Formal Requirements for Key Distribution Protocols

Paul Syverson and Catherine Meadows

Center for High Assurance Computing Systems
Naval Research Laboratory
Washington, DC 20375
USA

Abstract. We discuss generic formal requirements for reasoning about two party key distribution protocols, using a language developed for specifying security requirements for security protocols. Typically earlier work has considered formal analysis of already developed protocols. Our goal is to present sets of formal requirements for various contexts which can be applied at the design stage as well as to existing protocols. We use a protocol analysis tool we have developed to determine whether or not a specific protocol has met some of the requirements we specified. We show how this process uncovered a flaw in the protocol and helped us refine our requirements.

1 Introduction

Recently, there has been a growing interest in the development and use of formal methods to analyze security properties of cryptographic protocols. Together with this increased interest, there has been a growing recognition that it is not enough to possess a means of formally specifying and analyzing a protocol; one must also have a means of formally specifying the properties that a protocol must have. One way of gaining greater assurance that one is specifying and verifying the correct properties is to develop a formal requirements language that one can use to define the properties one wants to hold for the protocol. Although the use of a formal requirements language will not guarantee by itself that one has thought of all necessary protocol requirements, it will at least assist in understanding and using the requirements.

In an earlier paper, [10], we set forth such a requirements language that was intended for use with the NRL Protocol Analyzer, an automated tool for specifying and analyzing cryptographic protocols. The Protocol Analyzer verifies that a protocol meets a set of requirements by checking that every possible run of the protocol is one over which the requirements remain valid (unless of course this is not so, in which case the Analyzer shows that instead). In other words, the Protocol Analyzer functions as a semantic model checker with respect to the requirements language. In that paper, we looked at a simple one-sided pure authentication protocol to show how one could use the requirements language to specify a number of different requirements. And we showed how we could use the Protocol Analyzer to prove that the protocol met the requirements set forth in the language.

In this paper we provide further evidence of the usefulness of our language by using it to specify more realistic protocols. In particular, we use the language to define requirements for two-party key distribution protocols with one or more servers. These are the types of protocols that have received the most interest in the verification literature; so, it is useful to have a set of requirements for comparison.

2 The Language

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 script (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: $\Diamond\!\!\!\!-$. Complex formulae are built up from atomic formulae in the usual recursive fashion. (Note that this is only a formal language, not a logic; hence there are no axioms or inference rules.)

In general, an action symbol will be of the following form. It will have four arguments, the first representing the agent of the action in question, the second representing the other principals involved in the action, the third representing the words involved in the action, and the fourth representing the local round number of the agent of the action, where a round number local to a principal identifies all actions pertaining to a single session as far as that principal is concerned. Action symbols can describe such events as a principal sending a message, the learning of a word by the intruder, or a principal's making a change to one or more of its local state variables. An action symbol may map to more than one event, and for a given event, there may be more than one action symbol mapping to it. Requirements are stated in terms of conditions on traces of action symbols. For example, we may require that an event indicated by an action symbol can only take place if some event indicated by another action symbol has taken place previously.

3 The NRL Protocol Analyzer

In this section we give a brief overview of the NRL Protocol Analyzer. More complete descriptions may be found in [7, 6].

The NRL Protocol Analyzer is a software tool that can be used either to prove theorems about security properties of cryptographic protocols, or to find flaws if the theorems turn out not to be true. The model used by the Protocol Analyzer is an extension of the Dolev-Yao model [4]. 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. Honest principals keep track of where they are in the protocol by means of local state variables. A state in the Protocol Analyzer is described by some combination of words known by the intruder, values of local state variables, and sequences of events that have occurred some time in the past. One uses the NRL Protocol Analyzer by specifying an insecure state and attempting to prove it unreachable. This is done by reducing the state space to a manageable size by proving a set of inductive lemmas about the unreachability of infinite classes of states and then performing an exhaustive search on the remaining state space. If the state is unreachable, every path to the state should begin in a state that was proved unreachable. If a state is reachable, the Analyzer should generate a path to the state. One can use the Protocol Analyzer to prove that requirements stated in the requirements language are satisfied by mapping action terms to Protocol Analyzer events. One then replaces each requirement by its negation and attempts to prove that the state specified by the negation is not reachable.

