Verification of Security Protocols Using A Formal Approach

Qian Wang

Kongens Lyngby 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Security protocols are expected to build secure communications over vulnerable networks. However, security protocols may contain potential flaws. Therefore, they need formal verifications.

In this thesis, we investigate Paulson's inductive approach and apply this formal approach to a classical cryptographic protocol which has not been previously verified in this way. We also investigate the modelling of timestamps and further extension of the inductive approach with message reception and agent's knowledge. We modelled and verified Lowe's modified Denning-Sacco shared-key protocol using the inductive approach. The model and theorems are later updated with message reception and agent's knowledge.

Theorem proving is supported by the interactive theorem prover *Isabelle*. We have completed the proofs for both versions. As a result, Lowe's modified Denning-Sacco shared-key protocol has been formally verified using the inductive approach.

Preface

This master thesis was prepared at the Department of Informatics Mathematical Modelling (IMM), Technical University of Denmark (DTU) in partial fulfillment of the requirements for acquiring the M.Sc. degree in Computer Systems Engineering.

The thesis project, entitled *Verification of Security Protocols Using A Formal Approach*, was carried out in the period of 1st of February to 31st of August 2007 and corresponds to 35 ETCS points. This thesis was supervised by Associate Professor Jørgen Villadsen at IMM, DTU.

The thesis consists of a report describing the project, and two Isabelle/HOL theory files.

Lyngby, August 2007

Qian Wang

Acknowledgements

Foremost, I would like to thank my supervisor, Jørgen Villadsen, for his inspiring ideas, and for his guide and support during the period of my thesis project.

I would also like to thank my friends for their emotional support during this period.

Last but not least, I would like to thank my dearest parents for their love and support all the time, especially when I am abroad.

Contents

Abstract i

Preface iii

Acknowledgements v

1 Introduction 1

2 Security Protocols 5

2.1 Possible Attacks 5

2.2 Security Goals 7

2.3 Cryptography 8

2.4 Cryptographic Protocols 10

2.5 Attacks against Protocols 12

2.6 Protocol Analysis 14

3 The Inductive Approach 19
3.1 Basics 20
3.2 Agent's Knowledge 23
3.3 Operators 26
3.4 Other Useful Functions 28

4 Modelling the Protocol 31
4.1 Lowe's Modified Denning-Sacco Shared-key Protocol 32
4.2 Modelling Timestamps 33
4.3 Formalizing the Protocol by Induction 35
4.4 Discussions 39
4.5 Update the Model 41

5 Verifying the Protocol 45
5.1 Proving Reliability Lemmas 45
5.2 Proving Forwarding Lemmas 48
5.3 Proving Regularity Lemmas 48
5.4 Proving Unicity Theorems 49
5.5 Proving Authenticity Guarantees 50
5.6 Proving Confidentiality Theorems 52
5.7 Proving Authentication Theorems 57
5.8 Updating Theorems and Proofs 59

6 Conclusion 61

A Verifying Lowe's Denning-Sacco Shared-key Protocol **65**

B Updated Model and Theorems with Message Reception **75**

Chapter 1

Introduction

On computer networks, a number of computer systems are connected for the sake of communication. When two peers on the network want to communicate with each other, the message traffics between them have to pass through some other peers on the vulnerable network. *Secure* communications are always expected. Participants may wish the messages can keep secrecy from others and are not altered by someone before they reach the destination. By receiving a message, the participant may expect that the message is not fabricated by any malicious party and is really originated with the claimed creator.

Security protocols are designed to achieve these goals. Most security protocol employs *cryptography*, thus they are also called *cryptographic protocols*. In this thesis, we mainly discuss *shared-key protocols* and thus set a *symmetric key* environment. For shared-key protocols, an agent shares a long-term key with the server and uses this key to exchange *session keys* which are short-term and only used to encrypt actual message.

However, security protocols may contain flaws. They are claimed to achieve certain security goals by their designers, but they often fail in fact. Possible attacks against several well-known protocols have been reported. And it is believed that potential flaws still exist. Informal reasoning is not sufficient to guarantee the correctness of a security protocol. Thus, several formal approaches have been developed to verify security protocols. Formal verification can find errors

and can increase our understanding of a protocol by making essential properties explicit [27].

