Mechanized Proofs for a Recursive Authentication Protocol[1]

Lawrence C. Paulson
Computer Laboratory
University of Cambridge
Pembroke Street
Cambridge CB2 3QG
England

`lcp@cl.cam.ac.uk`

[1]An abridged version of this paper appeared in the Proceedings of the 10th Computer Security Foundations Workshop, 10–12 June 1997, pages 84–95. Copyright © 1997 IEEE

Abstract

A novel protocol has been formally analyzed using the prover Isabelle/HOL, following the inductive approach described in earlier work [11]. There is no limit on the length of a run, the nesting of messages or the number of agents involved. A single run of the protocol delivers session keys for all the agents, allowing neighbours to perform mutual authentication. The basic security theorem states that session keys are correctly delivered to adjacent pairs of honest agents, regardless of whether other agents in the chain are compromised. The protocol's complexity caused some difficulties in the specification and proofs, but its symmetry reduced the number of theorems to prove.

Contents

1 Introduction

Security protocols are notoriously prone to error. One problem is the combinatorial complexity of the messages that an intruder could generate. Another, quite different, problem is that of specifying precisely what the protocol is to accomplish: proof of identity, session key distribution, establishment of shared secrets, etc.

Researchers are developing methods of analyzing protocols formally—either to search for attacks [6] or to prove correctness properties [3, 5, 8]. Recently, I have announced a new proof method, based on inductive models of protocols and an intruder [11]. Unlike model-checking approaches, it imposes no restrictions for the sake of finiteness. The automated provers in Isabelle/HOL [10] let the user analyze a typical protocol in a few working days. Below I shall describe an application of the method to an unusual, variable-length protocol.

The Otway-Rees protocol [9], which assumes a shared-key setting, allows an agent A to establish a session with some other agent, B. An authentication server generates a fresh session key Kab and distributes it to A and B. This protocol is widely accepted as correct; in previous work, I have proved basic guarantees for each party. At the end of a run, the session key just received is known only to the server and the other party, provided both A and B are uncompromised. Like many similar protocols, it has a fixed number of steps and all messages have simple, fixed formats.

The *recursive authentication protocol* [4] generalizes Otway-Rees to an arbitrary number of parties. First, A contacts B. If B then contacts the authentication server then the run resembles Otway-Rees. But B may choose to contact some other agent C, and so forth; a chain of arbitrary length may form. During each such round, an agent adds its name and a fresh nonce to an ever-growing request message.

For the sake of discussion, suppose that C does not extend the chain but instead contacts the authentication server. The server generates fresh session keys Kab and Kbc: in the general case, one key for each pair of agents adjacent in the chain. It prepares two certificates for each session key: one for each party. It gives the bundle of certificates to the last agent (C). Each agent takes two certificates and forwards the remainder to its predecessor in the chain. Finally, A receives one certificate, containing Kab.

Such a protocol is hard to specify, let alone analyze. Neither the number of steps, nor the number of parties, nor the number of session keys are fixed in advance. The server's response to the agents' accumulated requests cannot be given as a simple pattern; it requires a recursive program.

Even which properties to prove are not obvious. One might simplify the protocol to distribute a single session key, common to all the agents in the chain. But then, security between A and B would depend upon the honesty of C, an agent possibly not known to A. There may be applications where

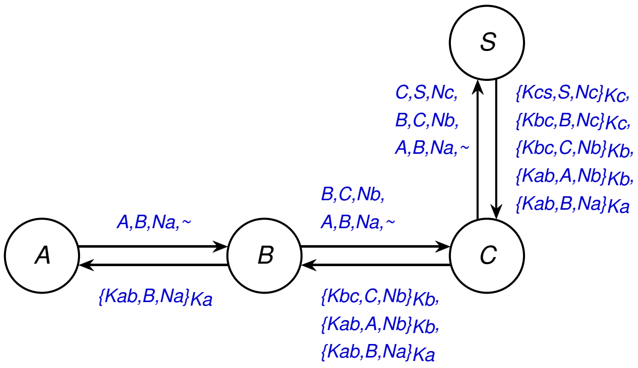

Figure 1: The Recursive Authentication Protocol with Three Clients

such a weak guarantee might be acceptable, but it seems better to give a separate session key to each adjacent pair. I have proved a general guarantee for each participant: if it receives a certificate containing a session key and the name of another agent, then only that agent (and the server) can know the key. (In the sequel, "server" will always mean "authentication server.")

The paper describes the protocol in detail (§2). It reviews the inductive approach to protocol analysis (§3) and describes how it was extended with hashing (§4). It presents the formal model of the protocol (§5) and describes the main results proved (§6). It discusses possible attacks on the protocol (§7) and offers a few conclusions (§8).

2 The Recursive Authentication Protocol

The protocol was invented by John Bull of APM Ltd., who (in a private communication) describes its objectives as follows:

> The project is exploring a model of security where application components are in control of security policy and its enforcement. The novelty of the approach is that the infrastructure no longer provides security for the applications, but provides them with the means to defend themselves.