4 Two party, one server key distribution protocols

4.1 Requirements for one time authentication protocols

We begin with the requirements for a key distribution protocol with a single key server. We restrict outselves to the case in which there are two parties involved in obtaining keys, one who initiates the protocol, who we designate as the initiator, and the other, who we designate as the receiver. The server can be either a separate entity, or the initiator or receiver. For this set of requirements, we assume that the server (given that he is distinct from the two principals) is honest. Individuals attempting to communicate may be either honest or dishonest. However, we only consider requirements for communication between two honest principals together with an honest server. This is because, under our assumptions, if any party is dishonest, they will share the key with the intruder, and so the fundamental requirement of key secrecy will not be satisfied.[1]

There are some obvious requirements on such a protocol. First of all, if a key is accepted by an honest principal for communication with another honest principal, it should not be learned by the intruder, either before or after the accept event, unless as a result of some key compromise that is outside the scope of the protocol. Secondly, replays of old keys should be avoided. Thus, if a key is accepted for communication by honest principal A with honest principal B, it should not have been accepted in the past, except possibly by B for communication with A. Thirdly, if a key has been accepted for communication between A and B, then it should have been generated by a server for use between A and B. Finally, we make the more subtle requirement that, if A or B accept a key for conversation with the other and with A as an initiator, then A did in fact initiate the conversation. Thus, A and B cannot be tricked into having a conversation that neither one of them initiated.

We begin by describing the various event statements that are involved in informal requirements that we have stated so far. They are as follows.

- Initiator A requests to talk to receiver B:
  $\texttt{request}(user(A, honest), user(B, Y), (), M)$
- Server S sends a key K for communication between A and B:
  $\texttt{send}(S, user(A, X), user(B, Y), K, M)$
- Initiator A accepts a key for conversation with receiver B:
  $\texttt{init\_accept}(user(A, honest), user(B, Y), K, M)$
- Receiver B accepts a key for conversation with initiator A:
  $\texttt{rec\_accept}(user(B, honest), user(A, X), K, M)$
- Penetrator P learns a key:
  $\texttt{learn}(P, (), K, M)$
- Key is compromised:
  $\texttt{compromise}(environment, (), K, M)$

[1] In other cases, for example in our analysis of some resource-sharing protocols, we develop requirements for the interaction of an honest principal with a possibly dishonest principal.

We can now set forth the requirements[2]:

1. If a key has been accepted, it should not be learned by the intruder, except through a compromise event:

$$\diamondminus(\texttt{init\_accept}(user(A, honest), user(B, honest), K, M1) \vee \\ \texttt{rec\_accept}(user(B, honest), user(A, honest), K, M2)) \rightarrow \\ \diamondminus(\texttt{learn}(pen, (), K, M?) \rightarrow \diamondminus\texttt{compromise}(environment, (), K, M?))$$

2. If a key is accepted for communication between two parties, it should not have been accepted in the past, except by the other party. This becomes two requirements, one for the initiator and one for the receiver. Since these requirements are mirror images of each other, we present only the requirement for the initiator:

$$\texttt{init\_accept}(user(A, honest), user(B, honest), K, M1) \rightarrow \\ \neg(\diamondminus\texttt{init\_accept}(user(C, honest), user(D, X), K, M?) \wedge \\ (\diamondminus\texttt{rec\_accept}(user(C, honest), user(D, X), K, M?) \rightarrow (C = B \wedge D = A))$$

3. If a key is accepted for communication between two entities, then it must have been requested by the initiating entity and sent by a server for communication between those two entities. Again, this becomes two requirements, one for the initiator and one for the receiver.