Paulson's *inductive approach* [26] is one of the successful formal methods. The protocol model, constructed by induction, is permissive and unbounded. A history of agent's behaviors is formalized as a list of *events*, which is called a *trace*. A protocol is then formalized as the set of all possible traces. Security properties are specified and proved by induction on a generic trace. If a security property can hold on a generic trace through extension of each inductive step, then it is proved that the protocol maintains this property. The proof is assisted by the interactive theorem prover *Isabelle* [25].

Thesis Objective

The main goal of this thesis project is to investigate Paulson's inductive approach [26], and apply this formal approach to a classical cryptographic protocol which has not been formally modelled and verified in this way. We choose Lowe's modified version of Denning-Sacco shared-key protocol [18]. It uses symmetric key cryptography to seal messages and relies on both *timestamp* and *nonce* to give evidences of *freshness*. With the inductive approach, this protocol is going to be analyzed and its correctness is going to be verified.

We start form the theoretical background about cryptographic protocols, possible attacks, and protocol analysis in both informal and formal ways. We then give emphasis to the investigation of the inductive approach, including Paulson's original approach [26], the modelling of timestamps [6], and Bella's extension with message reception [5]. The selected protocol is formalized and analyzed. Several security properties are specified and then proved with the support of the theorem prover. The protocol is first modelled using Paulson's original approach, and then the model is updated with Bella's extension of message reception. Subsequently, the proof for the original model is also updated with message reception.

Thesis Outline

The outline of this thesis is presented with brief description for each chapter.

Chapter 2 reviews several aspects about security protocols. General types of attacks and expected security goals are presented, and cryptography protocols are introduced. We take a protocol as the example to describe protocol flaws and possible attacks against protocols. Informal and formal methods for protocol analysis are reviewed.

Chapter 3 introduces Paulson's inductive approach which is adopted for this thesis project. Basic principles of the inductive approach are given, and basic constituents are discussed.

Chapter 4 presents how a cryptographic protocol is modelled using the inductive approach. We choose Lowe's modified Denning-Sacco shared-key protocol [18] as the example. Since it is a timestamp-based protocol, we first concerns how timestamps are modelled, and Bella's extension of inductive approach with timestamps [6] is presented. The protocol is first formalized using Paulson's original approach [26]. Then the model is updated with Bella's extensions of message reception [5]. We also discuss a couple of issues on the modelling of protocols.

Chapter 5 describes how Lowe's modified Denning-Sacco shared-key protocol is verified using the inductive approach. Based on our inductive model, the expected security goals are analyzed and then formalized as several theorems. These theorems are proved with the support of the theorem prover *Isabelle* [25]. After that, we also update the theorems and their proofs with message reception.

Chapter 6 gives a conclusion of the thesis project and summarizes our work.

Appendix A contains our Isabelle/HOL theory for modelling and verifying Lowe's modified Denning-Sacco shared-key Protocol.

Appendix B contains our updated theory with Bella's extension of message reception and agent's knowledge.

CHAPTER 2

Security Protocols

This chapter presents several aspects about security protocols. At the beginning, general types of possible security attacks over a vulnerable network are introduced (§ 2.1), and several general security goals are expected (§ 2.2). Then the network protocols employing cryptography (§ 2.3) are introduced. (§ 2.4) The cryptographic protocols are expected to provide secure communications over insecure networks and achieve some security goals. However, cryptographic protocols may contain flaws and thus can be affected by attackers. Possible attacks against protocols are discussed and illustrated by an example (§ 2.5). In this case, several methods arise for verifying security protocols. We mention the informal methods and then introduce some of the major formal approach (§ 2.6).

2.1 Possible Attacks

A computer network contains a number of computing systems. Since the message flows have to pass through other network nodes, communications over computer networks are not safe. Suppose Alice wants to communicate with Bob on an insecure network. An intruder may attack the communication in various ways. Compared with the unaffected normal flow, five different types of possible attacks are illustrated in Figure 2.1.