The description below uses traditional notation, slightly modified. Let $\mathsf{Hash}\, X$ be the hash of X and $\mathsf{Hash}_X\, Y$ the pair $\{\mathsf{Hash}\{X, Y\}, Y\}$. Typically, X is an agent's long-term shared key and $\mathsf{Hash}\{X, Y\}$ is a message digest, enabling the server to check that Y originated with that agent. Figure 1 shows a typical run, omitting the hashing.

Agent A starts a run by sending B a request:

$$1.\ A \to B : \mathsf{Hash}_{Ka}\{A, B, Na, -\}$$

Here Ka is A's long-term shared key, Na is a fresh nonce, and $(-)$ is a placeholder indicating that this message started the run. In response, B sends something similar but with A's message instead of the placeholder:

$$2.\ B \to C : \mathsf{Hash}_{Kb}\{|B, C, Nb, \mathsf{Hash}_{Ka}\{|A, B, Na, -|\}|\}$$

Step 2 may be repeated as many times as desired. Each time, new components are added to the message and a new message digest is prefixed. The recursion terminates when some agent performs step 2 with the server as the destination.

In step 3, the server prepares session keys for each caller-callee pair. It traverses the accumulated requests to build up its response. If (as in §1) the callers were A, B and C in that order, then the final request is

$$\mathsf{Hash}_{Kc}\{|\underset{\uparrow}{C}, S, Nc, \mathsf{Hash}_{Kb}\{|B, \underset{\uparrow}{C}, Nb, \mathsf{Hash}_{Ka}\{|A, B, Na, -|\}|\}|\}.$$

The arrows point to the occurrences of C, which appear in the outer two levels. C has called S (the server) and was called by B. The server generates session keys Kcs and Kbc and prepares the certificates $\{|Kcs, S, Nc|\}_{Kc}$ and $\{|Kbc, B, Nc|\}_{Kc}$. The session key Kcs is redundant because C already shares Kc with the server. Including it allows the last agent in the chain to be treated like all other agents except the first: the initiator receives only one session key.

Having dealt with C's request, the server discards it. Looking at the remaining outer two levels, the request message is

$$\mathsf{Hash}_{Kb}\{|\underset{\uparrow}{B}, C, Nb, \mathsf{Hash}_{Ka}\{|A, \underset{\uparrow}{B}, Na, -|\}|\}.$$

The server now prepares two certificates for B, namely $\{|Kbc, C, Nb|\}_{Kb}$ and $\{|Kab, A, Nb|\}_{Kb}$. Note that Kbc appears in two certificates, one intended for C (containing nonce Nc and encrypted with key Kc) and one for B.

At the last iteration, the request message contains only one level:

$$\mathsf{Hash}_{Ka}\{|\underset{\uparrow}{A}, B, Na, -|\}.$$

The $(-)$ token indicates the end of the requests. The server generates one session key and certificate for A, namely $\{|Kab, B, Na|\}_{Ka}$.

In step 3 of the protocol, the server replies to the request message by returning a bundle of certificates. In our example, it would return five certificates to C.

$$\begin{aligned} 3.\ S \to C :\ & \{|Kcs, S, Nc|\}_{Kc}, \{|Kbc, B, Nc|\}_{Kc}, \\ & \{|Kbc, C, Nb|\}_{Kb}, \{|Kab, A, Nb|\}_{Kb}, \\ & \{|Kab, B, Na|\}_{Ka} \end{aligned}$$

In step 4, an agent accepts the first two certificates and forwards the rest to its predecessor in the chain. Every agent performs this step except the one who started the run.

$$\begin{aligned} 4.\ C \to B &: \{Kbc, C, Nb\}_{Kb}, \{Kab, A, Nb\}_{Kb}, \\ &\quad \{Kab, B, Na\}_{Ka} \\ 4'.\ B \to A &: \{Kab, B, Na\}_{Ka} \end{aligned}$$

The description above describes a special case: a protocol run with three clients. The conventional protocol notation cannot cope with arbitrary numbers of participants, let alone recursive processing of nested messages. Section §5 below will specify the protocol as an inductive definition.

My version of the protocol differs from the original in several respects:

- The dummy session key Kcs avoids having to treat the last agent as a special case. All agents except the first take two certificates. An implementation can safely omit the dummy certificate. Removing information from the system makes less information available to an intruder.

- In the original protocol, an agent's two certificates were distinguished only by their order of arrival; an intruder could easily exchange them. To correct this flaw, I added the other party's name to each certificate. Such explicitness is recommended as good engineering practice [2]. It also simplifies the proofs; a similar change to the Otway-Rees protocol cut the proofs in half [11]. Bull and Otway have accepted my change to their protocol [4].

- The original protocol implements encryption using exclusive "or" (XOR) and hashing. For verification purposes, encryption should be taken as primitive. Correctness of the protocol does not depend upon the precise form of encryption, provided it implemented properly. The original use of XOR turned out to be flawed (see §7).

- Protocol messages contain some information that is important for engineering purposes but logically redundant. Omitting such information both simplifies and strengthens the proofs. Adding redundancy to a safe protocol cannot make it unsafe.