$$\texttt{init\_accept}(user(A, honest), user(B, honest), K, M1) \rightarrow \\ \diamondminus(\texttt{send}(S, (user(A, honest), user(B, honest)), K, M?) \wedge \\ \diamondminus\texttt{request}(user(A, honest), user(B, honest), (), M1))$$

$$\texttt{rec\_accept}(user(B, honest), user(A, honest), K, M2) \rightarrow \\ \diamondminus(\texttt{send}(S, (user(A, honest), user(B, honest)), K, M?) \wedge \\ \diamondminus\texttt{request}(user(A, honest), user(B, honest), (), M?))$$

4.2 Requirements for repeated authentication

Recently a number of protocols have been proposed that explicitly include reauthentication of principals to use a previously distributed session key. ([5], [8], [12]) When session keys can safely be used for more than the length of a single session these protocols provide reauthentication with fewer messages than the number required for initial distribution and require fewer session keys to be generated (by allowing reuse). This cuts down on expense in communication and computation. More importantly, a server is only required for the initial exchange; none is necessary for reauthentication.

Since these protocols may be less familiar than those addressed in the last section, we give an example of one, taken from [12]. This example will also be

[2] In all requirements, '$M?$' is not really a variable and does not require uniform substitution of round numbers.

used later to demonstrate specific protocol analysis. It is typical of such protocols in that it produces a ticket in the initial exchange to be used during subsequent authentication. It is derived from the protocols KSL and NS, presented in [5] and [8] respectively.

Modified Neuman-Stubblebine protocol

Initial exchange

(1) *A sends to B*: A, N_a
(2) *B sends to S*: $B, \{A, N_a, T_b\}_{K_{bs}}, N_b$
(3) *S sends to A*: $\{B, N_a, K_{ab}, T_b\}_{K_{as}}, \{A, K_{ab}, T_b\}_{K_{bb}}, N_b$
(4) *A sends to B*: $\{A, K_{ab}, T_b\}_{K_{bb}}, \{N_b\}_{K_{ab}}$

The initial exchange is straightforward: it is similar to single round key distribution protocols, and we mostly use standard notation here. A and B are the two principals, and S is the server. N_x is a nonce generated by X and used by X to determine freshness. K_{xy} is a key to be used exclusively for communication between X and Y and assumed to be known only to them or those they trust. '$\{Message\}_K$' represents a message encrypted with K, where $Message$ is the corresponding cleartext. 'T_x' usually indicates a timestamp generated by X. Here we use T_b to determine the expiration time of the ticket, $\{A, K_{ab}, T_b\}_{K_{bb}}$, and associated session key, K_{ab}. This ticket can be used for subsequent authentication. Following Kehne et al., we use 'K_{bb}' to represent a key used exclusively to produce a ticket to be checked only by B. Not following Kehne et al., the ticket key is assumed to be known to the server as well as to B. However, the server is expected to use it only for this purpose. And, B is expected to be able to detect the error should he receive either a putative ticket encrypted with K_{bs} or a non-ticket encrypted with K_{bb}. We now give the subsequent authentication part of the protocol.

Subsequent authentication

(1′) *A sends to B*: $N'_a, \{A, K_{ab}, T_b\}_{K_{bb}}$
(2′) *B sends to A*: $N'_b, \{N'_a\}_{K_{ab}}$
(3′) *A sends to B*: $\{N'_b\}_{K_{ab}}$

In the first message, A generates a new nonce and sends this to B, along with the ticket from the initial exchange. B then checks the expiration time of the ticket. If the key is still good he generates his own new nonce, which he sends to A. He also sends her back the nonce she generated encrypted with the session key. Since this key is used only by A and B and since she knows the nonce is fresh, upon her receiving this, B will be authenticated to A. Finally, A encrypts B's nonce with the session key and sends it back to him, thus authenticating A to B. We will return to look at this protocol in more detail below.

