burrows, Abadi, and Needham [BAN90] was that in their logic they make the assumption that principals do not divulge their keys;

[1] Honesty assumptions are relaxed in the version of BAN presented in [AT91].

since in this protocol the principals do divulge their keys, it does not satisfy the original assumption. But one can also argue that the use of an unsafe key distribution method is not the same as knowingly divulging your key.

The degree to which requirements and assumptions can vary, and the controversy that can be caused by a lack of precise understanding of what the requirements are, suggests that we need to pay more attention to understanding and stating them in a precise way. Once we have a clear and precise statement of what the goals and assumptions of a protocol are, we can attempt to prove it satisfies these goals with a high degree of confidence that we know what we are about.

In this paper we attempt to make it easier to state and reason about requirements in a precise manner by providing a requirements specification language for the NRL Protocol Analyzer. The NRL Protocol Analyzer has the advantage that it is tied to no particular set of assumptions about the kind of protocol it is used to verify. The specifier of a protocol can use it to prove that an insecure state is not reachable, or that an insecure sequence of events cannot occur; it is up to the specifier to decide what these states and sequences are. However, until now the user of the Analyzer had to specify the undesired states and sequences in terms of the protocol specification itself. Thus the requirements had to be rewritten for each protocol specification, even when the aims of the protocols were identical. With the requirements specification language, it is possible to specify a set of requirements for a class of protocols, and then map them to a particular instantiation. It is also possible to reason about the requirements in isolation without concerning ourselves with particular protocol instantiations.

The remainder of this paper is organized as follows. In section 1 we present the requirements language and give the interpretation of the language in the model of computation used by the Analyzer. We also give motivating examples of requirements of a simple authentication protocol. In section 2 we describe the NRL Protocol Analyzer and give the specification of the authentication protocol in the language used by the Analyzer. We then map the requirements specified in section 1 to the specification via responses to Analyzer queries. In section 3 we present our conclusions.

1 The Language

In this section we set out our formal requirements language. In general, our syntax is based on that of temporal logic (cf. [Gol92] or [vB91]) and in particular was motivated by the language of [Lam90] and [Aba90]; however, the intended meaning of the syntax is somewhat different than in those works. We begin with a simple example of the type of things we would like to express in our language. Then we give the general lingusitic constructs and finally the interpretation thereof in the model of computation.

1.1 An Example

In order to make clearer the abstract constructs we describe in this paper we set out some specific protocols as examples. We will return to these protocols throughout the paper to illustrate the formalisms and techniques described herein. The protocols we present are variants on an ISO draft version of a two pass one-sided message authentication protocol. [ISO91] That is, using two messages this protocol is intended to authenticate to one principal, B, that a message is current and from principal A. To make it slightly more interesting we have modified the original ISO protocol so that the confidentiality of the message is protected as well. We present two versions of the protocol, one using shared keys and one using public keys. The original ISO protocol uses only public keys and does not protect the confidentiality of the message.

Example 1.1 Shared Key Version

B sends to A: B, N_b
A sends to B: $B, N_a, N_b, \{N_a, N_b, \text{Message}\}_{K_{ab}}$

Here N_b is a nonce, a random number, generated by B, N_a is a nonce generated by A, and K_{ab} is a key shared between A and B. The last field of the second message indicates that N_a, N_b and A's message have been encrypted together using K_{ab}. □

Example 1.2 Public Key Version

B sends to A: B, N_b
A sends to B: $B, N_a, N_b, \{\{N_a, N_b, \text{Message}\}_{K_a^{-1}}\}_{K_b}$

Here N_a and N_b are nonces as before, K_a^{-1} is A's private key, and K_b is B's public key. Thus, the last field of the second message indicates that N_a, N_b, and A's message have been signed together using A's private key and then encrypted using B's public key. □

1.2 Requirements

One of the disadvantages of currently available logical languages for cryptographic protocol analysis is that for the most part each protocol has its own specification. Our approach goes some way towards a remedy by allowing a single set of requirements to specify a whole class of protocols. This has the advantage that a protocol analyst can largely identify the goals of any protocol in this class with that one specification, which seems to be a fairly intuitive way to view things. For instance one might want evaluate a protocol for two party session key distribution using ordinary public or shared key cryptography. While many of these protocols have special features and requirements, there are a number of requirements they all share—for example, that the distributed key be known only to the two principals and the server if there is one. We can express in our language general requirements for protocols for distributing session keys to two parties via a server. This specification should also satisfy protocols with no server, i.e., where one of the participants is the server. It should also work for interdomain communications. Although there are undoubtedly further requirements to be specified for servers from different domains to authenticate each other, that process should not affect the requirements for the two end parties in relation to whoever produces the key. While this is probably the type of protocol of broadest use and interest, for purposes of illustrating our requirements language and analysis technique it is clearer to stick to a simpler example than this.

What are the general security requirements for the type of authentication protocol given by our examples above? That is, if B were to accept the message from A, what need hold to preclude security violations? First of all, we need to make the distinction between an honest and a dishonest A. If A is dishonest, then we assume that A may violate any or all of the rules of the protocol, and is in collusion with the hostile intruder who we assume is trying to subvert the goals of the protocol. This does not mean that we may not put any requirements upon interchanges involving a dishonest A, but if we do, the requirements may be different than the requirements we put upon an honest A.

Now we consider a set of requirements. First, the penetrator (denoted in our requirements language by P), must not learn the content of the message. Second, A must have actually sent the message, and she must have done so after B's 'query'. We must assume that A is honest, since if A is dishonest we assume that the penetrator can learn the message as soon as A creates it and that A can send a message at any time.

In our formal language we express the requirements by indicating the temporal order in which these actions must occur. We use '$\rightarrow$' to represent the standard conditional, '$\wedge$' to represent conjunction, and '$\diamondminus$' to represent a temporal operator meaning *at some point in the past.* We assume that principals can keep track of rounds of protocols from their perspective via local round numbers, where a round number local to a principal identifies all actions pertaining to a single session as far as that principal is concerned. Thus $\mathtt{accept}(B, A, Mes, N)$ means that B accepts the message Mes as from A during B's local round N, $\mathtt{learn}(P, Mes))$ means the penetrator P learns the word Mes, $\mathtt{send}(A, B, (Query, Mes))$ means that A sends B Mes in response to query $Query$, and $\mathtt{request}(B, A, Query, N))$ means that B sends query $Query$ to A. Precise descriptions of the meaning of the syntactic expressions will be given later in the paper. For now we are simply trying to present realistic but also fairly intuitive examples of formulae in the language. We can then represent our requirements as follows:

Requirements 1.3

- $\neg(\diamondminus\mathtt{accept}(B, A, Mes) \wedge \diamondminus\mathtt{learn}(P, Mes))$
- $\mathtt{accept}(B, A, Mes, N) \rightarrow$
  $\diamondminus(\mathtt{send}(A, B, (Query, Mes)) \wedge$
  $\diamondminus\mathtt{request}(B, A, Query, N))$

□

In order to be secure a protocol must satisfy the conjunction of the requirements. They must both hold; although, it helps keep things clear if we list them separately. It will also facilitate application of the NRL Protocol Analyzer. Note that the request must come from B even though the protocols we are looking at provide no authentication of the first message. This may seem odd since A thus has no way of being sure who sent the request. Nonetheless, the request must come from B even if A does not know this: B will know whether or not the message is in response to his request when he decrypts it and checks the nonce. If the message is in response to anyone else's request, the nonce will not correspond to the one B used, and B should not accept the message as appropriate.

