Proving Security Protocols Correct

Lawrence C. Paulson

Computer Laboratory
University of Cambridge
Cambridge CB2 3QG
England
E-mail: lcp@cl.cam.ac.uk

Abstract

Security protocols use cryptography to set up private communication channels on an insecure network. Many protocols contain flaws, and because security goals are seldom specified in detail, we cannot be certain what constitutes a flaw. Thanks to recent work by a number of researchers, security protocols can now be analyzed formally.

The paper outlines the problem area, emphasizing the notion of freshness. It describes how a protocol can be specified using operational semantics and properties proved by rule induction, with machine support from the proof tool Isabelle. The main example compares two versions of the Yahalom protocol. Unless the model of the environment is sufficiently detailed, it cannot distinguish the correct protocol from a flawed version.

The paper attempts to draw some general lessons on the use of formalisms. Compared with model checking, the inductive method performs a finer analysis, but the cost of using it is greater.

1 Introduction

Computer networks are vulnerable. Suppose that Monica sends an e-mail to her friend Bill. Since her message must pass through other computers, other people can read or alter it (Fig. 1). They can send bogus messages that appear to come from Bill.

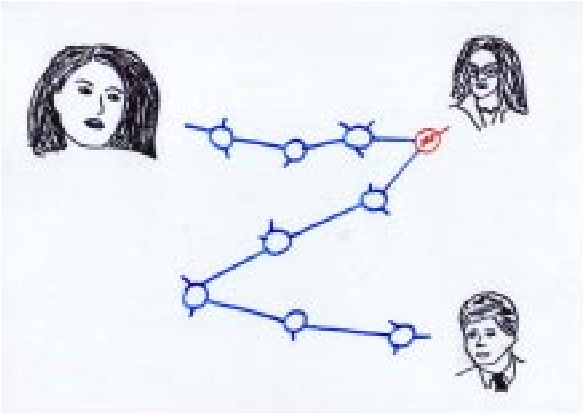

Figure 1. A vulnerable network

Cryptography can help, if it is used correctly. Monica and Bill can run a *security protocol*: a previously agreed message handshake. Figure 2 presents SSL (Secure Sockets Layer), which is used in Web servers. In the first two messages, Monica and Bill contact each other. Then, using public-key cryptography, they generate temporary encryption keys. Eventually they have their electronic conversation, using the new keys for encryption. Nobody can listen in, and Monica or Bill will notice if a message they receive has been modified during transmission.

An SSL handshake consists of up to eight messages and relies on several different cryptographic primitives. There are good reasons for this complexity. The protocol must thwart all known forms of attack, such as somebody's sending messages built from parts of messages she has intercepted. The protocol also tries to minimize the use of slow public-key operations.

Security protocols often contain serious errors. Formal verification can find errors and can increase our understanding of a protocol by making essential properties explicit. Communications protocols, which must work over an unreliable medium, have been verified for years. Researchers specify how the medium can go wrong: whether messages can be reordered, duplicated, etc. Similarly, with security protocols we must specify the adversary's capabilities. An *active attacker*—one who can send messages—is more dangerous than a passive eavesdropper. The former can launch a *middle-person attack* (Fig. 3), placing herself between the

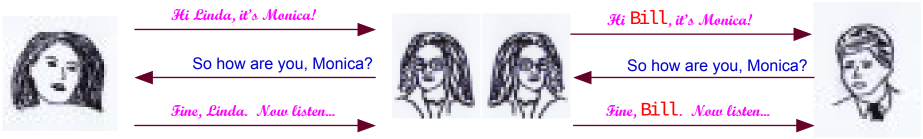

Figure 3. A middle-person attack

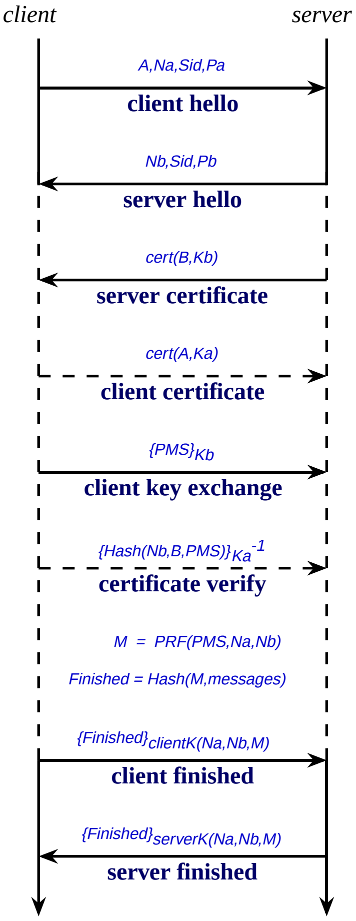

Figure 2. An Internet security protocol

two parties and typically modifying their messages before forwarding them. While the adversary is often called the “intruder”, she could be a corrupt insider, trusted by the others. We must assume the worst possible case.

Considering how simple they are, security protocols are surprisingly hard to verify. The main reason is that their goals are informally stated and poorly understood [10]. *Secrecy* is an obvious goal, but just as important is *authentication*: knowing who originated a message. Both properties are necessary in order to be sure that we are talking to the right person. Other security goals might be anonymity or non-repudiation [33]. The security community debates these concepts endlessly and delights in finding flaws in other people’s work.

By way of comparison, a floating-point division unit is much more complicated than a security protocol. However, there is a standard specification of what the unit should do. The verifier does not have to consider the risks posed by a hostile environment.