3 Review of the Inductive Approach

Informal safety arguments, which involve reasoning that dangerous states are unreachable, are made rigorous using induction. Protocols are modelled in standard predicate calculus and set theory.[1] A protocol specifies a set of

[1] I have used higher-order logic as a typed set theory. An untyped approach, based perhaps upon ZF set theory, is also feasible.

possible traces. Each trace is a list of events and may contain numerous runs, including interleaved and aborted runs. Each event has the form $\mathsf{Says}\, A\, B\, X$, which represents the attempt by A to send B the message X.

One simplification in the model is the lack of an event to represent the reception of a message. We cannot assume that each message reaches its destination; we cannot identify the sending of a message in step i with the receipt of that message. But we can identify the sending of a message in step $i+1$ with the receipt of a satisfactory message from step i. The model describes what may happen but never forces an agent to respond to any message. Thus, the model identifies the following circumstances:

- Message X was intercepted before it could reach B.
- Message X reached B, but B was down.
- Message X reached B, but B declined to respond.

An inductive definition consists of a set of rules. Modelling a protocol requires one rule for each protocol step. A typical protocol step $A \to B : X$ requires a rule saying that an existing trace can be extended with the event $\mathsf{Says}\, A\, B\, X$. If the step is the response to another message Y, then the rule will be subject to the condition $\mathsf{Says}\, B'\, A\, Y$. Other conditions may refer to messages that A has already sent earlier in the same run, typically for the purpose of confirming that a challenge has triggered an adequate response.

Agent names such as A and B are variables ranging over all agents, though rules frequently have conditions such as $A \neq B$ or $A \neq \mathsf{S}$.

An additional rule models the attacker, Spy. He cannot crack ciphers by brute force, but has somehow got hold of some agents' long-term keys. He reads all traffic, decrypting messages using keys he holds and accumulating everything so obtained. At any point, he may send spoof messages composed using the data at his disposal. (Interception of messages does not have to be modelled explicitly, as remarked above.) He is accepted by the others as an honest agent.

To model the spy's capabilities, three operators are defined on sets of messages.

- $\mathsf{parts}\, H$ is the set of all components of H that are potentially recoverable, perhaps using additional keys.
- $\mathsf{analz}\, H$ is the set of all messages that can be decrypted from H using only keys obtainable from H.
- $\mathsf{synth}\, H$ is the set of all messages that can be built up from H.

These operators are themselves defined inductively and satisfy numerous useful laws. The spy draws spoof messages from the set $\mathsf{synth}(\mathsf{analz}\, H)$, where H includes the history of past traffic and the spy's initial knowledge.

4 A Formalization of Hashing

Hitherto, I have considered messages to be built from agent names, nonces and keys by concatenation and encryption. The recursive authentication protocol assures integrity by means of hashing. Encryption could be used instead, but hashing is easily added to the model.

The datatype msg now admits messages of the form $\mathsf{Hash}\,X$, where X is a message. Like all the message primitives, Hash is assumed to be injective (collision-free). The operators analz and synth treat hashing in the obvious way. The definition of analz requires no change; the effect is to say that nothing can be decrypted from $\mathsf{Hash}\,X$. For synth, we insert the rule

$$X \in \mathsf{synth}\,H \Longrightarrow \mathsf{Hash}\,X \in \mathsf{synth}\,H,$$

allowing hashing to be used freely when composing spoof messages.

A question arises concerning the treatment of hashing by parts: is X a part of $\mathsf{Hash}\,X$? The protocol involves messages of the form $\mathsf{Hash}\{\!|Ka, X'|\!\}$. A *yes* answer would imply $Ka \in \mathsf{parts}\,H$, even if Ka were uncompromised; we could no longer reason about the security of long-term keys in the normal way [11]. A *no* answer seems right. It causes $\mathsf{parts}\,H$ to return the items *potentially recoverable* from H, which is a subset of the *ingredients* of H. Contrast the tasks of recovering the mainspring from a watch and recovering the eggs from a cake. An "ingredients" operator is not needed just now.

The new laws concerning $\mathsf{Hash}\,X$ resemble those for other atomic data, such as nonces. Here, $\mathsf{ins}\,X\,H$ is the set $\{X\} \cup H$.

$$\begin{aligned}
\mathsf{parts}(\mathsf{ins}(\mathsf{Hash}\,X)\,H) &= \mathsf{ins}(\mathsf{Hash}\,X)(\mathsf{parts}\,H)\\
\mathsf{analz}(\mathsf{ins}(\mathsf{Hash}\,X)\,H) &= \mathsf{ins}(\mathsf{Hash}\,X)(\mathsf{analz}\,H)\\
\mathsf{Hash}\,X \in \mathsf{synth}\,H &\Longrightarrow \mathsf{Hash}\,X \in H \lor X \in \mathsf{synth}\,H
\end{aligned}$$

These laws state that $\mathsf{Hash}\,X$ contributes nothing other than itself to the result of parts or analz. The addition of Hash as a new message form does not invalidate any of the 90 or so laws concerning parts, analz and synth.