This also indicates that we have here just one of many possible sets of requirements. Perhaps it is not necessary that A sent the message in response to B's query, only after B's query. For example, B may be requesting the value of some sensor, and it may only be important that the sensor value be from after B's request rather than a response specifically to it. We can capture this with the following simplification of the first set of requirements.

Requirements 1.4

- $\neg(\diamondminus\mathtt{accept}(B, A, Mes) \wedge \diamondminus\mathtt{learn}(P, Mes))$
- $\mathtt{accept}(B, A, Mes, N) \rightarrow$
  $\diamondminus(\mathtt{send}(A, B, Mes) \wedge \diamondminus\mathtt{request}(B, A, N))$

□

Alternatively, it might only be important that the message from A be recent. We may require that the message be recent by B's judgement (so that B will not accept a message that arrives too late after he requested it), or recent by A's judgement (so that B will not accept a message that arrives too late after A sent it), or both. We here represent the case in which both are required.

Requirements 1.5

- $\neg(\diamondminus\mathtt{accept}(B, A, Mes) \wedge \diamondminus\mathtt{learn}(P, Mes))$
- $\mathtt{accept}(B, A, Mes, N) \rightarrow$
  $\diamondminus(\mathtt{send}(A, B, Mes) \wedge \diamondminus\mathtt{request}(B, A, N))$
- $\mathtt{accept}(B, A, Mes, N) \rightarrow$
  $\diamondminus(\mathtt{send}(A, B, Mes) \wedge \neg(\diamondminus\mathtt{time\_out}(B, N)) \wedge$
  $\neg(\diamondminus\mathtt{time\_out}(A, Mes))$

□

Another possibility is that we need to guard against replay in the sense that if B accepts a message as from an honest A, then it must never have been accepted previously by another honest principal. Or, we can make the stronger requirement that if B accepts a message as from A, then it never was accepted previously, whether or not A was honest or dishonest. In this case, since we are dealing with both honest and dishonest principals it is helpful to make a notational distinction between them. We designate an honest user A by $user(A, honest)$, a dishonest user A as $user(A, dishonest)$, and a user who may be honest or dishonest as $user(A, Y)$, where Y is a variable that may take on the value "*honest*" or "*dishonest*". The case in which we require that if an honest B accepts a message as coming from an honest A, then it was never accepted previously by any other honest user, would be represented as follows:

Requirements 1.6

- $\neg\diamondminus \texttt{accept}(user(B, honest), user(A, honest), Mes)$ $\vee \neg\diamondminus \texttt{learn}(P, Mes))$
- $\texttt{accept}(user(B, honest), user(A, honest), Mes, N)$ $\rightarrow \diamondminus(\texttt{send}(user(A, honest), user(B, honest), Mes)$ $\wedge \diamondminus \texttt{request}(user(B, honest), user(A, honest), N))$
- $\texttt{accept}(user(B, honest), user(A, honest), Mes) \rightarrow$ $\neg\diamondminus \texttt{accept}(user(C, honest), user(D, Y), Mes)$

□

Note that our requirement says that the message must not have been previously accepted by any honest user as coming from *anybody*, whether honest or dishonest. Note also that it does not matter whether the protocol uses public or shared key cryptography. Nor do we specifically require that nonces be used. For some of the above sets of requirements it may or may not be more natural to have protocols using timestamps or sequence numbers. These points should provide some indication of the generality with which requirements can be stated even when being formal. We will return to look at the last of these sets of sample requirements below, after we have precisely set out the language and its interpretation.

1.3 Syntax

Our language contains a denumerable collection of constant singular terms, typically represented by letters from the beginning of the alphabet. We also have a denumerable collection of variable terms, typically represented by letters from the end of the alphabet. We also have, for each $n \geq 1$, n-ary function letters taking terms of either type as arguments and allowing us to build up functional terms in the usual recursive fashion. (We will always indicate whether a term is constant or variable if there is any potential for confusion.) We have a denumerable collection of n-ary action symbols for each arity $n \geq 1$. These will be written as words in typewriter scrypt (e.g., `accept`). The first argument of an action symbol is reserved for a term representing the agent of the action in question.

An atomic formula consists of an n-ary action symbol, e.g., '`act`' followed by an n-tuple of terms. We have the usual logical connectives: $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$, and also one temporal operator: $\diamondminus$. Complex formulae are built up from atomic formulae in the usual recursive fashion. Since we have already seen examples of formulae, we proceed directly to their interpretation. (Note that this is only a formal language, not a logic; hence there are no axioms or inference rules.)

1.4 Interpretations

The key notion to understand is that of an action. For us actions are transitions from one state to another. We represent these semantically by ordered pairs of the form (s, s'), where 's' represents the state prior to the action and 's'' represents the state subsequent to the action. The precise way this works is given in the definition of an interpretation.

Definition 1.7 A *state space* is a non-empty set S, and each $s \in S$ is a *state*. We represent time digitally using the integers. A *trace* is a sequence σ of elements of S that is infinite in both directions, for example, $\ldots, s_{i-1}, s_i, s_{i+1}, \ldots$. We can thus equate a trace with a function from times to states. If s is the value of $\sigma(t)$, we will generally adopt the notational convenience of representing this by 's_t'. Let α and β be formulae. An *interpretation* is a function I from atomic formulae of the language to subsets of $S \times S$, i.e., $I(\alpha) \subseteq S \times S$ for any atomic formula α.