This paper presents a tutorial on security protocols, with the hope of pointing out some pitfalls. It begins (§2) by distinguishing long-term keys from session keys and explaining why the latter need to be fresh. It presents two very different protocol flaws in order to illustrate their diversity (§3). Then it introduces my work (the *inductive method* [23]), with some notes on its origins in other work (§4). To demonstrate the method, the paper describes how the Yahalom protocol is modelled (§5) and how simple properties can be proved (§6). Secrecy properties are harder to prove, but they are more interesting (§7). The concluding section discusses the general limitations of formal analysis (§8).

2 Freshness of nonces and keys

Suppose you find a fish in the back of the fridge. You don’t know how old it is. It smells all right, and if cooked thoroughly it might be OK to eat—but most of us would prefer to go hungry. Freshness is equally important in security, and some items have a short shelf-life.

Most security protocols ultimately rely on long-term keys. Each participant (or their institution) will have one. They are guarded like the Crown Jewels: their loss compromises everything derived from them. Setting up these secrets takes considerable effort, including face-to-face contact. Even with public-key cryptography, Monica needs someone trustworthy to deliver her public key to Bill: how can he be sure the key is Monica's?

Since each encrypted message gives another hint to any code-breakers, long-term keys should be used as little as possible. Normally they are used to set up short-term *session keys* that carry the bulk of the traffic. With public-key systems, the two parties can exchange encrypted random numbers, then scramble them to arrive at a session key. An alternative is to rely on a trusted third party: the *authentication server*.

As their name implies, session keys are used for one session only. They cannot be guarded like the Crown Jewels because there are too many of them: they are generated in great quantities. Moreover, some of them are shared with our adversary or her co-conspirators. Like that fish, we must regard an old session key as potentially compromised.

By replaying past messages, an adversary might deceive Bill into using an old session key. To thwart replay attacks, most protocols use *nonces*: numbers that are generated to identify protocol runs uniquely. During a run, each party must be able to insert a nonce of their choice. Later, when accepting a session key or other credentials, they will expect to see the same nonce attached to it in a tamper-evident message (one that is protected by a cryptographic integrity check). They conclude that the message is new: it must have been generated more recently than their nonce.

Other protocols use synchronized clocks and timestamps in order to detect the use of old messages. Burrows et al. [6] have abstracted the notion of freshness from such mechanisms. An item is *fresh* if it has not been used before.

3 Protocol failure: two examples

We can see these concepts at work in the Otway-Rees protocol, which is designed to let anybody establish a secure connection with anybody else. The protocol assumes a shared-key cryptosystem: every user has a unique long-term key, which they share with S, the authentication server. Consider this faulty variant of the protocol (Fig. 4), by Burrows et al. [6]:

1. $A \to B : Na, A, B, \{Na, A, B\}_{Ka}$
2. $B \to S : Na, A, B, \{Na, A, B\}_{Ka}, Nb, \{Na, A, B\}_{Kb}$
3. $S \to B : Na, \{Na, Kab\}_{Ka}, \{Nb, Kab\}_{Kb}$
4. $B \to A : Na, \{Na, Kab\}_{Ka}$

In this notation, Ka is A's long-term key and Kb is B's. Note that $\{X\}_K$ stands for the message X encrypted with key K. The encryption is assumed to be strong: given $\{X\}_K$, nobody can obtain the plaintext X or create a new ciphertext $\{X'\}_K$ unless they have the key K. Here is an informal description of each protocol step.

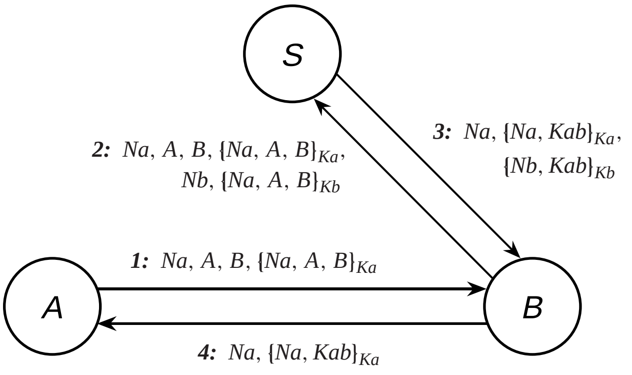
S
A
B
1: $Na, A, B, \{Na, A, B\}_{Ka}$
2: $Na, A, B, \{Na, A, B\}_{Ka}$, $Nb, \{Na, A, B\}_{Kb}$
3: $Na, \{Na, Kab\}_{Ka}$, $\{Nb, Kab\}_{Kb}$
4: $Na, \{Na, Kab\}_{Ka}$

Figure 4. A bad variant of Otway-Rees

1. A contacts B, choosing a fresh nonce Na to identify the run.

2. B contacts S, choosing a fresh nonce Nb. He cannot read the message $\{Na, A, B\}_{Ka}$ because it is encrypted using A's key. He simply forwards it to S, along with a similar encrypted message of his own.

3. S knows everybody's keys, so he can read A's and B's encrypted messages. He chooses a fresh session key, Kab. For A he encrypts Kab together with Na using Ka. Such a ciphertext is sometimes called a *certificate* or ticket. He prepares a similar certificate for B and sends him both of them.

4. B takes his certificate. He verifies that the nonce it contains is the same one (Nb) that he sent out in step two. Once satisfied, he forwards the other certificate to A, who will similarly inspect her certificate. If both of them are satisfied, then they will proceed to communicate using Kab as a session key.

A lot of things are going on here, but we might expect this protocol to be correct—especially as it is claimed to be in a landmark paper on protocol verification [6, p. 247]. However, a malicious user C can attack it (Fig. 5). The attack involves two interleaved runs. It works because the two certificates sent in message 3 have identical formats and because nonce Nb is unprotected in message 2. (The notation C_A stands for C masquerading as A.)