The requirements set out above are for protocols where the distributed key is only to be used for one session. While these requirements may be generically adequate for the initial exchange of a protocol allowing repeated authentication, further requirements are necessary for the subsequent authentication subprotocol. Also necessary is a small but significant change to handle the subtleties introduced to our notion of currency by such protocols. Until recently the primary approach to currency, i.e. connection to a particular protocol round, has been via authentication and freshness. One showed that a message was both recent and originated by the correct principal in order to show currency to a given round.[3] Recently, a number of papers have shown how to interleave messages from simultaneous rounds to produce attacks. (Cf., e.g., [1], [3], [9], [11].) Against such interleaving attacks freshness is no guarantee of currency.

The matter only becomes more complex for repeat authentication protocols. These protocols need to be concerned simultaneously with currency within a round and currency to a class of rounds: we must make sure that the messages involved in the reauthentication are current and that the session key involved is also current. These are two different judgements of currency. For, if currency is bounded only by connection to the initial exchange, then there is no need for reauthentication. And, if currency is relative only to the reauthentication, then the session key is no longer current.

Within a single protocol round, we must be concerned with freedom from interleaving attacks (whether it is the initial exchange or reauthentication that is executed). However, across multiple reauthentications of a given protocol such concern may or may not be important. For, if two principals were to have more than one ticket currently acceptable for potential reauthentication, there may be no problem in a random choice of either one to begin a round. In this case, there can be no question of interleaving because there is no notion of a single round across repeated authentications. On the other hand, if we wanted to reserve specific multisession keys for particular types of communication between principals, then we could conceivably have interleaving attacks: a principal could be tricked into using a key for one class of communication that was meant for another. We make some small adjustments to the structure of event statements in order to allow enough flexibility to express the types of requirements germane to these issues.

The event statements have the usual format of an action symbol with four arguments. The fourth argument, for round numbers, now is of the form $N.M$. The M indicates the local round number as before. The N indicates an extended local round number, which may or may not be required to stay the same across repeated authentications. In order to address currency of the session key, the third argument is now an ordered pair, e.g., (K, T). This gives both a key and an expiration time. Note that in the following event statements metalinguistic use of 'initiator' refers to the initiator of subsequent authentication, who need not be the initiator of the initial exchange. Similarly for 'receiver'. In addition to the previous event statements, appropriately reformatted, we have the following.

[3] We use 'current' as an attempt at a neutral term meaning connected in some appropriate sense to given protocol round(s).

– Reauthentication initiator A requests a subsequent session with receiver B:

$$\texttt{rerequest}(user(A, honest), user(B, Y), (), N.M)$$

– Reauthentication initiator A accepts a key for a subsequent conversation with receiver B:

$$\texttt{init\_reaccept}(user(A, honest), user(B, Y), (K, T), N.M)$$

– Reauthentication receiver B accepts a key for subsequent conversation with initiator A:

$$\texttt{rec\_reaccept}(user(B, honest), user(A, X), (K, T), N.M)$$

– Session key is assessed to have timed-out by principal A:

$$\texttt{timeout}(user(A, honest), user(X, Y), (K, T), N.M)$$

It might seem that in order for a principal to accept a key for a subsequent session we should require that it was properly requested for initial exchange, sent by the server, etc. Much of this can be accomplished, however, simply by requiring that for a principal to reaccept a key he should have accepted it in a previous session. This will in turn force all the requirements that such acceptance implies. We can thus focus exclusively on the requirements for the reacceptance of the key.