A *model* is an ordered 4-tuple, $\langle S, I, \sigma, t\rangle$ such that S is a state space, I is an interpretation, σ is a trace, and t is a time. The *satisfaction relation*, $\models$, is a relation between models and formulae. It is our way of specifying which formuale are true: given a formula α and a model $\langle S, I, \sigma, t\rangle$, '$\langle S, I, \sigma, t\rangle \models \alpha$' means that α is true at $\langle S, I, \sigma, t\rangle$. It is defined as the smallest relation between models and formulae satisfying the following:

$$
\begin{aligned}
\langle S, I, \sigma, t\rangle \models \alpha &=_{df} (s_t, s_{t+1}) \in I(\alpha) \\
\langle S, I, \sigma, t\rangle \models \neg\alpha &=_{df} \langle S, I, \sigma, t\rangle \not\models \alpha \\
\langle S, I, \sigma, t\rangle \models \alpha \wedge \beta &=_{df} \langle S, I, \sigma, t\rangle \models \alpha \text{ and } \langle S, I, \sigma, t\rangle \models \beta \\
\langle S, I, \sigma, t\rangle \models \alpha \vee \beta &=_{df} \langle S, I, \sigma, t\rangle \models \alpha \text{ or } \langle S, I, \sigma, t\rangle \models \beta \\
\langle S, I, \sigma, t\rangle \models \alpha \rightarrow \beta &=_{df} \langle S, I, \sigma, t\rangle \not\models \alpha \text{ or } \langle S, I, \sigma, t\rangle \models \beta \\
\langle S, I, \sigma, t\rangle \models \alpha \leftrightarrow \beta &=_{df} \langle S, I, \sigma, t\rangle \models \alpha \rightarrow \beta \text{ and } \langle S, I, \sigma, t\rangle \models \beta \rightarrow \alpha \\
\langle S, I, \sigma, t\rangle \models \diamondminus\alpha &=_{df} \langle S, I, \sigma, t'\rangle \models \alpha \text{ for some } t' \text{ such that } t' < t
\end{aligned}
$$

Given a class of models κ, we say that a formula α has a κ-*model* or is κ-*satisfiable* if there exists $\langle S, I, \sigma, t\rangle \in \kappa$ such that $\langle S, I, \sigma, t\rangle \models \alpha$. We say that α is κ-*valid* if

$\langle S, I, \sigma, t\rangle \models \alpha$ for all $\langle S, I, \sigma, t\rangle \in \kappa$. This is written $\models_\kappa \alpha$. When κ is clear from context or when κ is the class of all models we drop explicit reference to it in these expressions. □

1.5 Models of Computation

We will not be looking at the class of all models for purposes of protocol analysis. We now set out the class we will be using. We begin by describing some of the technical machinery we need. Our description of states and actions is motivated primarily by the formalisms operated on by the NRL Protocol Analyzer. (Recall that our goal is to use the Analyzer as a model checker to see if a given protocol meets a set of requirements.)

The model used by the Protocol Analyzer is an extension of the Dolev-Yao model [DY83]. We assume that the participants in the protocol are communicating in a network under the control of a hostile intruder who may also have access to the network as a legitimate user or users. The intruder has the ability to read all message traffic, destroy and alter messages, and create his own messages. Since all messages pass through the intruder's domain, any message that an honest participant sees can be assumed to originate from the intruder. Thus a protocol rule describes, not how one participant sends a message in response to another, but how the intruder manipulates the system to produce messages by causing principals to receive certain other messages.

As in Dolev-Yao, the words generated in the protocol obey a set of reduction rules (that is, rules for reducing words to simpler words), so we can think of the protocol as a machine by which the intruder produces words in the term-rewriting system. Also, as in Dolev-Yao, we make very strong assumptions about the knowledge gained when an intruder observes a message. We assume that the intruder learns the complete significance of each message at the moment that it is observed. Thus, if the intruder sees a string of bits that is the result of encrypting a message from A to B with a session key belonging to A and B, he knows that is what it is, although he will not know either the message or the key if he has not observed them.

A specification in the Protocol Analyzer describes how one moves from one state to another via honest participants sending data, honest participants receiving data, honest participants manipulating stored data, and the intruder's manipulation of data sent by the honest participants. Dishonest participants are identified with the intruder, and so are not modeled separately. The sending and receipt of messages by the intruder is not modeled separately, since it is automatically assumed that any message sent is received by the intruder, and any message received is sent by the intruder, even if it is only passed on by the intruder unchanged. Thus every receipt of a message by an honest principal implies the sending of a message by the intruder, and every sending of a message by an honest principal implies the receipt of a message by the intruder.

Given this, we look at the notion of a state more closely. One of the primary components of a state is a learned fact. Each honest protocol participant possesses a set of learned facts. Each learned fact is relevant to a given round of the protocol. A learned fact is described using an *lfact function*, which has four arguments. The first identifies the participant A for whom it is a learned fact. This will give us the agent of an action. The second identifies the round of the protocol via a round number that is local to the principal denoted by the first argument. This will allow each principal to attach each relevant action to a particular round of a particular protocol. The third indicates the nature of the fact. Generally this will indicate the action that the agent is taking. The fourth gives the present value of A's counter. In effect, this gives us a local clock value. The value of the lfact is either a list of words that make up the content of the fact, or if the fact does not have any content, it is "[ ]", the empty list.

One way we represent actions semantically is via changes in learned facts; however, we do not allow arbitrary changes in the value of lfact. A nonempty list can be the value of lfact for a given principal, round, and action, at the principal's local time T only if the value of lfact for that principal, round, and action, at the time immediately prior to T was [ ].

Thus, for example, suppose that A has attempted to initiate a conversation with B during local round N at time T. This can be expressed by the action (s, s') where the difference between s and s' is that in s,

lfact(user(A,honest),N,init_conv,T) = [ ]

and in s',

lfact(user(A,honest),N,init_conv,T+1) = [user(B)]

At any time prior to T, the value of the lfact would also be [ ].

It is also useful to allow certain actions to be 'forgotten'. This is accomplished by having a transition in which the value of lfact goes from a nonempty list to [ ].

Another component of a state is the intruder's knowledge, represented as a monotonically nondecreasing function of time. It is necessary to represent this in a manner distinct from the learned facts because the Analyzer represents the intruder in a different way than it represents ordinary principals. There are two kinds of actions associated with intruder knowledge that we allow. In the first of these, the intruder learns some word, that is, a string of symbols. For instance, suppose that A sends a message W to B at time $t1$, and the intruder intercepts (and thus learns) W at time $t2$. According to what we have set out above, this can be represented by (s, s'), where in s,

lfact(user(A),N,send_to_B,T) = [ ]

and in s',

lfact(user(A),N,send_to_B,T+1) = [W]

Then, the intruder learning of this action is given by (s', s''), where the only change from s' to s'' is that in s'' we have

$$\text{intruderknows(t1)} = \text{intruderknows(t2)} \cup \{[W]\}$$

where intruderknows(t) is the set of words known by the intruder at the global time t, and t1 and t2 are the global times corresponding to A's local times T and T + 1, respectively.