The $\mathsf{Hash}_X\,Y$ notation is trivially defined in Isabelle:

$$\mathsf{Hash}[X]Y \equiv \{\!|\mathsf{Hash}\{\!|X, Y|\!\}, Y|\!\}.$$

Requests in the protocol have the form $\mathsf{Hash}_X\,Y$, where Y may contain another request. Rewriting a request by the definition of $\mathsf{Hash}_X\,Y$ would cause exponential blowup. Instead, we can apply laws that treat $\mathsf{Hash}_X\,Y$ as a primitive. The following law concerns parts; an analogous one holds for analz:

$$\mathsf{parts}(\mathsf{ins}(\mathsf{Hash}_X\,Y)\,H) = \mathsf{ins}(\mathsf{Hash}_X\,Y)(\mathsf{ins}(\mathsf{Hash}\{\!|X, Y|\!\})(\mathsf{parts}(\mathsf{ins}\,Y\,H)))$$

A further law is subject to $X \notin \mathsf{synth}(\mathsf{analz}\, H)$, as when X is an uncompromised long-term key:

$$\mathsf{Hash}_X\, Y \in \mathsf{synth}(\mathsf{analz}\, H) \iff \mathsf{Hash}\{\!|X, Y|\!\} \in \mathsf{analz}\, H \land Y \in \mathsf{synth}(\mathsf{analz}\, H)$$

This law says that the message $\mathsf{Hash}_X\, Y$ can be spoofed iff Y can be and a suitable message digest is available (an unlikely circumstance). Blowup can still occur using such laws, but it is no longer inevitable. A formula such as $\mathsf{Nonce}\, N \in \mathsf{parts}\, H$ will simplify to the obvious outcome.

5 Modelling the Protocol

For the most part, this protocol is modelled just like the fixed-length protocols considered previously [11]. The inductive definition rules for the empty trace and the spy are standard. The other rules can be paraphrased as follows:

1. If evs is a trace, Na is a fresh nonce and B is an agent distinct from A and S, then we may add the event

   $$\mathsf{Says}\, A\, B\, (\mathsf{Hash}_{\mathsf{shrK}\, A}\{\!|A, B, Na, -|\!\}).$$

   A's long-term key is written $\mathsf{shrK}\, A$. For the token $(-)$ I used the name S, but any fixed message would do as well.

2. If evs is a trace containing the event $\mathsf{Says}\, A'\, B\, Pa$, where $Pa = \{\!|Xa, A, B, Na, P|\!\}$, and Nb is a fresh nonce and $B \neq C$, then we may add the event

   $$\mathsf{Says}\, B\, C\, (\mathsf{Hash}_{\mathsf{shrK}\, B}\{\!|B, C, Nb, Pa|\!\}).$$

   The variable Xa is how B sees A's hash value; he does not have the key needed to verify it. Component P might be $(-)$ (if A started the run) or might have the same form as Pa, nested to any depth. Agent C might be the server or anybody else.

   All the proofs about the protocol become simpler if the equation $Pa = \cdots$ is never applied. The proofs therefore hold of a weaker protocol in which any agent may react to any message by sending an instance of step 2. Ill-formed requests may result, but the server will ignore them.

3. If evs is a trace containing the event $\mathsf{Says}\, B'\, \mathsf{S}\, Pb$, and $B \neq \mathsf{S}$, and if the server can build from request Pb a response Rb, then we may add the event

   $$\mathsf{Says}\, \mathsf{S}\, B\, Rb.$$

The construction of Rb includes verifying the integrity of Pb; this process is itself defined inductively, as we shall see. The rule does not constrain the agent B, allowing the server to send the response to anybody. We could get the right value of B from Pb, but the proofs do not require such details.

4. If evs is a trace containing the two events

$$\begin{aligned}&\mathsf{Says}\,B\,C\,(\mathsf{Hash}_{\mathsf{shrK}\,B}\{\!|B, C, Nb, Pa|\!\})\\ &\mathsf{Says}\,C'\,B\,\{\!|\mathsf{Crypt}(\mathsf{shrK}\,B)\{\!|Kbc, C, Nb|\!\},\\ &\qquad\qquad \mathsf{Crypt}(\mathsf{shrK}\,B)\{\!|Kab, A, Nb|\!\},\ R|\!\}\end{aligned}$$

and $A \neq B$, then we may add the event

$$\mathsf{Says}\,B\,A\,R.$$

B decrypts the two certificates, compares their nonces with the value of Nb he used, and forwards the remaining certificates (R).

The final step of the protocol is the initiator's acceptance of the last certificate, $\mathsf{Crypt}(\mathsf{shrK}\,A)\{\!|Kab, B, Na|\!\}$. This step need not be modelled since A makes no response.

For many protocols, an "oops" message can model accidental loss of session keys. One then proves that an old, compromised session key cannot later become associated with new nonces [11]. An oops message cannot easily be expressed for the recursive authentication protocol because a key never appears together with both its nonces. Despite the lack of an oops message, the spy can get hold of session keys using the long-term keys of compromised agents; I trust the model is adequately realistic.