The requirements are then as follows:

1. If a key has been accepted for subsequent use, it should not be learned by the intruder, except through a compromise event. (This is virtually the same as requirement 1 of the last section.)

$$\begin{aligned}
&\diamondminus(\texttt{init_reaccept}(user(A, honest), user(B, honest), (K, T), N1.M1)\vee \\
&\quad \texttt{rec_reaccept}(user(B, honest, user(A, honest), (K, T), N2.M2)) \rightarrow \\
&\diamondminus(\texttt{learn}(pen, (), K, M?) \rightarrow \diamondminus\texttt{compromise}(environment, (), K, M?))
\end{aligned}$$

2. If a key is accepted for subsequent use, then it should have been previously accepted by both principals in an initial exchange. As above, this yields two requirements, one for the initiator and one for the receiver. Since one is the mirror image of the other, we only give the first of these requirements.

$$\begin{aligned}
&\texttt{init_reaccept}(user(A, honest), user(B, honest), (K, T), N1.M1) \rightarrow \\
&\quad (\diamondminus(\texttt{init_accept}(user(A, honest), user(B, honest), (K, T), N1.M?)\wedge \\
&\quad \diamondminus\texttt{rec_accept}(user(B, honest), user(A, honest), (K, T), N?.M?))\vee \\
&\quad (\diamondminus(\texttt{rec_accept}(user(A, honest), user(B, honest), (K, T), N1.M?)\wedge \\
&\quad \diamondminus(\texttt{init_accept}(user(B, honest), user(A, honest), (K, T), N?.M?))
\end{aligned}$$

3. If a key is accepted for subsequent use, then a subsequent session must be requested by the initiator:

$$\begin{aligned}
&\texttt{init_reaccept}(user(A, honest), user(B, honest), (K, T), N1.M1) \rightarrow \\
&\qquad \diamondminus\texttt{rerequest}(user(A, honest), user(B, honest), (K, T), N1.M1))
\end{aligned}$$

$$\begin{aligned}
&\texttt{rec_reaccept}(user(B, honest), user(A, honest), (K, T), N2.M2) \rightarrow \\
&\qquad \diamondminus\texttt{rerequest}(user(A, honest), user(B, honest), (K, T), N?.M?))
\end{aligned}$$

This requirement assumes that the request is for authenticating a particular key. If we need only that the request is for the authentication is for some current key we have:

$$\texttt{init\_reaccept}(user(A, honest), user(B, honest), (K, T), N1.M1) \rightarrow \\ \Diamond\texttt{rerequest}(user(A, honest), user(B, honest), (), N1.M1))$$

$$\texttt{rec\_reaccept}(user(B, honest), user(A, honest), (K, T), N2.M2) \rightarrow \\ \Diamond\texttt{rerequest}(user(A, honest), user(B, honest), (), N?.M?))$$

4. If a key is accepted for subsequent use, it should not have previously expired:

$$\texttt{init\_reaccept}(user(A, honest), user(B, honest), (K, T), N1.M1) \rightarrow \\ \neg(\Diamond\texttt{timeout}(user(A, honest), user(X, Y), (K, T), N?.M?)))$$

$$\texttt{rec\_reaccept}(user(B, honest), user(A, honest), (K, T), N2.M2) \rightarrow \\ \neg(\Diamond\texttt{timeout}(user(B, honest), user(X, Y), (K, T), N?.M?)))$$

5 Analysis of a Modified Version of the NS Protocol

In this section we describe how we applied the set of requirements developed in this paper to use the NRL Protocol Analyzer to evaluate a version of the reauthentication protocol of Neuman and Stubblebine [8]. This lead to the discovery of an implementation-dependent flaw similar to the one found in [12] and [2], as well as of an attack that pointed out a place where our requirements might be too stringent. As in the earlier case, the discovery of an implementation-dependent flaw does not mean that implementations of the protocol are necessarily or even likely to be flawed, but rather that there is a hidden assumption in the specification whose violation would cause a security flaw. In this case, as in the flaw discovered in [12] and [2], the hidden assumption is that the principals have the ability to recognize different types of data, such as keys, nonces, and timestamps.