The second way the intruder may increase his knowledge is by performing some available internal operations on things he already knows. In other words, assuming ω is some n-ary operation of which the intruder is capable, if $\{W_1, \ldots, W_n\} \subseteq$ intruderknows(t), then

$$\text{intruderknows(t2)} = \text{intruderknows(t1)} \cup \{\omega(W_1, \ldots, W_n)\},$$

where t1 and t2 are again global times.

Definition 1.8 The four types of actions just given will be called '*basic actions*'. A *basic model* is one in which, for any given trace σ, one basic action may occur per unit time, and these specify the only allowable differences between a state and its successor. □

While basic models provide us with a simple model of computation in which to interpret the expressions of our language, they are too simple to be practical in most cases, especially as a basis for analysis using the NRL Protocol Analyzer. What we would like is a model in which state transitions can be complex enough to be useful but simple enough to provide assurance that our model is a reasonable one. To this end we introduce compressed models.

Definition 1.9 A *compressed model* is a model $\mathcal{M}$ for which there exists a basic model $\mathcal{M}'$ satisfying the following:

- The state space and interpretation for $\mathcal{M}$ and $\mathcal{M}'$ is the same.
- The trace σ in $\mathcal{M}$ is a subtrace of σ', the trace in $\mathcal{M}'$

□

In particular, this means that for every transition (s_t, s_{t+1}) in σ, there exists a subsequence of σ', $(\sigma'(i), \ldots, \sigma'(i+n))$, such that $s_t = \sigma'(i)$ and $s_{t+1} = \sigma'(i+n)$. Now that we have the essentials of our semantics worked out, we can look at the satisfaction of the requirements that we mentioned above. We focus on requirements 1.4 for example.

Recall the two formulae constituting the requirements:

- $\neg(\diamondminus \mathtt{accept}(B, A, Mes) \wedge \diamondminus \mathtt{learn}(P, Mes))$
- $\mathtt{accept}(B, A, Mes, N) \rightarrow \diamondminus(\mathtt{send}(A, B, Mes) \wedge \diamondminus \mathtt{request}(B, A, N))$

We can give a very simple description of the computational truth conditions of these requirements. For example, $\langle S, I, \sigma, t\rangle \models \mathtt{accept}(B, A, Mes, N)$ iff in s_t lfact(user(B), N, accept_from_A, T) = [ ] and in s_{t+1} lfact(user(B), N, accept_from_A, T+1) = [Mes]. This is of course not very revealing. While what constitutes a send or receive action should be immediately clear, an `accept` action is somewhat complex. Thus, while we can present a model with such a simple interpretation, we need to give a more detailed interpretation of an `accept` action if we are to get any use out of it.

It would be hopeless to give general truth conditions for an `accept` action. Fortunately, at this point we can turn to the protocol in question to see what would constitute a reasonable interpretation of accepting a message. Accepting a message is what occurs when all the relevant checks have been verified by the accepting principal. Thus, for the protocol of example 1.2 we would have as part of the first state of the `accept` action that the nonce of the second message be verified as the same nonce that was sent in the first message. There are other things to verify as well, and different protocols generally have different sets of checks to verify as conditions on an acceptance of a message. Of course the atomic actions and their interpretations can be quite different when we move to an entirely distinct class of protocols, e.g., key distribution protocols. The exact details of this will be set out below when we describe how to specify the protocol for the Analyzer.

Once we have set an interpretation for all of the expressions used in the statement of requirements and have specified the protocol itself, we are in a position to determine whether or not the protocol meets the requirements. Given a fixed state space S and interpretation I, we consider the class κ of all models $\langle S, I, \sigma, t\rangle$ for which σ is a trace of the protocol specification. To see if the protocol meets the requirements we simply see if the formulae that constitute the requirements are valid in κ. Of course, while the check is very simple in theory, it is rather difficult in practice. This is where the NRL Protocol Analyzer comes in: it helps us to make the determination. That is, to see if the protocol meets the requirements we present the Analyzer with the requirements and the interpretation of atomic actions therein. We then ask it to determine if the models in κ are a subclass of those that make the requirements true. We will show how to do this below for the sample protocol and sample requirements presented above. The analysis is primarily conducted in the language of the Analyzer, which for us amounts to a semantic description language. Thus, we present a description of the Analyzer and its language before further examining our sample protocol with respect to our sample requirements.

2 The NRL Protocol Analyzer

2.1 The Specification Language Used by the NRL Protocol Analyzer

A specification in the NRL Protocol Analyzer consists of four sections. The first section consists of transition rules governing the actions of honest principals. It may also contain rules describing possible system failures that are not necessarily the result of actions of the intruder, for example, the compromise of a session key. The second section describes the operations that are available to the honest principals and possibly to the intruder, e.g., encryption and decryption. The third section describes the atoms that are used as the basic building blocks of the words in the protocol. The fourth section describes the rewrite rules obeyed by the operations.

A transition rule has three parts. The first part gives the conditions that must hold before the rule can fire. These conditions describe the words the intruder must know (that is, the message that must be received by the principal), the values of the lfacts available to the principal, and any constraints on the lfacts and words. At the moment, the syntax of the constraints on words is somewhat restricted; they can only say that words must or must not be of a given length or that they must or must not be equal to other words. The second part describes the conditions that hold after the rule fires in terms of words learned by the intruder (that is, the message sent by the principal) and any new values taken on by lfacts. Each time a rule fires, the principal's local time is incremented; this is also recorded in the preconditions and postconditions of the rule. The third part of the rule consists of an event statement. It is used to record the firing of a rule and is useful for indicating what the rule does. It is derived from the first two parts of the rule. The event statement describes a function with four arguments. The first gives the name of the relevant principal. The second gives the number of the protocol round. The third identifies the event. The fourth gives the value of the principal's counter after the rule fire. The value of the event is a list of words relevant to the event.

An example of a rule is the following. Suppose we are at the point in the ISO protocol in which an honest principal, user(honest,B), has decided to request a message from another principal, user(A,Y), and sends him a nonce. This can be modeled by the following rule:

```
If:
count(user(B,honest)) = [M],
lfact(user(B,honest),N,recwho,M) =
     [user(A,Y)],
not(user(A,Y) = user(B,honest)),
then:
count(user(B,honest)) = [s(M)],
intruderlearns([user(B,honest),
                rand(user(B,honest),M)]),
lfact(user(B,honest),N,recsendsnonce,s(M)) =
[rand(user(B,honest),M)],
EVENT:
event(user(B,honest),N,requestedmessage,s(M))
     = [user(A,Y),rand(user(B,honest),M)].
```

In this rule the recwho lfact is used to hold the name of the user user(B,honest) is trying to talk too, and the recsendsnonce lfact holds the random nonce that user(B,honest) sends to user(A,Y). The event statement going with this rule is denoted by "requestedmessage" and holds the words used in this rule: namely, the name of user(A,Y) and the nonce.