5.1 Modelling the Server

The server creates the list of certificates according to another inductive definition. It defines not a set of traces but a set of triples (P, R, K) where P is a request, R is a response and K is a session key. Such triples belong to the set $\mathsf{respond}\,evs$, where evs (the current trace) is supplied to prevent the reuse of old session keys. Component K returns the newest session key to the caller for inclusion in a second certificate.

The occurrences of Hash in the definition ensure that the server accepts requests only if he can verify the hashes using his knowledge of the long-term keys. The inductive definition consists of two cases.

1. If Kab is a fresh key (that is, not used in evs) then

$$\begin{aligned}&(\,\mathsf{Hash}_{\mathsf{shrK}\,A}\{\!|A, B, Na, -|\!\},\\ &\ \ \mathsf{Crypt}(\mathsf{shrK}\,A)\{\!|Kab, B, Na|\!\},\\ &\ \ Kab) \quad \in \quad \mathsf{respond}\,evs.\end{aligned}$$

This base case handles the end of the request list, where A seeks a session key with B.

2. If $(Pa, Ra, Kab) \in \mathsf{respond}\, evs$ and Kbc is fresh (not used in evs or Ra) and

$$Pa = \mathsf{Hash}_{\mathsf{shrK}\, A}\{\!|A, B, Na, P|\!\}$$

then

$$\begin{aligned}
&(\,\mathsf{Hash}_{\mathsf{shrK}\, B}\{\!|B, C, Nb, Pa|\!\},\\
&\ \{\!|\mathsf{Crypt}(\mathsf{shrK}\, B)\{\!|Kbc, C, Nb|\!\},\\
&\quad \mathsf{Crypt}(\mathsf{shrK}\, B)\{\!|Kab, A, Nb|\!\},\ Ra|\!\},\\
&\ Kbc) \quad \in \quad \mathsf{respond}\, evs.
\end{aligned}$$

The recursive case handles a request list where B seeks a session key with C and has himself been contacted by A. The respond relation is best understood as a pure Prolog program. Argument Pa of (Pa, Ra, Kab) is the input, while Ra and Kab are outputs. Key Kab has been included in the response Ra and must be included in one of B's certificates too.

An inductive definition can serve as a logic program. Because the concept is Turing powerful, it can express the most complex behaviours. Yet, such programs are easy to reason about.

5.2 A Coarser Model of the Server

For some purposes, the $\mathsf{respond}\, evs$ relation is needlessly complicated. Its input is a list of n requests, for $n > 0$, and its output is a list of $2n + 1$ certificates. Many routine lemmas hold for any list of certificates of the form $\mathsf{Crypt}(\mathsf{shrK}\, B)\{\!|K, A, N|\!\}$. The inductive relation $\mathsf{responses}\, evs$ generates the set of all such lists. It contains all possible server responses and many impossible ones.

The base case is simply $(-) \in \mathsf{responses}\, evs$ and the recursive case is

$$\{\!|\mathsf{Crypt}(\mathsf{shrK}\, B)\{\!|K, A, N|\!\},\ R|\!\} \in \mathsf{responses}\, evs$$

if $R \in \mathsf{responses}\, evs$ and K is not used in evs.

In secrecy theorems (those expressed in terms of analz), each occurrence of Crypt can cause a case split, resulting in a substantial blowup after simplification. Induction over responses introduces only one Crypt, but induction over respond introduces three. Of course, responses includes some invalid outputs; some of the main theorems can only be proved for respond.

5.3 Alternative Formats for Certificates

In the original protocol, B took two certificates of the form $\mathsf{Crypt}(\mathsf{shrK}\,B)\{\!|K, Nb|\!\}$. He inferred the name of the other key-holder from the certificates' arrival order, which an attacker could change. This vulnerability had to be corrected. I experimented with putting both session keys into a single certificate of the form

$$\mathsf{Crypt}(\mathsf{shrK}\,B)\{\!|Kab, Kbc, Nb|\!\}.$$

An enemy can no longer exchange the keys. But each session key still appears in two certificates; Kbc would also appear in

$$\mathsf{Crypt}(\mathsf{shrK}\,C)\{\!|Kbc, Kcd, Nc|\!\}.$$

This format would have required two sets of proofs: one concerning the first session key in a certificate and one concerning the second. I therefore took Abadi and Needham's advice and chose certificates of the form

$$\mathsf{Crypt}(\mathsf{shrK}\,A)\{\!|Kab, B, Na|\!\}.$$

Now, permuting the certificates can do no harm: each quotes the name of the other holder of the session key. Moreover, this format preserves the protocol's symmetry.