In [12] and [2] an attack was found on the Neuman-Stubblebine protocol which depends upon the receiver's inability to distinguish a nonce from a key. We do not present the attack here, but note that it depends upon the receiver's confusing the message it generates in the second step in the protocol with the message it receives in the fourth step. It was conjectured in [12] that this attack could be foiled by using two different encryption keys for the two messages. Thus each principal B would share two keys with the server, K_{bs} and K_{bb}. We attempted to verify this claim by applying the NRL Protocol Analyzer to the requirements set forth in this paper. What we found was that, although the attack on the receiver's key no longer succeeded, it was possible to mount a similar attack on the initiator's key.

We did this by specifying the modified Neuman-Stubblebine protocol and ran the the NRL Protocol Analyzer on the requirement that, if a key is accepted as good by the sender, then it must have been requested by the sender and subsequently generated by a key server.

We attempted to verify that the protocol satisfied this requirement by showing that the negation of the requirement was unreachable. In other words, we attempted to show that there was no path to the state in which the initiator of the protocol had accepted a key as good, but in which the sequence in which the initiator requested a key and the key server had generated the key did not occur. The Analyzer was able to generate the following path by which such a state could be reached.

(1) A *sends to* E_b: A, N_a
(1*) E_b *sends to* A: B, N_a
(2*) A *sends to* E_s: $A, \{B, N_a, T_a\}_{K_{as}}, N_a'$
(2) Omitted.
(3) E_s *sends to* A: $\{B, Na, T_a(= \{K_{ab}, T_b\})\}_{K_{as}}, Garbage_1, Garbage_2$

The attack is subtle, and makes use of the interleaving of two instances of the protocol, one initiated by A with B, and one initiated by the intruder acting as B attempting to initiate an instance of the protocol with A. In (1), A sends a message to B initiating a session with B. This is intercepted by the intruder E. In (1*), E impersonating B attempts to initiate a session with A, this time sending N_a as B's nonce. In (2*), A encrypts B's message together with a timestamp and forwards it to S. This message is also intercepted by E. In (3), E forwards the encrypted message from (2*) as if it were the server's response to B's response to A's initial message. The last two parts of the message are not used by A, so E can substitute anything she likes. A decrypts the message and checks for the nonce. She then assumes that T_a must be $\{K_{ab}, T_b\}$.

We also ran the Protocol Analyzer on the same requirement from the point of view of a receiver B. In this case we were able to prove that, if B accepts a word as a key, then that word must have been generated as a key by a key server. In other words, B cannot be fooled into accepting a piece of a timestamp as a key. However, if the intruder E is able to find out the timestamp, then E can use T_a to impersonate B to A. Since timestamps may not be as well protected as keys, this may be possible.

The success of the attack we found with the Protocol Analyzer relies upon a number of assumptions which may or may not hold in the actual implementation of the protocol. The first of these is that timestamps are of variable length. In the last step, A must be able to confuse a timestamp with a key concatenated with a timestamp. The second assumption is that the initiator of a protocol does not check a timestamp generated by the receiver. Again, this is not specified by Neuman and Stubblebine, but one could imagine cases in which the receiver would want to check a timestamp in order to avoid replying to messages that are obviously out of date. Finally, we must assume that there is no way A can distinquish between keys and timestamps. Thus, for example, there is no field in a message to tell A whether to expect the next field to be a timestamp or a key.

In spite of the fact that it is not likely that a particular implementation will satisfy all these assumptions, knowledge of this attack can be of help in our attempt to gain understanding of how to design a protocol for security. It can tell us which assumptions we should be careful about relaxing for fear of opening up a protocol to attack, and it can tell us which features are relevant to security, and thus should be protected against subversion by a hostile intruder. Thus, for example, any typing mechanism used in an implementation of the Neuman-Stubblebine protocol is relevant to the security of that protocol, and we must be careful to ensure that the mechanism is strong enough so that an intruder cannot cause a message of one type to be passed off as a message of another.