The second section of the specification defines the operations that can be made by honest principals and by the intruder. If an operation can be made by the intruder, the Analyzer translates it into a transition rule similar to the above, except that the relevant principal is the intruder instead of an honest principal, and no lfacts are involved. An example of a specification of an operation is the following, describing public key encryption:

```
fsd1:pke(X,Y):length(X) = 1:
     length(pke(X,Y)) = length(Y):pen.
```

The term "fsd" stands for "function symbol description." The next term gives the operation and the arguments. The third gives conditions on the arguments. In this case, we make the condition that the key be a certain length, which in this case we make a default unit length one. The next term gives the length of the resulting word, which in this case is the length of Y. The last field is set to "pen" if we are assuming that the penetrator can perform the operation, and "nopen" if we are assuming that he can't. Thus the decision to put "pen" or "nopen" into the last field may vary with our assumptions about the environment in which the protocol is operating.

Some operations are built into the system. These are: concatenation, taking the head of a list, taking the tail of a list, and id_check, which is used by an honest principal to determine whether or not two words are equal.

The third section describes the words that make up the basic building blocks. We call these words "atoms". Examples would be user names, keys, and random numbers. Again, we indicate whether or not the word is known to the intruder in the last field of an atom specification, it is "known" if the intruder knows it initally, and "notknown" if the intruder doesn't know it initially.

The last section describes the rewrite rules by which words reduce to simpler words. An example of a rewrite rule would be one which describes the fact encryption with corresponding public and private keys cancel each other out:

```
rr1: pke(privkey(X),pke(pubkey(X),Y)) => Y.
rr2: pke(pubkey(X),pke(privkey(X),Y)) => Y.
```

One queries the Analyzer by asking it to find a state that consists of a set of words known by the intruder, a set of lfacts, and a sequence of events that must have occurred. One can put conditions on the words, lfacts and events by putting conditions on the words that appear in them.

One can also put conditions on the results by specifying that certain sequences of events must *not* have occurred.

The Analyzer then matches up output of each rule with the specified state, if possible, by performing substitutions on the output that make it reducible, via the reduction rules, to the state specified. It may match either the entire state or some subset. The input of the rule together with any part of the state then becomes a new state to be queried.

The way in which the Analyzer interprets rules allows considerable freedom in how the matching is done. Variables are local to rules, and, each time a rule is applied, a new set of variables is generated. This allows the Analyzer, for example, to develop scenarios involving multiple instantations of protocol rounds, as well scenarios in which the same principal plays more than one role.

2.2 An Example Specification

In this section we give the specification of the modified ISO protocol in example 1.2. The protocol consists of two messages. In the first message, a principal who wishes to receive a message sends a nonce to the principal he wishes to receive a message from. In the next message the sender sends a message, the receiver's nonce, and his own nonce, signed and encrypted. The specification is given below.

In the first two transitions, an honest user, user(B,honest), sends a request for a message to user(A,Y) , who may or may not be honest. In the first, user(B,honest) chooses user(A,Y); in the second, he sends the request.

```
/*user(B,honest) chooses sender of message*/

rule(1)
If:
count(user(B,honest)) = [M],
then:
count(user(B,honest)) = [s(M)],
lfact(user(B,honest),M,recwho,s(M)) =
      [user(A,Y)],
EVENT:
event(user(B,honest),M,chosewho,s(M)) =
      [user(A,Y)].

/*user(B,honest) sends random number*/

rule(2)
If:
count(user(B,honest)) = [M],
lfact(user(B,honest),N,recwho,M) =
      [user(A,Y)],
not(user(A,Y) = user(B,honest)),
then:
count(user(B,honest)) = [s(M)],
intruderlearns([rand(user(B,honest),M)]),
lfact(user(B,honest),N,recsendsnonce,s(M)) =
      [rand(user(B,honest),M)],
EVENT:
event(user(B,honest),N,requestedmessage,s(M))
      = [user(A,Y),rand(user(B,honest),M)].
```

In the third transition, user(A, honest) recieves a request for a message from user(B,X), who may or may not be honest. He sends an encrypted, signed, message to user(B,X), including the token that was sent. He checks that the length of the word is 1 (since this unit length is the length specified for these random numbers in the later part of the specification) and he also checks that user(B,X) is not the same as user(A,honest) (since if that were the case the use of this protocol and the cancellation properties of public-key cryptography would result in an unsigned, unencrypted message being sent. He sends an encrypted, signed, message to user(B,X), including the token that was sent.

```
/*User A sends out signed message with random
      number attached*/

rule(3)
If:
count(user(A,honest)) = [M],
intruderknows([user(B,X),W1]),
length(W1) = 1,
not(user(B,X) = user(A,honest)),
then:
count(user(A,honest)) = [s(M)],
intruderlearns([rand(user(A,honest),M),
      W1,user(B,X),
      pke(pubkey(user(B,X)),
        pke(privkey(user(A,honest)),
          (rand(user(A,honest),M),
             W1,user(B,X),
             mess(user(A,honest),M))))]),
EVENT:
event(user(A,honest),N,sentsignedmess,s(M)) =
      [rand(user(A,honest),M),W1,user(B,X),
        mess(user(A,honest),M)].
```

In the next two transitions, user(B,honest) receives the message and checks it. If it passes the tests, he accepts it as coming from user(A,Y).

```
/*User B receives message and verifies it*/

rule(4)
If:
count(user(B,honest)) = [R],
intruderknows([T1,W1,user(B,honest),Mes]),
lfact(user(B,honest),M,recwho,R) =
      [user(A,Y)],
lfact(user(B,honest),M,recsendsnonce,R) =
      [W1],
then:
count(user(B,honest)) = [s(R)],
lfact(user(B,honest),M,recmessage,s(R)) =
      [tail(tail(tail(
        pke(pubkey(user(A,honest)),
          pke(privkey(user(B,honest)),
            Mes)))))],
lfact(user(B,honest),M,checkaddress,s(R)) =
      [id_check(head(tail(tail(
          pke(pubkey(user(A,honest)),
            pke(privkey(user(B,honest)),
```

```
                Mes))))),
          user(B,honest))],
lfact(user(B,honest),M,checknonce,s(R)) =
        [id_check(head(tail(
          pke(pubkey(user(A,honest)),
          pke(privkey(user(B,honest)),Mes)))),
            W1)],
EVENT:
event(user(B,honest),M,recsignedmess,s(R)) =
        [Mes].
```