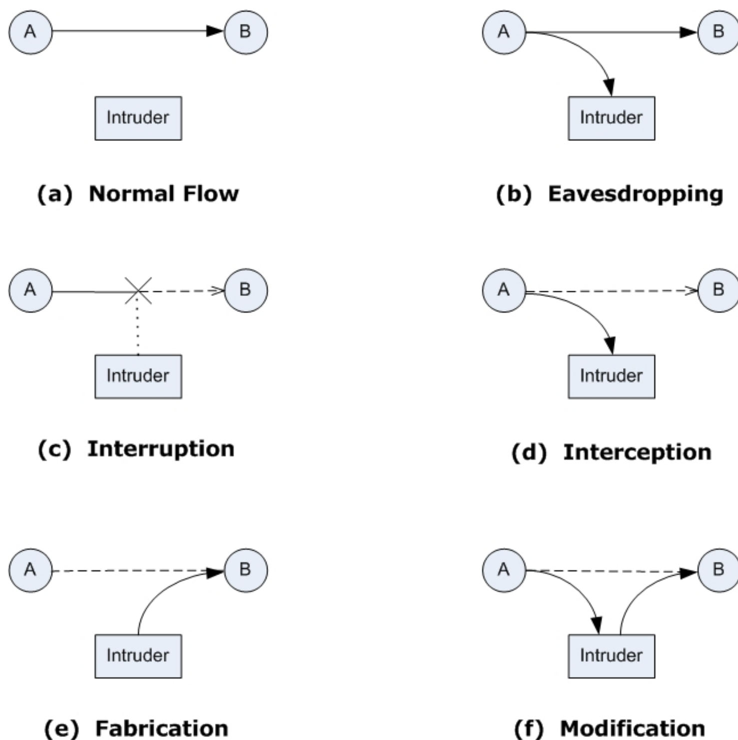

Figure 2.1: Possible Attacks

Eavesdropping

The spy may listen to the messages that are sent from Alice to Bob. In this case, the spy does not modify or block the message, and he does not send any message to the agents involved in the protocol. Since the spy doest not directly affect their communications, it is called a *passive attack*. This kind of attacks is difficult for the honest agents to detect. As the spy keeps listening and collecting information, he may gain some significant information by analyzing the traffic flow.

Interruption

The spy may block the message that is sent from Alice to Bob, but did not get the information or introduce new messages to the agents. As this kind of attack affects the normal flow between agents, *interruption* is among the so-called *active attacks*. Types of possible active attacks could be various, which will be

exemplified in the following cases.

Interception

The spy may thieve the message that is sent from Alice to Bob. That is, the spy obtains the information, while Bob does not receive it. Similarly, it is also an active attack, because the spy breaks the normal communications.

Fabrication

The spy may pretend to be Alice and send fake message to Bob. The spy attempts to make Bob believe that the massage was originated with Alice, while Alice knows nothing at all. This is also a kind of active attacks, as the spy performs active interventions. Here the message could be a synthetic message created by the spy or an earlier message that the spy obtained from past traffic. In case the spy sends an earlier message he obtained, this is a commonly called a *replay attack*.

Modification

The spy may intercept the message that is sent from Alice to Bob, modify the original information, and send the new message to Bob instead. This attack can be considered as a combination of interception and fabrication. In this case, the message was altered by the spy, but neither the sender nor the receiver has any knowledge about this modification. Obviously, this is an active attack as well.

2.2 Security Goals

On insecure networks, several security goals are expected for secure communications. In general, the main security properties for protocols are concluded as follows.

Confidentiality

Confidentiality means that unauthorized parities can not access the secret information. This would be the most obvious and straightforward security property. To achieve this goal, secret information should not be transmitted over the net-

work in plain text, instead it must be protected. *Cryptography* is the common technique to keep data secret.

Integrity

Integrity means that unauthorized parties can not modify the messages. As an agent receives a message, it ensures that the message was not altered during the transmission. Therefore, modification affects the integrity of the message.

Authentication

Authentication means that unauthorized parties can not pretend to be anyone else. As an agent receives a message, it guarantees that the message is really originated with the agent as indicated. And the sender of a message can guarantee that he is really communicating with the expected agent. Thus, fabrication affects the authentication of the message.

Availability

Availability means that the message should be accessible to authorized parties at appropriate times [29]. In other words, for those authorized parities, their legitimate access to the information should not be denied or blocked. Interruption and interception affect the availability of the message. However, the *denial-of-service* attack is beyond the scope of this thesis project.

Non-repudiation

Non-repudiation means that the agents should agree on what they have done. The sender of a message should not be able to falsely deny that he sent the message [31].

2.3 Cryptography

Cryptography is the universal technique to protect data from reading, modification and fabrication. The plaintext message is encrypted in some way and then becomes unreadable *ciphertext*. And the reverse process, decryption, turns the ciphertext back into the original plaintext. In modern cryptography, the algorithms for encryption and decryption always employ *keys*. This is shown in Figure 2.2.