6 Main Results Proved

For the most part, the analysis resembles that of the Otway-Rees protocol. Simple consistency checks, or "possibility properties," are proved first. Regularity lemmas come next: elementary facts such as that the long-term keys of uncompromised agents never form part of any message. (They do form part of hashed messages, however; recall the discussion in §4 above.) Secrecy theorems govern the use of session keys, leading to the main guarantee: if the certificate $\mathsf{Crypt}(\mathsf{shrK}\,A)\{\!|Kab, B, Na|\!\}$ appears as part of any traffic, where A and B are uncompromised, then Kab will never reach unintended parties. Another theorem guarantees that such certificates (for uncompromised agents) originate only with the server.

Possibility properties do not express liveness, but merely that the protocol can sometimes run to completion. They imply basic properties, such as that message formats are compatible from one step to the next. Though logically trivial, their machine proofs require significant effort (or computation) due to the complexity of the terms that arise and the number of choices that can be made at each step. I proved cases corresponding to runs with up to three agents plus the server and spy. General theorems for n agents could be proved by induction on n, but the necessary effort hardly seems justified.

Security properties are proved, as always, by induction over the protocol definition. For this protocol, the main inductive set (recur) is defined in terms of another (respond). All but the most trivial proofs require induction over both definitions. The inner induction over respond might be proved as a preliminary lemma if it is used more than once or if its proof is complicated.

The set responses specifies the general form of the outputs generated by respond:

```
(PA,RB,K): respond evs ==> RB: responses evs
```

This easily-proved result justifies performing induction over responses instead of respond; if it leads to a proof at all, it will lead to a simpler proof.

Elementary results are no harder to prove than for a fixed-length protocol. The outer induction yields six subgoals: one for each protocol step, plus the base and fake cases. The inner induction replaces the step 3 case by two subcases, namely the server's base case and inductive step. There are thus seven subgoals, few of which typically survive simplification. Only the theorems described below have difficult proofs.

Nonces generated in requests are unique. This theorem states that there can be at most one hashed value containing the key of an uncompromised agent ($A \notin lost$) and any specified nonce value, Na.[2]

```
∃ B' P'. ∀ B P.
  Hash {|Key(shrK A), Agent A, Agent B, Na, P|}
    ∈ parts (sees lost Spy evs)
  ⟶  B=B' ∧ P=P'
```

Although it is not used in later proofs, unicity of nonces is important. It lets agents identify runs by their nonces. The theorem applies to all requests, whether generated in step 1 or step 2. For the Otway-Rees protocol, each of the two steps requires its own theorem. The reasoning here is similar, but one theorem does the work of two, thanks to the protocol's symmetry. The nesting of requests does not affect the reasoning.

Secrecy proofs require the lemma

```
(Key K ∈ analz (ins(Key Kab)(sees lost Spy evs)))
=
(K=Kab ∨ Key K ∈ analz (sees lost Spy evs))
```

where Kab is a session key. (Equality between formulae denotes if-and-only-if.) Rewriting using it extracts session keys from the argument of analz. The lemma may be paraphrased as saying that existing session keys cannot be used to learn new ones [11]. It is hard to prove. For the induction to go through, it must be generalized to an arbitrary set of session keys:

[2]The set sees $lost$ Spy evs consists of everything the spy can see: all the messages in the trace evs and the long-term keys of all the agents in the set $lost$. This particular theorem is not concerned with the spy but with possible message histories.

```
∀ K KK. KK ⊆ Compl (range shrK) ⟶
  (Key K ∈ analz (Key``KK ∪ (sees lost Spy evs)))
  =
  (K ∈ KK ∨ Key K ∈ analz (sees lost Spy evs))
```

The inner induction over respond leads to excessive case splits. It was to simplify this proof that I defined the set responses.

Unicity for session keys is unusually complicated because each key appears in two certificates. Moreover, the certificates are created in different iterations of respond. The unicity theorem states that, for any K, if there is a certificate of the form

$$\mathsf{Crypt}(\mathsf{shrK}\,A)\{\!|K, B, Na|\!\}$$

(where A and B are uncompromised) then the only other certificate containing K must have the form

$$\mathsf{Crypt}(\mathsf{shrK}\,B)\{\!|K, A, Nb|\!\},$$

for some Nb. If $(PB, RB, K) \in \mathsf{respond}\, evs$ then

```
∃ A' B'. ∀ A B N.
  Crypt (shrK A) {|Key K,Agent B,N|} ∈ parts{RB}
  ⟶  (A'=A ∧ B'=B) ∨ (A'=B ∧ B'=A)
```

This theorem seems quite strong. An agent who receives a certificate immediately learns which other agent can receive its mate, subject to the security of both agents' long-term keys. One might hope that security of session keys would follow without further ado. Informally, we might argue that the only messages containing session keys contain them as part of such certificates, and thus the keys are safe from the spy. But such reasoning amounts to another induction over all possible messages in the protocol. The theorem must be stated (stipulating A, $A' \notin lost$) and proved:

```
Crypt (shrK A) {|Key K, Agent A', N|}
  ∈ parts (sees lost Spy evs)
⟶ Key K ∉ analz (sees lost Spy evs)
```

The induction is largely straightforward except for the step 3 case. The inner induction over respond leads to such complications that it must be proved beforehand as a lemma. If $(PB, RB, Kab) \in \mathsf{respond}\, evs$ then

```
∀ A A' N. A ∉ lost ∧ A' ∉ lost
  ⟶
  Crypt (shrK A) {|Key K,Agent A',N|} ∈ parts{RB}
  ⟶
  Key K ∉ analz (ins RB (sees lost Spy evs))
```

A slightly stronger result is that the key enclosed in a certificate does not reach any unintended agents, even honest ones.