In the above rule, "head" is used to denote the first element of a list, and "tail" denotes what is left after the first element is removed. Thus, for example, head(tail((a,b,c))) = b.

```
/*User B accepts message if check succeeds*/

rule(5)
If:
count(user(B,honest)) = [R],
lfact(user(B,honest),M,recwho,R) =
        [user(A,Y)],
lfact(user(B,honest),M,recsendsnonce,R) =
        [Nonce],
lfact(user(B,honest),M,checkaddress,R) =
        [ok],
lfact(user(B,honest),M,checknonce,R) = [ok],
lfact(user(B,honest),M,recmessage,R) = [S1],
then:
count(user(B,honest)) = [s(R)],
lfact(user(B,honest),M,recwho,s(R)) = [ ],
lfact(user(B,honest),M,recmessage,s(R)) =
        [ ],
lfact(user(B,honest),M,recsendsnonce,s(R)) =
        [ ],
lfact(user(B,honest),M,accept,s(R)) =
        [user(A,Y),S1],
EVENT:
event(user(B,honest),M,acceptmess,s(R)) =
       [user(A,Y),S1].
```

The remainder of the specification describes the way words are generated and operations the intruder can perform. They are listed below. The functions symbol specification describes the function symbol pke designating the public key encryption function. The atom specification describes the various basic words produced by the system such as user names and private and public keys. Finally, the reduction rule section describes the various reduction rules that operate: in this case, we make the assumption that encryption with corresponding public and private keys cancels out.

```
fsd1:pke(X,Y):length(X) = 1:
     length(pke(X,Y)) = length(Y):pen.

atom1:user(A,X):1:known.
atom2:mess(user(A,dishonest),N):1:known.
atom3:mess(user(A,honest),N):1:notknown.
atom4:privkey(user(A,honest)):1:notknown.
atom5:privkey(user(A,dishonest)):1:known.
atom6:pubkey(user(A,X):1:known.
atom7:rand(user(A,honest),N):1:notknown.
atom8:rand(user(A,dishonest),N):1:known.

rr1: pke(privkey(X),pke(pubkey(X),Y)) => Y.
rr2: pke(pubkey(X),pke(privkey(X),Y)) => Y.
```

2.3 Mapping the Requirements to the Specification

In this section we describe how the statements in requirements 1.6 would be mapped to the protocol so that they could be verified using the Protocol Analyzer. We also show in detail how one of the statements is verified using the Analyzer.

When we present a query to the Analyzer, we have several options. We can ask it to find a set of lfact values, a set of words the intruder knows, a sequence of events that occurred, or some combination of the above. We can also put conditions on the results it finds. We can require that words have certains properties, and require that certain sequences of events do *not* occur.

We use the Analyzer to attempt to prove a state is unreachable. Thus we must translate each requirement into a description of an unreachable state. This is done in two parts. First, each requirement R is translated into an equivalent requirement of the form not(R'). Secondly, the actions described in the requirement are translated into the corresponding event statement used by the Analyzer, transforming R' into a state description R'' that can be presented to the Analyzer. The Analyzer is then used to prove R'' unreachable. If this can be done, it has been proved that the requirement holds.

We begin by mapping action statements that describe actions of honest principals to event statements. Again, we note that this mapping depends on the context of the protocol. We begin by finding the point at which we decided that an honest user accepts a message and mapping the accept statement to the corresponding event statement. Thus the action statement

$$\texttt{accept}(user(B, honest), user(A, honest), Mes)$$

maps to the event statement

event(user(B,honest),M,acceptmess,T1) =
[user(A,honest),Mes].

Likewise, we find the point at which user(A,honest) sent the message to user(B,honest). Thus we have the action $\texttt{send}(user(A, honest), user(B, honest), Mes)$ mapped to the event statement

event(user(A,honest),M,sentsignedmess,Q1) =
[R,W,user(B,honest),Mes]

and the action $\texttt{request}(B, A, N)$ to the event statement

```
event(user(B,honest),N,requestedmessage,Q2) =
        [user(A,honest),R1].
```

Mapping intruder actions to the protocol specification is trickier, since intruder actions, which consist of learning words, do not map to specific transitions, but instead to any transition which can produce a word of the appropriate form. However, we recall that, in querying the Analyzer, we can ask it to produce a state in which the intruder knows a word or word. This corresponds to asking it to find a state in which the intruder learned that word in the past. If all we wish to prove is that the intruder learned that word in the past, and we are not concerned about ordering, then this is sufficient. In most cases, we are mainly concerned with proving that an intruder never learns a word; for example, we want to prove that the intruder never learns a key, not that he does not learn it before or after it is used. Thus, in most cases, the way in which the Analyzer is queried will be sufficient. In cases in which it is not, we simply discard the output in which the events occur in the wrong order.

We now show how we would present the various requirements to the Analyzer. The Analyzer is used by specifying an insecure state and showing that it is unreachable, so we use the Analyzer by specifying the negation of the requirement and showing that it is unreachable. We begin with the requirement

$$\neg(\diamondminus \texttt{accept}(user(B, honest), user(A, honest), Mes) \wedge \diamondminus \texttt{learn}(P, Mes))$$

This requirement it presented to the Analyzer by asking it to find all cases in which the accept event occurred and the word was learned.

The second requirement is

$$\texttt{accept}(user(B, honest), user(A, honest), Mes, N) \rightarrow \diamondminus(\texttt{send}(user(A, honest), user(B, honest), Mes) \wedge \diamondminus \texttt{request}(user(B, honest), user(A, honest), N))$$

It says that, if the accept event occurred, then some sequence of events must have occurred. Thus, in order to prove that this requirement is satisfied, we must prove that the state in which the accept event occurred and the previous events did not occur. Thus, we ask it to look for the case in which

```
event(user(B,honest),N,acceptmess,M) =
      [user(A,honest),Mes]
```

occurred, but the sequence of events

```
event(user(B,honest),N,requestedmessage,M1) =
      [user(A,honest),R1]
event(user(A,honest),P,sentsignedmess,Q) =
      [R,W,user(B,honest),Mes]
```

did not occur.

The third requirement is

$$\texttt{accept}(user(B, honest), user(A, honest), Mes) \rightarrow \neg\diamondminus \texttt{accept}(user(C, honest), user(D, Y), Mes)$$

It says that, if the accept event occurred, then an accept event for the same message did *not* occur in the past. In this case, we ask the Analyzer to look for the sequence of events

```
event(user(C,honest),N1,recsignedmess,M1) =
      [user(A1,Y),Mes]
event(user(B,honest),M,recsignedmess,M) =
      [user(A,honest),Mes]
```

We now examine the second requirement in detail. The proofs of the other two are similar, but more lengthy.