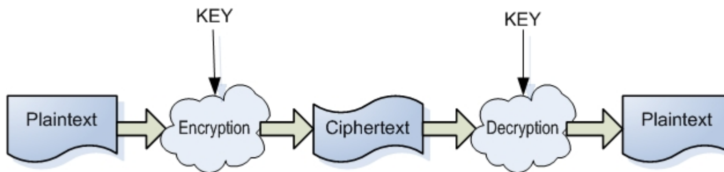

Figure 2.2: Encryption and Decryption with Keys

For the encryption process, the original plaintext, the encryption algorithm and the key value determine the resulting ciphertext altogether. As the algorithm is always open and published, the security concerns turn to the secrecy of keys.

Sometimes, the keys used in encryption and decryption are the same ones, or the two keys are not same but can be calculated form each other. Such kind of cryptography is *symmetric*. In most symmetric cases, the encryption key and the decryption key are the same. This is shown in Figure 2.3.

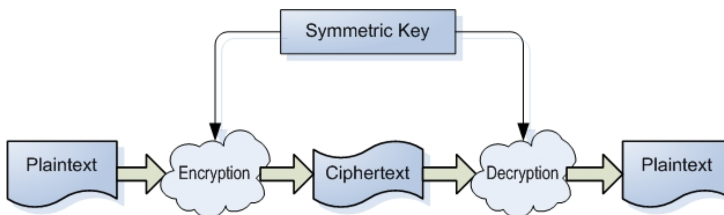

Figure 2.3: Symmetric Cryptography

Another type of key-based cryptography is *asymmetric cryptography*. This means the encryption key and decryption key are different keys, and they are not able to be calculated from each other. This is shown in Figure 2.4.

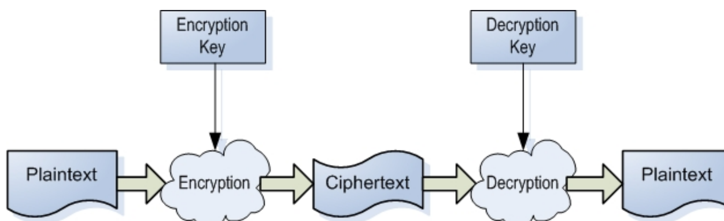

Figure 2.4: Symmetric Cryptography

2.4 Cryptographic Protocols

A *protocol* is a series of steps, involving two or more parties, designed to accomplish a task [31]. The protocol using cryptography is called *cryptographic protocol*. In common cryptographic protocols, each conversation is encrypted with a unique key. The key is only used for the certain communication session, and it is only valid for a short-term, hence the name *session keys* or *short-term keys*. Normally, session keys are always symmetric keys. Suppose that Alice and Bob want to communicate on a network. They have to both get hold of a particular session key and agree on it before sending secret informations. Secure exchange of secret session keys and mutual authentication of participants could be complicated issues, and they are just the two main problems for protocol designers to solve.

2.4.1 Shared-Key Protocol

A *shared-key protocol*, also called a *symmetric key protocol*, requires that each participant on the network shares a unique secret key with a trusted third party, the *authentication server*, before the protocol runs. This shared secret key is long-term and symmetric. The protocol assumes that the shared keys are already distributed properly and securely. The long-term keys are only used to encrypt the exchange of session keys but not to encrypt any actual message between participants.

A typical example is the Needham-Schroeder shared-key protocol [23] (see Figure 2.5).

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

Figure 2.5: The Needham-Schroeder Shared-key Protocol

In this thesis, protocols are represented in the notation used above. Suppose that the protocol session is initiated by a participant A and responded by B. Ka is the long-term secret key of A, which is shared with the server S, and Kb is B's long-term key. Kab stands for the session key. The notation $\{X\}_K$

indicates the resulting ciphertext for encrypting message X by key K. Then the protocol steps can be interpreted as follows.

(1) A generates a fresh *nonce* Na, and send it to the server with his own name and B's name. Generally, the *nonce* is a random number generated by the protocol participant. Since the value of nonce is random and unguessable, it is used to identify a unique protocol session, in order to prevent replay attacks.

(2) As A's request arrives, the server issues a fresh session key Kab. Since the server has Ka and Kb, he encrypts the session key and A's name with Kb, then uses Ka to encrypt this encrypted message together with A's nonce, B's name and the session key Kab. The resulting message is sent back to A.