Clearly, protocols must be checked by mechanical tools. Attacks of such complexity are hard even for experts to notice. But protocols can fail in other ways. For the second example, I have inserted a subtle flaw into the Yahalom[1]

[1] pronounced Ya-ha-LOM

1. $A \to C_B : Na, A, B, \{Na, A, B\}_{Ka}$
1′. $C \to A : Nc, C, A, \{Nc, C, A\}_{Kc}$

A tries to contact *B*. However, *C* intercepts the message and starts a new run, contacting *A* in the normal way.

2′. $A \to C_S : Nc, C, A, \{Nc, C, A\}_{Kc}, Na', \{Nc, C, A\}_{Ka}$
2″. $C_A \to S : Nc, C, A, \{Nc, C, A\}_{Kc}, Na, \{Nc, C, A\}_{Ka}$

A responds to *C*'s message as the protocol requires, by trying to contact *S*. But *C* modifies her message, replacing nonce Na' by *Na* (circled above).

3′. $S \to C_A : Nc, \{Nc, Kca\}_{Kc}, \{Na, Kca\}_{Ka}$

In reponse to the modified message, *S* sends *A* a session key to be shared between her and *C*.

4. $C_B \to A : Na, \{Na, Kca\}_{Ka}$

C grabs the two certificates and sends *A* the wrong one. It contains *Na*, so *A* accepts *Kca* as a session key sent by *B* in the first run. But this key is actually shared with *C*.

Figure 5. An attack on Otway-Rees

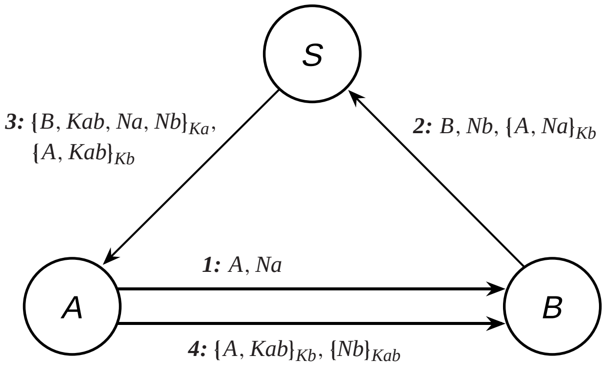
S
3: $\{B, Kab, Na, Nb\}_{Ka}$, $\{A, Kab\}_{Kb}$
2: $B, Nb, \{A, Na\}_{Kb}$
1: A, Na
A
B
4: $\{A, Kab\}_{Kb}, \{Nb\}_{Kab}$

Figure 6. A bad variant of Yahalom

protocol:

1. $A \to B : A, Na$
2. $B \to S : B, Nb, \{A, Na\}_{Kb}$
3. $S \to A : \{B, Kab, Na, Nb\}_{Ka}, \{A, Kab\}_{Kb}$
4. $A \to B : \{A, Kab\}_{Kb}, \{Nb\}_{Kab}$

The triangular configuration (Fig. 6) is designed to allow both *A* and *B* to know that the other party has been active. But the variant shown above is vulnerable: *B*'s certificate $\{A, Kab\}_{Kb}$ contains no nonce and thus gives no evidence of freshness. Let *K* be an old session key that had once been shared by *A* and *B*. If *C* has somehow obtained *K* and has kept the old certificate $\{A, K\}_{Kb}$ containing this key, then she can execute a bogus run with *B*:

1. $C_A \to B : A, Nc$
2. $B \to C_S : B, Nb, \{A, Nc\}_{Kb}$
4. $C_A \to B : \{A, K\}_{Kb}, \{Nb\}_K$

Now *B* has accepted an old, compromised session key and regards it as shared with *A*. As with Otway-Rees, the correct version of the protocol encrypts nonce *Nb* in message 2. But the similarity is superficial: the two attacks could hardly be more different.

- Otway-Rees requires *Nb* to be protected from modification. The attack involves interleaved runs and a malicious insider. The victim receives a fresh key, shared unfortunately with the wrong person.
- Yahalom requires *Nb* to be kept secret. The attack is trivial and can be carried out by an outsider. The victim receives a key that was once fit for the purpose but has become compromised.

It is hard to find a verification method that is sensitive to both types of problem, even in such simple protocols as these. The BAN logic of Burrows, Abadi and Needham [6] was for many years the basis of most research on protocol verification. Freshness reasoning is its main strength: it can analyze Yahalom. But it does not attempt to cover every aspect of correctness; it is silent on matters of secrecy. It finds nothing wrong with the version of Otway-Rees given above. Other protocols verified using BAN have since been attacked, for example by Lowe's model checking techniques [13].

Model checking is inexpensive and highly effective, but it too has limitations. The state-space explosion forces the model to be kept simple, typically limiting the number of participants and other parameters to one or two. Recently, Lowe [15] and Roscoe [27] have made advances on this problem; for some types of protocol, correctness in the general case can be shown by checking a finite model.

An alternative is to rely entirely on proof. The rest of this paper is devoted to my own work: inductive proofs mechanized using the Isabelle system. Other recent work deserves attention: the theory of *strand spaces* [31] is a promising foundation for proving protocols correct.

4 An inductive model of security protocols

This section outlines the inductive method [23]. It attempts to suggest principles about how to model a system using mechanical proof tools; towards this end, it describes how the main primitives evolved.

A model must be detailed enough to be useful and simple enough to be usable. If proofs are to be performed by machine, then simplicity becomes even more important. (Mechanical proofs are harder than paper ones, not easier, because even obvious steps must be justified formally.) Also, a model should be easy to explain to other people, especially to sceptical security experts.