Our analysis of the requirements on the conditions under which the receiver will accept a key turned up another attack, although in this case the attack pointed to a place in which the requirement may be too stringent, rather than a flaw in the protocol itself. It was found that if a compromise event occurs right after the server generates a key, the intruder can cause a receiver B to accept a key as coming from a sender A even though A never requested it: the intruder requests the key while pretending to be A, waits for S to send the key, compromises the key, and then impersonates A to B by proving knowledge of the key in the final step. We note, however, that although such an attack could be prevented, it is probably not worthwhile to do so. In general, protocols are designed to be secure against compromise of keys outside of a given round, not within a round. For example, there is no way to recover against an intruder's compromising a key during a session except to generate a new session key. Thus our discovery of this "attack" shows us that our requirement is too stringent, and it should be modified to one of the following form:

$$\begin{aligned}
&\texttt{rec_accept}(user(B, honest), user(A, honest), K, M2) \wedge \\
&\qquad \neg(\diamondminus(\texttt{compromise}(environment, (), K, M?))) \rightarrow \\
&\quad \diamondminus(\texttt{send}(S, (user(A, honest), user(B, honest)), K, M?) \wedge \\
&\qquad\quad \diamondminus\texttt{request}(user(A, honest), user(B, honest), (), M?))
\end{aligned}$$

6 Conclusion

In this paper we have shown how a requirements language based on temporal logic can be of assistance in the specification and verification of cryptographic protocols. 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. Once the general class of protocol requirements has been identified, it is possible to fine-tune the requirements for the particular application. This is what we have done in this paper. We first gave a general set of requirements for key distribution protocols involving a key server. We then showed how the requirements should be augmented to handle key reauthentication. Finally, we showed how the key reauthentication requirements could be modified to express or leave out the requirement for binding reauthenticated keys to the initial communication, depending whether or not this was needed.

Once we have developed a set of requirements, we can use them together with a formal analysis of a particular protocol both to help us to understand the strengths and weaknesses of the protocol better and to help us improve our understanding of the requirements. In our analysis of the modified Neuman-Stubblebine protocol with the NRL Protocol Analyzer, we were able to make progress in both of these areas. Thus we have provided evidence for the usefulness of our approach.

References

1. Ray Bird, Inder Gopal, Amir Herzberg, Phil Janson, Shay Kutten, Refik Molva, and Moti Yung. Systematic Design of Two-Party Authentication Protocols. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1992.
2. Ulf Carlsen. Using Logics to Detect Implementation-Dependent Flaws. In *Proceedings of the Ninth Annual Computer Security Applications Conference*, pages 64–73. IEEE Computer Society Press, Los Alamitos, California, December 1993.
3. Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes, and Cryptography*, 2:107–125, 1992.
4. D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.
5. Kehne, Schönwälder, and Langendörfer. A Nonce-Based Protocol for Multiple Authentications. *Operating Systems Review*, 26(4):84–89, October 1992.
6. Richard Kemmerer, Catherine Meadows, and Jonathan Millen. Three Systems for Cryptographic Protocol Analysis. *Journal of Cryptology*, 7(2), 1994.
7. C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1:5–53, 1992.
8. B. Clifford Neuman and Stuart G. Stubblebine. A Note on the Use of Timestamps as Nonces. *Operating Systems Review*, 27(2):10–14, April 1993.
9. Einar Snekkenes. Roles in Cryptographic Protocols. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society Press, Los Alamitos, California, 1992.
10. Paul Syverson and Catherine Meadows. A Logical Language for Specifying Cryptographic Protocol Requirements. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE Computer Society Press, Los Alamitos, California, 1993.
11. Paul F. Syverson. Adding Time to a Logic of Authentication. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 97–101. ACM Press, New York, November 1993.
12. Paul F. Syverson. On Key Distribution Protocols for Repeated Authentication. *Operating Systems Review*, 27(4):24–30, October 1993.