(3) A can decrypt the outer layer of the message by his own key and obtain the session key. He accepts this session key if the nonce Na which arrives in this step is the same as the one generated in step (1). The message $\{Kab, A\}_{Kb}$ is not readable by A, so A just forward it to B.

(4) B decrypts the message by his own key and gets the session key Kab. Then B generates another fresh nonce Nb, encrypts Nb with the session key Kab, and send the encrypted message to A.

(5) A decrypts the message with the session key that he obtained and accepted in step (3). Then A calculates $Nb - 1$, encrypts the result with Kab, and sent it back to B.

(6) This is the implicit last step. B decrypts the message with the session key and then verifies $Nb - 1$. If it is satisfied, B will accept the session key Kab and use it to communicate with A.

In this protocol, nonces are employed to prevent replay attacks. A generates a nonce Na in step (1), and then he can verify the nonce he receives in step (2) to ensure that this response is freshly originated with the server and not a replayed one from old sessions. Similarly, by the reception of $Nb-1$ in step (5), B can verify that the message is sent by A and not a replay from previous runs of the protocol. However, there is still a possible replay attack on the Needham-Schroeder shared-key protocol. This weakness will be discussed in section 2.5.2.

2.4.2 Public-Key Protocol

The *asymmetric key protocol* uses asymmetric cryptography to exchange and agree on a session key. It is also known as the *public-key protocol*. For each agent on the network, he owns a pair of keys for encryption and decryption separately, containing a *public key* and a *private key*. That is, when one key of the pair is used to encrypt a message, the other key is the only decryption key for that ciphertext. Each agent has his own private key and keeps it secret from other agents, while the public keys are published and known by the world.

A simple example for public-key protocols is the Needham-Schroeder public-key protocol [23](see Figure 2.6).

(1) $A \rightarrow B : \{A, Na\}_{Kb}$
(2) $B \rightarrow A : \{Nb, Na\}_{Ka}$
(3) $A \rightarrow B : \{Nb\}_{Kb}$

Figure 2.6: The Needham-Schroeder Public-key Protocol

Similarly, we assume that the public keys have been distributed in some way before the protocol starts. That is, Alice and Bob already have the public keys of each other. So the steps for public key distribution are irrelevant and thus omitted. Here, Ka stands for A's public key and Kb is B's public key.

2.5 Attacks against Protocols

2.5.1 Assumptions

Security issues on cryptographic protocols can be related to the cryptographic algorithm and techniques, or concern the protocols themselves. In this thesis, we focus on the verification of security properties of protocols. Therefore we assume the cryptography itself is secure enough. This means, we are using perfect encryption that the following properties hold.

- Given an encrypted message $\{M\}_K$, the attacker without key K cannot get the original message M.
- Given an encrypted message $\{M\}_K$, the attacker without key K cannot transform $\{M\}_K$ into $\{M'\}_K$ for any expected M'.

- By analyzing a series of encrypted messages $\{M_1\}_K$, $\{M_2\}_K$, ..., $\{M_n\}_K$, the attacker is not able to find key K and any plaintext message M_i.

On the other hand, in order to detect potential flaws of a protocol, we should assume that the adversary is not only a passive eavesdropper but also an active attacker with enough power. The capability of the adversary should include all of the following possible cases.

- The adversary can act as a passive eavesdropper and observe all the messages sent over the network.
- For all the messages sent over the network, the adversary can intercept them and attempt to modify the messages using all of his knowledge. Modified messages can be sent to the receiver instead, or re-directed to any other agents.
- The adversary can fabricate new messages using all of his knowledge.
- The adversary may get hold of *old* session keys from past protocol sessions.
- The adversary could be a corrupt insider (a legitimate protocol participant) or an outsider (an external party) or a combination of both [24].

2.5.2 An Example

Consider the Needham-Schroeder shared-key protocol that has been described in section 2.4.1 . A possible replay attack on this protocol was revealed by Denning and Sacco [11] afterwards. The spy can listen to the messages in step (3) for each protocol execution and store them. As long as the spy gets hold of an old session key, he can pretend to be A, and convince B to accept this old and compromised session key by the follow steps.