When people reason about protocols informally, they often use induction. They argue that each protocol step preserves the desired safety properties. If we model protocols inductively, then the machine proofs will have the same structure as informal ones, which is a major advantage. Note that an inductive definition can be seen as an operational semantics.

Lowe [13] applies model checking to a theory jointly developed with Schneider [30] under the supervision of Roscoe [26, 28]. This outstanding work is based on Communicating Sequential Processes [11], but in essence it uses an operational semantics. Its ideas, with suitable modifications, form the basis for the inductive method.

4.1 Agents and messages

Before we can talk about particular protocols, we must formalize the agent population and the general structure of messages. These are defined in Isabelle [22] essentially as follows (some details are omitted):

```
datatype agent = Server | Friend nat | Spy
datatype msg = Agent agent
             | Nonce nat
             | Key   key
             | MPair msg msg
             | Crypt key msg
```

There are three kinds of agents: the server S (called `Server` in the text), the friendly agents (which are indexed by natural numbers) and the spy. The spy is an active attacker and is accepted as a valid user. The spy has taken control of an arbitrary set of compromised agents, but the server is always trustworthy.

A message can be the name of an agent, a nonce, a key, a pair of messages, or a message encrypted using a key. The familiar notation $\{X_1, \ldots X_{n-1}, X_n\}$ is defined to abbreviate $\mathsf{MPair}\ X_1\ \ldots (\mathsf{MPair}\ X_{n-1}\ X_n)$.

The `datatype` declarations specify types that are free algebras: the constructors are injective and their ranges are mutually disjoint. In particular, we have

$$\mathsf{Crypt}\,K\,X = \mathsf{Crypt}\,K'X' \Longrightarrow K = K' \wedge X = X'.$$

This theorem states that a ciphertext can be decrypted using only one key and can yield only one plaintext. Some real-world encryption methods do not fit this black-box model. With RSA for example, multiplication commutes with encryption [25]. Fortunately, most security protocols assume an underlying implementation of strong encryption. It is not clear how to model those that do not.

4.2 The function parts

The next step is to define operations on messages. They are needed for expressing assertions and describing the possible actions of the adversary. The function parts maps sets of messages to sets of messages: $\mathsf{parts}\,H$ consists of the components of elements of H. The components of a message include the message itself; the components of $\mathsf{Crypt}\,K\,X$ include all the components of the plaintext X, but not necessarily the key K. An obvious inductive definition gives $\mathsf{parts}\,H$ the required closure properties.

Why should the notion of component be formalized as a function from sets to sets? Lowe [13, §5.1] defines the contains relation on messages, where X contains Y holds provided Y is a component of X. This approach may look straightforward, but it is cumbersome in practice. Lowe immediately defines the set of all sub-messages of a message, namely the set of all messages that it contains. He then derives lemmas that contain many formulæ of the form $\exists X \in H\,.\,X\ \mathsf{contains}\ Y$. Isabelle supports logical variables and can prove existential formulæ automatically, but we should not burden it needlessly. So contains is not a good basis for mechanization. It happens that $\mathsf{parts}\,H$ equals

$$\{Y \mid \exists X \in H\,.\,X\ \mathsf{contains}\ Y\},$$

but we can dispense with contains entirely.

A further advantage of parts is that it enjoys equational reasoning. Obviously contains is transitive, but transitivity reasoning requires search and can diverge. With parts, transitivity is expressed by a neat equation:

$$\mathsf{parts}(\mathsf{parts}\,H) = \mathsf{parts}\,H$$

This equation and others shown below are easy to prove. The two inclusions of the set equality are shown separately, typically by rule induction (induction over the definition).

4.3 The function analz

The function analz is intended to model what the adversary can extract from a corpus of messages. The closure properties of the set $\mathsf{analz}\,H$ are defined inductively. The only difference from the definition of $\mathsf{parts}\,H$ is in the rule for Crypt:

$$\frac{\mathsf{Crypt}\,K\,X \in \mathsf{analz}\,H \quad K^{-1} \in \mathsf{analz}\,H}{X \in \mathsf{analz}\,H}$$

This rule states that decryption requires the matching key. (The inverse of a key is meaningful for public-key cryptosystems; for shared keys, $K^{-1} = K$.)

As with parts, set equalities can be proved by rule induction. Transitivity is again expressed by idempotence:

$$\mathsf{analz}(\mathsf{analz}\,H) = \mathsf{analz}\,H,$$

The close relationship between the two operators is revealed by two further equations:

$$\mathsf{parts}(\mathsf{analz}\,H) = \mathsf{parts}\,H \qquad \mathsf{analz}(\mathsf{parts}\,H) = \mathsf{parts}\,H.$$

Again, introducing a map from sets to sets may seem unconventional. Schneider [30] writes $H \vdash X$ to state that an intruder can derive X from a set H of messages. (Bolignano's *known_in* relation [5] is similar.) This relation already concerns a set of messages, so by analogy with parts we could formalize the set $\mathsf{entails}\,H = \{X \mid H \vdash X\}$. The set would be both upwards and downwards closed under message construction. But this approach can be simplified. The relation $H \vdash X$ comprises two distinct questions:

1. Can X be extracted from H by decrypting ciphertexts with available keys?

2. Can X can be built up from elements of H?

Most of the time we are only interested in question 1, which expresses safety properties. We only care about question 2 when considering what active attacks the intruder could launch. It is better to replace $\mathsf{entails}\,H$ by two separate operators, $\mathsf{analz}\,H$ and $\mathsf{synth}\,H$.