Before we began presenting the requirements to the Analyzer, we did a syntactic analysis in which we proved that a number of trivial states were unreachable. For example, we proved that certain words were unobtainable under certain conditions. We also proved that some lfacts were reachable only if certain conditions held. The results of this syntactic analysis were then fed into the Analyzer, which automatically checked for these conditions every time it produced a solution. If the conditions were not satisfied it either rejected the solution, or, if some more specific case of the solution satisfied the conditions, substituted the more specific case for the general one.

We began by asking for a complete description of all states in which the accept event occurred but the corresponding send and request events did not occur. A transcript follows. User input is preceded by "—:"

```
 ?- begin.

Give the number of the parent solution,
   if any.
|:

What words is the intruder looking for?
|:
What state variable values is the intruder
looking for?
|:

List the sequence of events that you want
to have occurred.
|: event(user(B,honest),N,acceptmess,M) =
         [user(A,honest),Mes]
|:
What conditions do you want to put
on all of these?
|:

List the sequences of events that you
don't want to have occurred.

Enter a list
|: event(user(B,honest),N,
       requestedmessage,M1) =
     [user(A,honest),R]
```

```
|: event(user(A,honest),P,sentsignedmess,Q) =
    [R1,W1,user(B,honest),Mes]
|:

Enter a list
|:
```

One solution is produced:

```
Solution number 1

The events that occurred are
   R1 = event(user(_10416,honest),[_10418],
                        acceptmess,s(_10421)) =
         [user(_10425,honest),_10427].
The lists of events to avoid are

   H1 = event(user(_10416,honest),[_10418],
                    requestedmessage,_10654) =
          [user(_10425,honest),_10666].
   H2 = event(user(_10425,honest),[_10679],
                     sentsignedmess,_10674) =
          [_10681,_10683,user(_10416,honest),
                _10427].

Input state variables are:

   S1 = count(user(_10416,honest)) = _10421.
   S2 = lfact(user(_10416,honest),[_10418],
                              recwho,_10421) =
          [user(_10425,honest)].
   S3 = lfact(user(_10416,honest),[_10418],
                      recsendsnonce,_10421) =
          [rand(user(_10416,honest),_10476)].
   S4 = lfact(user(_10416,honest),[_10418],
                   checkaddress,_10421) = [ok].
   S5 = lfact(user(_10416,honest),[_10418],
                   checknonce,_10421) = [ok].
   S6 = lfact(user(_10416,honest),[_10418],
                recmessage,_10421) = [_10427].


Rule number 5 was used.
```

We try to find out if this state is reachable by asking the Analyzer how to find the state in which the the lfacts S2, S4, S5, and S6 hold. The Analyzer attempts to match every subset of these lfacts. It turns up only one solution, the following, matching S4, S5, and S6. S2 is thus required to be part of the intput state.

```
Solution number 1.1
The events that will occur are:

   F1 = event(user(_4251,honest),[_4253],
                      acceptmess,s(s(_4258))) =
          [user(_4262,honest),_4264].
The events that occurred are
   R1 = event(user(_4251,honest),[_4253],
                      recsignedmess,s(_4258)) =
          [user(_4262,honest),
           pke(pubkey(user(_4251,honest)),
            pke(privkey(user(_4262,honest)),
            (_4307,
             rand(user(_4251,honest),_4314),
              user(_4251,honest),_4264)))].
The lists of events to avoid are

  H1 = event(user(_4251,honest),[_4253],
                    requestedmessage,_4751) =
        [user(_4262,honest),_4763].
  H2 = event(user(_4262,honest),[_4776],
                      sentsignedmess,_4771) =
        [_4778,_4780,user(_4251,honest),
           _4264].

Input words are:

  W1 = _4326
  W2 = rand(user(_4251,honest),_4314)
  W3 = user(_4251,honest)
  W4 = pke(pubkey(user(_4251,honest)),
        pke(privkey(user(_4262,honest)),
         (_4307,
           rand(user(_4251,honest),_4314),
           user(_4251,honest),_4264)))
Input state variables are:

  S1 = count(user(_4251,honest)) = _4258.
  S2 = lfact(user(_4251,honest),[_4253],
                               recwho,_4258) =
        [user(_4262,honest)].
  S3 = lfact(user(_4251,honest),[_4253],
                      recsendsnonce,_4258) =
        [rand(user(_4251,honest),_4314)].
States found are:
  D1 = lfact(user(_4251,honest),[_4253],
                      recmessage,s(_4258)) =
        [_4264].
  D2 = lfact(user(_4251,honest),[_4253],
               checkaddress,s(_4258)) = [ok].
  D3 = lfact(user(_4251,honest),[_4253],
                checknonce,s(_4258)) = [ok].
```

We now ask the Analyzer how to find the state in which the intruder knows W4 and lfacts S2 and S3 hold. In this case we get two results, each matching W4 and requiring S2 and S3 to be part of the input state. Note that the second solution requires the intruder's use of the public-key encryption function.

```
Solution number 1.1.1
The events that will occur are:

  F1 = event(user(_10036,honest),[_10038],
                   recsignedmess,s(_10043)) =
      [user(_10047,honest),
        pke(pubkey(user(_10036,honest)),
         pke(privkey(user(_10047,honest)),
          (rand(user(_10047,honest),_10053),
           rand(user(_10036,honest),_10109),
           user(_10036,honest),
     mess(user(_10047,honest),_10053))))].
  F2 = event(user(_10036,honest),[_10038],
```

```
                 acceptmess,s(s(_10043))) =
      [user(_10047,honest),
        mess(user(_10047,honest),_10053)].
The events that occurred are
  R1 = event(user(_10047,honest),[_10138],
                sentsignedmess,s(_10053)) =
      [rand(user(_10047,honest),_10053),
        rand(user(_10036,honest),_10109),
        user(_10036,honest),
        mess(user(_10047,honest),_10053)].
The lists of events to avoid are

  H1 = event(user(_10036,honest),[_10038],
                 requestedmessage,_10452) =
      [user(_10047,honest),_10464].

Input words are:

  W1 = user(_10036,honest)
  W2 = rand(user(_10036,honest),_10109)
Input state variables are:

  S1 = count(user(_10047,honest)) = _10053.
  S2 = lfact(user(_10036,honest),[_10038],
                          recwho,_10043) =
      [user(_10047,honest)].
  S3 = lfact(user(_10036,honest),[_10038],
                   recsendsnonce,_10043) =
      [rand(user(_10036,honest),_10109)].

Words found are:
  E1 = pke(pubkey(user(_10036,honest)),
        pke(privkey(user(_10047,honest)),
         (rand(user(_10047,honest),_10053),
          rand(user(_10036,honest),_10109),
          user(_10036,honest),
          mess(user(_10047,honest),_10053))))
Rule number 3 was used.

Solution number 1.1.2
The events that will occur are:

  F1 = event(user(_9440,honest),[_9442],
                 recsignedmess,s(_9447)) =
        [user(_9451,honest),
          pke(pubkey(user(_9440,honest)),
          pke(privkey(user(_9451,honest)),
           (_9494,
             rand(user(_9440,honest),_9501),
             user(_9440,honest),_9453)))].
  F2 = event(user(_9440,honest),[_9442],
                 acceptmess,s(s(_9447))) =
        [user(_9451,honest),_9453].
The events that occurred are
  R1 = event(pen,[_9521],pke,s(_9521)) =
        [pubkey(user(_9440,honest)),
          pke(privkey(user(_9451,honest)),
            (_9494,
            rand(user(_9440,honest),_9501),
            user(_9440,honest),_9453))].
The lists of events to avoid are

  H1 = event(user(_9440,honest),[_9442],
                   requestedmessage,_9773) =
        [user(_9451,honest),_9785].
  H2 = event(user(_9451,honest),[_9798],
                      sentsignedmess,_9793) =
        [_9800,_9802,user(_9440,honest),
          _9453].

Input words are:

  W1 = pubkey(user(_9440,honest))
  W2 = pke(privkey(user(_9451,honest)),
        (_9494,
           rand(user(_9440,honest),_9501),
           user(_9440,honest),_9453))
Input state variables are:

  S1 = count(pen) = _9521.
  S2 = lfact(user(_9440,honest),[_9442],
                          recwho,_9447) =
        [user(_9451,honest)].
  S3 = lfact(user(_9440,honest),[_9442],
                      recsendsnonce,_9447) =
       [rand(user(_9440,honest),_9501)].

Words found are:
  E1 = pke(pubkey(user(_9440,honest)),
         pke(privkey(user(_9451,honest)),
          (_9494,
          rand(user(_9440,honest),_9501),
          user(_9440,honest),_9453)))
Rule number 301 was used.
```