(3') The spy may intervene in the current execution by intercepting the message from A to B in step (3). Then he can impersonate A and replay an old message from a previous session in which the corresponding old session key K' is known by the spy.
$Spy(A) \rightarrow B : \{K', A\}_{Kb}$

(4') B decrypts the message and obtains the compromised key K'. Then he generates Nb, encrypts it with K' and sends it to A.
$B \rightarrow Spy(A) : \{Nb\}_{K'}$

(5') The spy intercepts this message, decrypts it to get Nb and sends encrypted $Nb - 1$ to B.

$$Spy(A) \rightarrow B : \{Nb - 1\}_{K'}$$

(6') By checking $Nb - 1$, B is convinced that he is talking with A and accepts the old key K' as fresh.

Here and henceforth, the notation $Spy(A)$ is used to denote the Spy imitating A. The above steps show how the replay attack against Needham-Schroeder shared-key protocol achieves. To fix this weakness, timestamps can be used to enhance this protocol, suggested by Denning and Sacco [11]. This will be discussed in section 4.1.

2.6 Protocol Analysis

2.6.1 Informal Reasoning

A protocol is proposed by its designer and claimed to hold the expected security properties. However, the protocol may still contain potential subtle flaws. The proposed protocol is then analyzed by others to check if it is really secure as it was claimed. In the early days, cryptographic protocols were analyzed and tested in informal ways. An ideal instance is a thorough and complete test that checks all possible paths and status for the protocol without any omission. Theoretically, it could find every potential flaw at last. However, this is impractical for most cases. Alternatively, people set a limited number of test cases to check. In this way, a potential flaw can be found only if certain test case is wisely included. Otherwise, it is failed to detect the flaw, while the flaw does exist.

As cryptographic protocols are more and more widely used, there is stronger demand to discover those potential flaws. Although informal analysis becomes more and more conscientious, it may still miss subtle but not trivial flaws. Some literature reported their findings in several well-known protocols. Most of these flaws and weakness are discovered after several years since the protocol was proposed.

- The three-messages protocol in the CCITT.X.509 standard was proposed in 1987 [10]. A parallel session attack was presented after 2 years, by Burrows, Abadi and Needham in 1989 [9].

- Needham-Schroeder shared-key protocol was proposed in 1978 [23]. A possible replay attack on this protocol was reported by Denning and Sacco after 3 years in 1981 [11].

- Denning-Sacco public-key protocol was proposed in 1981 [11]. Abadi and Needham detected a possible masquerade attack after 13 years in 1994 [3].

- Needham-Schroeder public-key protocol was proposed in 1978 [23]. After 17 years, a man-in-the-middle attack on this protocol was discovered finally, by Lowe in 1995 [17].

These are only some of the examples. Why they take years? The main reason is that the security goals are informally stated and poorly understood [14]. Confidentiality is the most straightforward security property, but it is not the only goal. It is also very important to authenticate both sides of the communication. Another important reason is that the attacker can be active and powerful. Attack can affect the protocol in various ways, while they could act as a protocol participant or an external party or even a combination of the both. This makes the cases very complicated for informal reasoning. Thus, subtle flaws and weaknesses may survive in some informal analysis.

2.6.2 Formal Approaches

Formal verification can significantly help to detect protocol flaws and can increase our understanding of a protocol by making essential properties explicit [27]. It can also help to yield general principles of secure protocol design [5]. Several formal methods have been proposed and already used for years in the analysis of cryptographic protocols. Some survey papers [19][20] gave detailed introduction to the formal approaches and related researches. In this section, we only refer to some of the major approaches in this field besides Paulson's inductive approach.

BAN Logic

BAN logic [9], proposed by Burrows, Abadi and Needham, is well-known and have been widely used. It provides statements to idealize protocols into initial logic formulae. Some examples of the BAN logic statements are as follows.

P said X means the principal P sent a message containing X.

P sees X means the principal P receives a message containing X, and X is readable to P.

P believes X means the principal P acts as if X is true.

X is fresh means X has not appeared in any message at any time before the current session of the protocol.

BAN logic also provides several inference rules to apply to the logic formulae for reasoning about beliefs of a protocol. A set of beliefs are then derived from the initial logic formulae. The protocol is considered to be correct if the set of beliefs is adequate based on some predefined notions.

BAN logic has been used for years as a popular formal approach for verifying cryptographic protocols. It successfully detected some potential flaws for several protocols after they have been proposed for years. However, some protocols that passed the verification by BAN logic were proposed to contain flaws. One example is a faulty variant of Otway-Rees protocol that the nonce of the responder is not encrypted, which is exemplified in [26] and [27]. However, BAN logic is not adequate to detect its fault [27]. Another example is the Needham-Schroeder public-key protocol. It has been proved correct using BAN logic [9], but a possible attack against this protocol was then reported by Lowe in [17].