4.4 The function synth

The set $\mathsf{synth}\,H$ captures the "building up" aspect of entailment. Its inductive definition says that it includes all agent names but no nonces or keys other than those in H. (Agent names are guessable, while nonces and keys aren't.) If X and Y belong to $\mathsf{synth}\,H$ then so does $\{X, Y\}$ and $\mathsf{Crypt}\,K\,X$, provided $K \in H$. (Two elements can be combined and can be encrypted using available keys.)

This operator is also idempotent,

$$\mathsf{synth}(\mathsf{synth}\,H) = \mathsf{synth}\,H.$$

Two other equations express its relationship with parts and analz.

$$\mathsf{parts}(\mathsf{synth}\,H) = \mathsf{parts}\,H \cup \mathsf{synth}\,H$$
$$\mathsf{analz}(\mathsf{synth}\,H) = \mathsf{analz}\,H \cup \mathsf{synth}\,H$$

These equations show that the "breaking down" and "building up" aspects of entailment are independent. There do not appear to be similar equations for $\mathsf{synth}(\mathsf{parts}\,H)$ and $\mathsf{synth}(\mathsf{analz}\,H)$. The latter combination of operators expresses the set of fake messages that an intruder could invent starting from H. By analogy with modal logic, we have five modalities: parts, analz, synth, $\mathsf{synth} \circ \mathsf{parts}$ and $\mathsf{synth} \circ \mathsf{analz}$. The last of these seems to satisfy only a few equational laws, such as

$$\begin{aligned}&\{X, Y\} \in \mathsf{synth}(\mathsf{analz}\,H)\\ &\quad\Longleftrightarrow\; X \in \mathsf{synth}(\mathsf{analz}\,H) \wedge Y \in \mathsf{synth}(\mathsf{analz}\,H).\end{aligned}$$

Symbolic evaluation is another benefit of separating $\mathsf{entails}$ into analz and synth. Proof by induction typically requires symbolic evaluation to reduce an assertion about $\{X\} \cup H$ to one about H, where we have an induction hypothesis about H and X is a new message. This is obviously impossible for synth (and $\mathsf{entails}$ would be even worse) because infinitely many new messages can be generated from X. However, if X is sufficiently definite then $\mathsf{analz}(\{X\} \cup H)$ can be simplified to a complicated expression built from $\mathsf{analz}\,H$. The simplifier therefore does a tremendous amount of work that would otherwise have to be done manually, with many case splits.

4.5 Traces of events

Still following the CSP approach [13, 30], let us specify the system's behaviour as the set of possible traces of events. A trace model is concrete and easy to explain. An *event* is an atomic action, one of three forms:

- $\mathsf{Says}\,A\,B\,X$ occurs when A attempts to send B the message X.

- $\mathsf{Gets}\,A\,X$ occurs when A receives the message X. She cannot be expected to know where it came from.

- $\mathsf{Notes}\,A\,X$ occurs when A stores the message X internally. (Here X may be the result of a calculation.)

For each Gets event there is a corresponding Says event. G. Bella [2] introduced the Gets event to allow explicit reasoning about agent knowledge. We can do much without it [23], but it improves the readability of protocol definitions and theorem statements.

5 Modelling the Yahalom protocol

The Yahalom protocol [6, p. 257] is a good example for evaluating verification methods. It is short (four messages) but its analysis is difficult. As mentioned above, it demonstrates the importance of freshness: session keys degrade with age. It also involves a particularly tricky treatment of secrecy.

Many authors, including myself, have demonstrated their work on a different example: the Needham-Schroeder public-key protocol. Lowe famously found a flaw in this protocol [13]. However, Yahalom is richer in every way.

Protocol specifications make use of additional functions that operate on a trace evs. Their intuitive meanings are as follows:

- $ev\#evs$ extends the trace with the new event ev. (The operator # is the familiar list "cons"; traces are constructed in reverse order.)
- $\mathsf{set}\,evs$ is the set of all past events. (The function set maps a list to the set of its elements.)
- $\mathsf{knows}\ \mathsf{Spy}\,evs$ is the set of messages the spy can see, which includes all messages sent over the network.
- $\mathsf{used}\,evs$ contains all components of all past messages.

The functions set, knows and used are defined by list recursion.

Figure 7 presents the Isabelle specification of the Yahalom protocol. It defines the correct version, with nonce Nb encrypted in message 2. As usual with operational semantics, the rules are quite readable; compare with the informal description given in §3. Full details of the Yahalom proofs appear elsewhere [21].

The inductive definition of $\mathsf{yahalom}$, which is a set of traces, comprises eight rules. The first three are required in all specifications. There is a rule to allow the empty trace. The **Fake** rule models an active attacker sending a message built from components taken from past traffic. In detail, he can extend the current trace, evs, with the event $\mathsf{Says}\ \mathsf{Spy}\ B\ X$, where X is constructed from previous traffic, $\mathsf{knows}\ \mathsf{Spy}\,evs$, using the functions synth and analz. The **Reception** rule says that if A sends a message to B then B might receive it. (But he might not: the rules do not have to be applied, so there will be traces in which some messages are lost.)

The next four rules are the protocol steps themselves, as they would be performed by honest agents. The function used expresses that agents choose *fresh* nonces, presumably by generating big random numbers. Also, $\mathsf{shrK}\ A$ is the formal syntax for Ka, the long-term key that A shares with the authentication server.

The last and most interesting rule is **Oops**, which compromises session keys. Such a rule is necessary; otherwise session keys remain secure forever and the model cannot distinguish the correct version of Yahalom from the bad one presented in §3.