Notice that, in Solution 1.1.1, the second event in our series of undesirable events has occurred. When we ask the Analyzer how to find lfacts S2 and S3 in Solution 1.1.1, it finds that the only way that this can occur is if the request event occurs. This is the third and last undesirable event in the series, and so it rejects the solution and declares the state unreachable.

In Solution 1.1.2, the Analyzer used the fact that the word W4 was of the form

```
pke(pubkey(user(_9440,honest)),
     pke(privkey(user(_9451,honest)),
        (_9494,
        rand(user(_9440,honest),_9501),
        user(_9440,honest),_9453)))
```

to prove that the word was obtainable if

```
W1 = pubkey(user(_9440,honest))
```

and

```
W2 = pke(privkey(user(_9451,honest)),
          (_9494,
          rand(user(_9440,honest),_9501),
          user(_9440,honest),_9453))
```

can be found by the intruder. All public keys are assumed to be generally known. Thus we attempt to determine whether or not the intruder can find W2. The

Analyzer finds several solutions, but in our previous syntactic analysis we proved them unreachable. Thus the Analyzer judges the word W2 to be unobtainable by the intruder, and says that the state is unreachable.

We have now followed all paths to the end and proved that each one begins in an unreachable state. Thus we have proved the original state we specified unreachable, and hence that the requirement is satisfied.

3 Conclusions

In this paper we have presented a formal language for specifying and reasoning about cryptographic protocol requirements. We have given examples of simple sets of requirements in that language. We have looked at two versions of a protocol that might meet those requirements and shown how to specify them in the language of the NRL Protocol Analyzer. We have also shown how to map one of our sets of formal requirements to the language of the NRL Protocol Analyzer and used the Analyzer to show that one version of the protocol meets those requirements.

We regard the applications in this paper as very elementary and primarily for illustrative purposes. We have begun work on more substantive and more commonly applicable requirements. In particular we have specified requirements for various types of two party key distribution protocols, including general requirements covering public or shared key protocols and requirements for protocols using Diffie-Hellman type key exchange. Interestingly, the reason we cannot cover all of the above with a general complete set of requirements is only because the session key is not produced from a single source in Diffie-Hellman schemes. We have also begun to specify requirements for resource sharing of the kind found in [BM90]. We expect to find still more applications for our language and technique in the future.

4 Acknowledgement

We would like the thank Jim Gray, Yacov Yacobi, and the anonymous referees for their careful and insightful comments.

References

[Aba90] Martín Abadi. An Axiomatization of Lamport's Temporal Logic of Action. Research Report 65, Digital Systems Research Center, October 1990.

[AT91] Martín Abadi and Mark Tuttle. A Semantics for a Logic of Authentication. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 201–216. ACM Press, August 1991.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A Logic of Authentication. Research Report 39, Digital Systems Research Center, February 1989. Parts and versions of this material have been presented in many places including *ACM Transactions on Computer Systems*, 8(1): 18–36, Feb. 1990. All references herein are to the SRC Research Report 39 as revised Feb. 22, 1990.

[BAN90] Michael Burrows, Martín Abadi, and Roger Needham. Rejoinder to Nessett. *Operating Systems Review*, 24(2):39–40, April 1990.

[BM90] J. Burns and C.J. Mitchell. A Security Scheme for Resource Shoring Over a Network. *Computers and Security*, 9:67–76, February 1990.

[DY83] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[Gol92] Robert Goldblatt. *Logics of Time and Computation, 2^{nd} edition*, volume 7 of *CSLI Lecture Notes*. CSLI Publications, Stanford, 1992.

[ISO91] ISO/IEC JTC1/SC27. *Information Technology – Security Techniques – Entity Authentication Mechanisms – Part 3: Entity Authentication Using a Public-Key Algorithm*, November 1991. Committee Draft 9798–3, version #4.

[Lam90] Leslie Lamport. A temporal logic of action. Research Report 57, Digital Systems Research Center, April 1990.

[Mea91] C. Meadows. A System for the Specification and Analysis of Key Management Protocols. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 182–195. IEEE Computer Society Press, Los Alamitos, California, 1991.

[Mea92] C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1:5–53, 1992.

[Nes90] D. M. Nessett. A Critique of the Burrows, Abadi, and Needham Logic. *Operating Systems Review*, 24(2):35–38, April 1990.

[SG92] S.G. Stubblebine and V.D. Gligor. On Message Integrity in Cryptographic Protocols. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 85–104. IEEE Computer Society Press, Los Alamitos, California, 1992.

[Syv91] Paul F. Syverson. The Use of Logic in the Analysis of Cryptographic Protocols. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 156–170. IEEE Computer Society Press, Los Alamitos, California, 1991.

[vB91] Johan van Bentham. *The Logic of Time*, volume 156 of *Synthese Library*. Kluwer Academic Publishers, Dordrecht, The Netherlands, second edition, 1991.