Although each session key appears in two certificates, they both have the same format. A single set of proofs applies to all certificates. The protocol's symmetry halves the effort compared with Otway-Rees, which requires two sets of proofs.

7 Potential Attacks

One must never assume that a "verified" protocol is infallible. The proofs are subject to the assumptions implicit in the model. Attacks against the protocol or implementations of it can still be expected. One "attack" is quite obvious: in step 2, agent B does not know whether A's message is recent; at the conclusion of the run, B still has no evidence that A is present. The spy can masquerade as A by replaying an old message of hers, but cannot read the resulting certificate without her long-term key.

Allowing type confusion (such as passing a nonce as a key) often admits attacks [7, 8] in which one encrypted message is mistaken for another one that is intended for another purpose. The recursive authentication protocol appears to be safe from such attacks: it has only one form of encrypted message, with only one purpose. However, the implementation of encryption must be secure.

The original protocol description suggested an unsound form of encryption. Each session key was encrypted by forming its exclusive "or" with a hash value, used as a one-time pad. Unfortunately, each hash value was used twice: B's session keys Kab and Kbc were encrypted as

$$Kab \oplus \mathsf{Hash}\{\!|Kb, Nb|\!\} \quad \text{and} \quad Kbc \oplus \mathsf{Hash}\{\!|Kb, Nb|\!\}.$$

By forming their exclusive "or", an eavesdropper could immediately obtain $Kab \oplus Kbc$, $Kbc \oplus Kcd$, etc. Compromise of any one session key would allow all the others to be read off. Using say $\mathsf{Hash}\{\!|Kb, Nb+1|\!\}$ to encrypt the second key would be secure, assuming a good hash function.

This attack (found by Peter Ryan and Steve Schneider) is a valuable reminder of the limitations of formal proofs. It does not contradict the proofs, which regard encryption as primitive. The inductive approach can probably be extended to cope with exclusive "or". But such a low-level description violates the principle of *separation of concerns*. It requires longer proofs and can only be recommended if the protocol specifically requires one form of encryption. Perhaps we need formal tools to allow implementations of encryption to be proved correct independently of the protocols using them.

8 Conclusions

Analyzing this protocol took about two weeks of normal working days, including the formalization of hashing and experimentation with different formats for certificates. Streamlining and maintaining the proofs has taken additional time. The proofs are modest in scale: fewer than 30 results are proved, using under 135 tactic commands; they run in less than five minutes.

The inductive approach readily models this complex protocol. It is perfectly suited to Isabelle/HOL's inductive definition package, simplifier and classical reasoner. Proofs can be generated using modest human and computational resources. The approach does not search for attacks, but establishes positive guarantees. In the present case, it has suggested ways of strengthening the protocol by adding explicitness. Bull has suggested that the protocol might be modified to distribute session keys between agents that are not adjacent in the request chain. Such variants can probably be analyzed with little difficulty by modifying the existing proof script.

Acknowledgement Many thanks to John Bull and Dave Otway for explaining their protocol to me. Discussions with A. Gordon, F. Massacci, R. Needham, P. Ryan and K. Wagner were helpful. Giampaolo Bella and the referees commented on a draft. The research was funded by the EPSRC, grant GR/K77051 "Authentication Logics", and by the ESPRIT working group 21900 "Types".

References

[1] 10th computer security foundations workshop. In *10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[2] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[3] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *Third ACM Conference on Computer and Communications Security*, pages 106–118. ACM Press, 1996.

[4] John A. Bull and David J. Otway. The authentication protocol. Technical Report DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/03, Defence Research Agency, Malvern, UK, 1997. In press.

[5] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, 426:233–271, 1989.

[6] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems. Second International Workshop, TACAS '96*, LNCS 1055, pages 147–166, 1996.

[7] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *10th Computer Security Foundations Workshop* [1]. In press.

[8] Catherine A. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Computer Security — ESORICS 96*, LNCS 1146, pages 351–364. Springer, 1996.

[9] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, January 1987.

[10] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[11] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th Computer Security Foundations Workshop* [1]. In press.