The simplest type of Oops rule states that any session key distributed by the server can reach the spy. Naturally, any security guarantees proved about a key K will require that K has not been lost in this way. With such a model, we can show that a protocol still works even if some session keys are lost. However, that model is not detailed enough to distinguish between these two scenarios:

- B received a bad session key because it became compromised soon after being issued. (B was unlucky.)
- B received a bad session key because an intruder managed to replace the good key by an old one. (The protocol has a bug.)

Time is the missing parameter: the Oops message must indicate when the key was lost. Bella modelled the BAN-Kerberos protocol [4] making time explicit, and his Oops rule simply said that no session key could be lost during its stated lifetime. (It is reasonable to assume that session keys are safe for this short period.) Time is implicit for Yahalom, so the Oops event hands $\{Na, Nb, K\}$ to the spy. Since the key and the relevant nonces are bundled together, we can distinguish between a recent loss and an old one.

6 Simple lemmas about Yahalom

Protocol analyses using the inductive method follow the usual techniques for reasoning about an operational semantics. Lemmas are established, mostly using rule induction, until finally the desired properties are proved. Quite a variety of protocols have been analyzed: using public-key encryption or shared-key encryption (or both), distributing nonces or keys (or both) or even calculating session keys. While there is some variety in the theorems that are proved, there is also much uniformity.

Assertions such as

$$X \in \mathsf{parts}(\mathsf{knows}\ \mathsf{Spy}\,evs) \Longrightarrow \cdots$$

or

$$ev \in \mathsf{set}\,evs \Longrightarrow \cdots$$

are often easy to prove. (The function parts has good simplification laws.) Such facts are called *regularity lemmas*. For example, long-term keys remain secret:

$$\mathsf{Key}(\mathsf{shrK}\ A) \in \mathsf{parts}(\mathsf{knows}\ \mathsf{Spy}\,evs) \Longrightarrow A \in \mathsf{bad}\,.$$

empty trace

```
[] ∈ yahalom
```

Fake

```
⟦ evs ∈ yahalom;  X ∈ synth (analz (knows Spy evs)) ⟧
⟹ Says Spy B X  # evs ∈ yahalom
```

Reception

```
⟦ evsr ∈ yahalom;  Says A B X ∈ set evsr ⟧
⟹ Gets B X # evsr ∈ yahalom
```

Message 1

```
⟦ evs1 ∈ yahalom;  Nonce NA ∉ used evs1 ⟧
⟹ Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom
```

Message 2

```
⟦ evs2 ∈ yahalom;  Nonce NB ∉ used evs2;
  Gets B {|Agent A, Nonce NA|} ∈ set evs2 ⟧
⟹ Says B Server
        {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
      # evs2 ∈ yahalom
```

Message 3

```
⟦ evs3 ∈ yahalom;  Key KAB ∉ used evs3;
  Gets Server {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
    ∈ set evs3 ⟧
⟹ Says Server A
         {|Crypt (shrK A) {|Agent B, Key KAB, Nonce NA, Nonce NB|},
           Crypt (shrK B) {|Agent A, Key KAB|}|}
      # evs3 ∈ yahalom
```

Message 4

```
⟦ evs4 ∈ yahalom;  A ≠ Server;
  Gets A {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},  X|}
    ∈ set evs4;
  Says A B {|Agent A, Nonce NA|} ∈ set evs4 ⟧
⟹ Says A B {|X, Crypt K (Nonce NB)|} # evs4 ∈ yahalom
```

Oops

```
⟦ evso ∈ yahalom;
  Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},  X|}
    ∈ set evso ⟧
⟹ Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ yahalom
```

Figure 7. Specifying the Yahalom protocol

Here, the constant *bad* stands for the set of compromised agents, and the definition of knows makes their keys available to the spy. The theorem says that if A is honest then Ka will never form part of any traffic, let alone reach the spy.

Unicity lemmas are useful. For example, if the server has sent the two messages

$$\{B, K, Na, Nb\}_{Ka}$$
$$\{B', K, Na', Nb'\}_{Ka'}$$

(with the same K each time) then all the corresponding message components agree. This theorem holds because the server never uses a session key twice.

Authenticity theorems are important. A trivial induction shows that if

$$\{B, Kab, Na, Nb\}_{Ka}$$

appears in traffic then it was sent by the server in message 3 (if $A \notin \mathsf{bad}$). This simple result assures A that this message came from the server. Because the session key is securely bundled with A's nonce, she knows it to be fresh. Similarly, one can prove inductively that the server sends this message only after receiving a copy of message 2. Continuing in this vein, we can reason backwards from any point in a protocol run.

For Yahalom, B's authenticity theorem is hard to prove. We can easily show that if $\{A, Kab\}_{Kb}$ appears in traffic (and $B \notin \mathsf{bad}$) then it was sent by the server. But this certificate contains no nonces, so it says nothing about the freshness of Kab. For that we must turn to the second half of message 4. If $\{Nb\}_{Kab}$ appears in a trace then the server used Kab and Nb in the same instance of message 3. Together with the result about $\{A, Kab\}_{Kb}$, this theorem assures B that the session key is fresh. However, the theorem makes a significant assumption: that Nb is kept secret. A crucial step in its proof is that the spy could not have created the message $\{Nb\}_{Kab}$ because he does not know Nb.

7 Secrecy theorems for Yahalom

For all protocols that involve session keys, we must prove that those keys remain secret. For Yahalom, we must moreover prove that Nb remains secret. We must prove secrecy in the presence of the Oops rule, which compromises some keys and nonces; we must establish bounds on the damage caused by such losses. Secrecy theorems are expressed in terms of analz, the function that returns all that can be decrypted from its argument, and the simplification rules for analz are inconvenient. For all these reasons, secrecy is hard to prove.

The *session key secrecy theorem* asserts correctness of the protocol from the server's point of view. It is subject to several conditions:

- evs is a Yahalom trace.
- A and B are honest agents.
- The server has sent message 3, issuing the key K to A and B:

$$\begin{aligned}\mathsf{Says\ S}\ A\ \{\,&\mathsf{Crypt}(\mathsf{shrK}\ A)\{B, K, Na, Nb\},\\ &\mathsf{Crypt}(\mathsf{shrK}\ B)\{A, K\}\} \in \mathsf{set}\ evs\end{aligned}$$

- The session key was not lost in this run, that is, in an Oops event involving the *same* nonces:

$$\mathsf{Notes\ Spy}\ \{Na, Nb, K\} \notin \mathsf{set}\ evs$$

If these hypotheses hold, the theorem concludes that the key is safe from the spy:

$$K \notin \mathsf{analz}(\mathsf{knows\ Spy}\ evs).$$

As mathematicians know, sometimes a lemma can be as interesting as the main result. Proving the theorem above requires such a lemma. The induction presents us with goals of the form

$$K \notin \mathsf{analz}(\mathsf{knows\ Spy}\ (ev\#evs))$$

where $ev\#evs$ is the current trace extended with ev, the latest event. If that event is an instance of message 3, then it contains another session key K', which is possibly being distributed to a bad agent. So we have to simplify the formula

$$K \in \mathsf{analz}(\{K'\} \cup \mathsf{knows\ Spy}\ evs), \qquad (1)$$

where both K and K' are session keys. A little thought reveals that the formula ought to be equivalent to

$$K = K' \lor K \in \mathsf{analz}(\mathsf{knows\ Spy}\ evs). \qquad (2)$$

Knowing the extra key K' does not make it any easier to decrypt K from past traffic because the protocol never encrypts session keys using other session keys.

When we try to prove the equivalence between (1) and (2), induction fails. Message 3 is again the culprit; it introduces yet another session key. We have to generalize the induction formula to a set $\mathcal{K}$ of session keys:

$$\begin{aligned}&K \in \mathsf{analz}(\mathcal{K} \cup \mathsf{knows\ Spy}\ evs)\\ &\qquad \iff K \in \mathcal{K} \lor K \in \mathsf{analz}(\mathsf{knows\ Spy}\ evs)\end{aligned}$$

With this lemma in hand, we can prove the session key secrecy theorem. The lemma is important in its own right because it expresses a robustness principle: the loss of one session key does not compromise others. So I call it the *session key compromise theorem*. Some protocols do not satisfy this theorem, such as Kerberos IV, where the loss of one session key can compromise any number of other keys. Yet the losses do not cascade; the protocol can be verified [3].

Yahalom poses similar problems: we must prove secrecy of *Nb*, but the loss of a session key compromises the corresponding nonce. It was for Yahalom that the necessary technique was first developed [21]. Very briefly, the idea is to formalize the relation between session keys and the items they encrypt. Then use this relation to express the session key compromise theorem in restricted form. If the server has associated Nb' with K in message 3 then

$$Nb \in \mathsf{analz}(\{K\} \cup \mathsf{knows\ Spy}\ evs) \iff Nb \in \mathsf{analz}(\mathsf{knows\ Spy}\ evs)$$

provided $Nb \neq Nb'$. In other words, Nb' is the only nonce that could be affected by the loss of K.

Let us review the analysis of Yahalom. Near the end of §6, I outlined the proof that *B* receives a fresh key. In part, the proof argued that the spy could not have created the message $\{Nb\}_{Kab}$ because he did not know *Nb*. To prove that *Nb* was secret required a complicated and lengthy argument.

As an experiment, I have formalized the bad version of Yahalom (Fig. 6) *without* Oops to demonstrate why that rule is needed. In this version, *Nb* is not secret. But the session key secrecy theorem holds as usual, and with Oops omitted, session keys remain secret for ever. So we can argue that the spy could not have created the message $\{Nb\}_{Kab}$ because he did not know *Kab*. This proof concludes that *Kab* is a good session key even though it is not fresh. This unrealistic proof is only half as long as the original one and is easy to find: the experiment described in this paragraph took only an hour, starting from the original Yahalom proof script.

Yahalom could be dismissed as a toy; a real-world example is the Internet protocol SSL (Fig. 2). It has more messages than Yahalom and uses fancier cryptographic primitives, but its design is straightforward. The primitives it needs are easily modelled [24]. A cryptographic hash function is expected to be collision-free, so the model assumes it to be injective. Hashing is expected to be non-invertible in practice, so the model does not equip the spy with the means to invert it. (But we must add a rule to let the spy to perform hashing.) Unlike Yahalom, there is nothing tricky about this protocol; it is hard to analyze simply because it is big and complicated.

8 Conclusions

Dijkstra has famously remarked [9, p. 6] that testing can show the presence of bugs but never their absence. Program proofs are generally thought to be free of this limitation. But we must abandon this claim as soon as we try to model an unreliable or hostile environment. Our proof has to be based upon a formal model, and a real-world adversary need not conform to our model.

Mao [16, p. 45] has commented "when BAN logic finds a proof of correctness, people seem to have trouble believing that it is a proof." The BAN paper itself makes only modest claims: "[the logic] guides us in identifying mistakes and suggesting corrections" [6, p. 269]. BAN reasons correctly about freshness, but it does not attempt to prove secrecy. It is also insensitive to attacks such as that of Fig. 5: although the session key is fresh, it is associated with the wrong agent.