A Isabelle Specifications and Theorems

Isabelle users will understand this material best, but I hope all readers will gain an impression of the notation. Figure 2 defines the relation respond evs, which specifies the server's interpretation of requests. The set responses evs is omitted because it does not form part of the specification itself (it merely simplifies some of the proofs). Figure 3 specifies the protocol itself. I have omitted some comments.

```
consts     respond :: "event list => (msg*msg*key)set"
inductive "respond evs"
  intrs
    (*The message "Agent Server" marks the end of a list.*)
    One  "[| A ≠ Server;  Key KAB ∉ used evs |]
          ⟹ (Hash[Key(shrK A)]
                        {|Agent A, Agent B, Nonce NA, Agent Server|},
               {|Crypt(shrK A){|Key KAB,Agent B,Nonce NA|}, Agent Server|},
               KAB)   ∈ respond evs"

    (*The most recent session key is passed up to the caller*)
    Cons "[| (PA, RA, KAB) ∈ respond evs;
             Key KBC ∉ used evs;  Key KBC ∉ parts {RA};
             PA = Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, P|};
             B ≠ Server |]
          ⟹ (Hash[Key(shrK B)] {|Agent B, Agent C, Nonce NB, PA|},
               {|Crypt (shrK B) {|Key KBC, Agent C, Nonce NB|},
                 Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
                 RA|},
               KBC)   ∈ respond evs"
```

Figure 2: Specifying the Server

Figure 4 presents the formal statements of the main theorems of §6, in close to raw Isabelle syntax. The proof scripts are omitted because they are unintelligible. A proof requires four commands on average, of which at least two are quite predictable: induction and simplification.

```
consts     recur   :: agent set => event list set
inductive "recur lost"
  intrs
         (*Initial trace is empty*)
    Nil  "[] ∈ recur lost"

         (*The spy MAY say anything he CAN say.*)
    Fake "[| evs ∈ recur lost;  B≠Spy;
             X ∈ synth (analz (sees lost Spy evs)) |]
          ⟹ Says Spy B X  # evs ∈ recur lost"

         (*Alice initiates a protocol run.*)
    RA1  "[| evs ∈ recur lost;  A≠B;  A≠Server;  Nonce NA ∉ used evs |]
          ⟹ Says A B
                (Hash[Key(shrK A)]
                 {|Agent A, Agent B, Nonce NA, Agent Server|})
              # evs ∈ recur lost"

         (*Bob's response to Alice's message.  C might be the Server.*)
    RA2  "[| evs ∈ recur lost;  B≠C;  B≠Server;  Nonce NB ∉ used evs;
             Says A' B PA ∈ set_of_list evs;
             PA = {|XA, Agent A, Agent B, Nonce NA, P|} |]
          ⟹ Says B C (Hash[Key(shrK B)]{|Agent B, Agent C, Nonce NB, PA|})
              # evs ∈ recur lost"

         (*The Server receives Bob's message and prepares a response.*)
    RA3  "[| evs ∈ recur lost;  B≠Server;
             Says B' Server PB ∈ set_of_list evs;
             (PB,RB,K) ∈ respond evs |]
          ⟹ Says Server B RB # evs ∈ recur lost"

         (*Bob receives the returned message and compares the Nonces with
           those in the message he previously sent the Server.*)
    RA4  "[| evs ∈ recur lost;  A≠B;
             Says C' B {|Crypt (shrK B) {|Key KBC, Agent C, Nonce NB|},
                         Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
                         RA|}
               ∈ set_of_list evs;
             Says B  C {|XH, Agent B, Agent C, Nonce NB,
                         XA, Agent A, Agent B, Nonce NA, P|}
               ∈ set_of_list evs |]
          ⟹ Says B A RA # evs ∈ recur lost"
```

Figure 3: Specifying the Protocol

```
(PA,RB,KAB) ∈ respond evs ⟹ RB ∈ responses evs


[| evs ∈ recur lost; A ∉ lost |]
⟹ ∃ B' P'. ∀ B P.
        Hash {|Key(shrK A), Agent A, B, NA, P|}
          ∈ parts (sees lost Spy evs)  ⟶  B=B' ∧ P=P'


evs ∈ recur lost ⟹
∀ K KK. KK <= Compl (range shrK) ⟶
         (Key K ∈ analz (Key``KK Un (sees lost Spy evs))) =
         (K ∈ KK ∨ Key K ∈ analz (sees lost Spy evs))


[| evs ∈ recur lost;  KAB ∉ range shrK |] ⟹
        Key K ∈ analz (ins (Key KAB) (sees lost Spy evs)) =
        (K = KAB ∨ Key K ∈ analz (sees lost Spy evs))


(PB,RB,KXY) ∈ respond evs
⟹ ∃ A' B'. ∀ A B N.
              Crypt (shrK A) {|Key K, Agent B, N|} ∈ parts {RB}
              ⟶   (A'=A ∧ B'=B) ∨ (A'=B ∧ B'=A)


[| (PB,RB,KAB) ∈ respond evs;  evs ∈ recur lost |]
⟹ ∀ A A' N. A ∉ lost ∧ A' ∉ lost ⟶
              Crypt (shrK A) {|Key K, Agent A', N|} ∈ parts{RB} ⟶
              Key K ∉ analz (ins RB (sees lost Spy evs))


[| Crypt (shrK A) {|Key K, Agent A', N|} ∈ parts (sees lost Spy evs);
   A ∉ lost;  A' ∉ lost;  evs ∈ recur lost |]
⟹ Key K ∉ analz (sees lost Spy evs)
```

Figure 4: Some Theorems in Isabelle's Syntax