Is model checking better? Mitchell et al.'s model of SSL [18] is rather abstract. The same authors' analysis of Kerberos [17] does not cover multiple runs or freshness. Model checking amounts to exhaustive testing of a system cut down to finite size.

At first sight, the inductive approach appears to give a high level of confidence. The protocols analyzed include three of industrial strength:

- The model of the SSL protocol (actually TLS, its successor) includes some details of the calculation of session keys for both initial runs and resumptions [24].

- Bella's inductive analysis of Kerberos [3] found a flaw in the treatment of session key lifetimes. The model admits the compromise of session keys and the proof copes with the cascading losses that can ensue.

- The analysis of the Recursive Authentication Protocol [23] copes with runs of arbitrary length.

All that may sound wonderful, but Ryan and Schneider found serious flaws in the Recursive Authentication Protocol [29]—or rather in the proposed implementation, which replaced the encryption by exclusive-OR and cryptographic hashing. Once we realize that perfection is not attainable, model checking regains its attraction, especially since Lowe's work [14] makes it almost cost-free.

Security analysts know that any system can be broken, given enough effort. Too many things can go wrong. Suppose that Monica's messages to Bill are always YES or NO, encrypted with Bill's public key. After observing the traffic for some time, a spy can probably work out which ciphertext means YES and which means NO. Such deductions lie outside of most models. However complicated our model is, significant details will be left out. Even stray electromagnetic radiation from equipment is a security risk; Wright gives an amusing account in *Spycatcher* [32, pp. 109–111].

Formulæ can be misinterpreted; here is an example. A recent version of Otway-Rees [1] is particularly easy to verify: certificates, if received, are genuine. But nothing in the Isabelle proofs requires nonces to be fresh, so what happens if somebody keeps using the same nonce? It takes a little thought to realize that even a genuine certificate would become worthless, since the nonce could give no evidence of freshness.

Alert readers may have noticed similarities between this paper and Roger Needham's of last year [19], in which he outlined the history of the BAN logic. BAN's main contribution, he said, was to express protocol mechanisms in terms of abstractions such as freshness. Abstractions made BAN usable but also made its analysis incomplete. We have considered similar issues above. Adding detail to our model (the Oops rule, for example) allows it to make important distinctions, but the proofs become much harder.

BAN derivations are short and straightforward; they do not require automated support. Model checking establishes properties automatically, and the preliminary formalization needs typically hours or days. Proofs using the inductive method typically require days or weeks of specialist work. When certifying a system, somebody must decide how much confidence is required and balance the cost of verification against that of failure. Often a combination of tools will be used, along with expert scrutiny. Automation can never replace informed judgement.

Acknowledgement Prof. C. A. R. Hoare scrutinized this paper and made extensive suggestions. G. Bella, K. Eastaughffe and F. Massacci also commented. The research was funded by the EPSRC grants GR/K77051 *Authentication Logics* and GR/K57381 *Mechanizing Temporal Reasoning* and by the ESPRIT working group 21900 *Types*.

References

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

[2] Giampaolo Bella. Message reception in the inductive approach. Technical Report 460, CUCL, March 1999.

[3] Giampaolo Bella and Lawrence C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Computer Security — ESORICS 98*, LNCS 1485, pages 361–375. Springer, 1998.

[4] Giampaolo Bella and Lawrence C. Paulson. Mechanising BAN Kerberos by the inductive method. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification: 10th International Conference, CAV '98*, LNCS 1427, pages 416–427. Springer, 1998.

[5] 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.

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

[7] *8th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1995.

[8] *11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[9] Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[10] Dieter Gollmann. What do we mean by entity authentication? In *Symposium on Security and Privacy* [12], pages 46–54.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] IEEE Computer Society. *Symposium on Security and Privacy*, 1996.

[13] 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. Springer, 1996.

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

[15] Gavin Lowe. Towards a completeness result for model checking of security protocols. In Computer Security Foundations Workshop [8], pages 96–105.

[16] Wenbo Mao. An augmentation of BAN-like logics. In Computer Security Foundations Workshop [7], pages 44–56.

[17] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murφ. In *Symposium on Security and Privacy*, pages 141–153. IEEE Computer Society, 1997.

[18] John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of SSL 3.0 and related protocols. In Orman and Meadows [20].

[19] Roger M. Needham. Logic and over-simplification. In *13th Annual Symposium on Logic in Computer Science*, pages 2–3. IEEE Computer Society Press, 1998.

[20] Hilarie Orman and Catherine Meadows, editors. *Workshop on Design and Formal Verification of Security Protocols*. DIMACS, September 1997.

[21] Lawrence C. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. *Journal of Computer Security*. in press.

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

[23] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[24] Lawrence C. Paulson. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Information and System Security*, in press.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[26] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In Computer Security Foundations Workshop [7], pages 98–107.

[27] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In Computer Security Foundations Workshop [8], pages 84–95.

[28] A. W. Roscoe and M. H. Goldsmith. The perfect “spy” for model-checking cryptoprotocols. In Orman and Meadows [20].

[29] Peter Y. A. Ryan and Steve A. Schneider. An attack on a recursive authentication protocol: A cautionary tale. *Information Processing Letters*, 65(1):7–10, January 1998.

[30] Steve Schneider. Security properties and CSP. In *Symposium on Security and Privacy* [12], pages 174–187.

[31] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In Computer Security Foundations Workshop [8], pages 66–77.

[32] Peter Wright. *Spycatcher: The Candid Autobiography of a Senior Intelligence Officer*. Heinemann Australia, 1987.

[33] Jianying Zhou and Dieter Gollmann. A fair non-repudiation protocol. In *Symposium on Security and Privacy* [12], pages 55–61.