BAN logic is designed for reasoning about the evolution of the belief and trust of the participants in a cryptographic protocol [32]. As BAN logic models beliefs rather than knowledge, it can deal with authentication goals, while it is weak at reasoning about confidentiality. And it dose not clearly formulate the possible actions of an attacker. In spite of its inadequacy, BAN logic is still a simple and usable approach to detect some of the flaws. The logic is straightforward, and the proof is normally short and handmade, where machine proof is not required.

Model Checking

Model checking approaches model a cryptographic protocol as a finite state machine, and verify the protocol by checking whether certain properties hold for all reachable states. The checking process could be automatically performed by some applications, known as the *model checker*. On the other side, model checking is restricted by the size of protocol. As the technique is based on state machine, it can only deal with simple protocols of small size.

Most versions of the model checking approach are developed based on the Dolev-Yao Model [12]. One of the successful examples is the NRL Protocol Analyzer [21], which specifies an insecure state and tries all possibilities to find a path that could start from the initial state and reach the insecure state. The protocol is claimed to be insecure if such a path can be found. Another well-known model checker is FDR [30]. In this method, the protocol and the property are separately specified using CSP [15]. The two sets of traces are then examined to check whether the protocol process is a subset of the property set, in order to verify the protocol.

The Lysa-Calculi

Lysa [8] is a process calculus designed for analysis of cryptographic protocols. This is a more recent approach compared with the aforementioned ones. Lysa is patterned after Spi-calculus [2] which is also used for analysis of security protocols. Spi-calculus is based on the Π-calculus [22], but provides support for cryptography related operations. The derivation is illustrated in Figure 2.7.

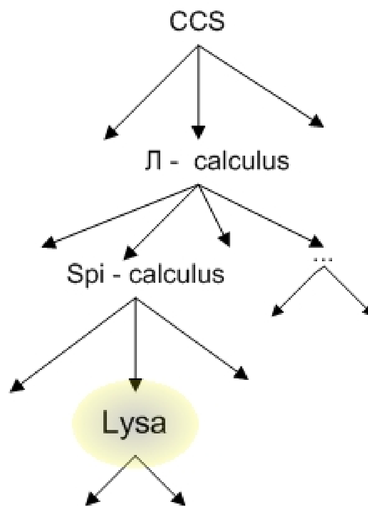

Figure 2.7: Process Calculus [24]

Like most process calculus, Spi-calculus uses channels to enable communications between processes. However, the derived Lysa-calculi does not retain the concept of channel, which would be considered as the most significant difference

from Spi-calculus on syntax. Compared with the aforementioned calculus, Lysa is tiny but much powerful for modelling security protocols, and subject to automatic analysis [24].

This approach formalizes the protocol as Lysa-process, then apply static analysis to verify the protocol. The static analysis was adjusted to be less complicated while making efforts to provide much useful information. Lysa is mainly focused on verification of authentication. It was demonstrated that this approach is adequate for detecting several authentication flaws in both shared-key and public-key cryptographic protocols [8]. Developing on Lysa-calculus is still in progress. It has been proposed to be extended in future for several purposes. However, Lysa-calculus, as well as Spi-calculus, is relatively difficult to grasp. In addition, since time is not representable in Lysa, this approach cannot be used to analyze those protocols with timestamps.

Chapter 3

The Inductive Approach

This chapter introduces Paulson's inductive approach [26] which is adopted for protocol verification in this thesis project. The protocol is formally specified using operational semantics. The protocol model is defined with induction. A *trace* is a list of events that have been taken place on a network system running the protocol. And each possible action by an agent extends the event trace. The protocol is then formalized as a set of traces involving all possible traces. Security properties are also formally specified and proved by provided induction rules. The proof is aided by the interactive theorem proof assistant Isabelle [25].

All the Isabelle theory files for verifying cryptographic protocols can be found in the folder `Auth` [1] at `Isabelle/HOL/Auth`. The theory dependencies for the main part of this division are illustrated in Figure 3.1. Generally, basics for verifying a cryptographic protocol are involved in three theories named `Message`, `Event` and `Public`.

In this chapter, we introduce the basic types (§ 3.1) and operators (§ 3.3) for the inductive approach, as well as the definitions of agent's knowledge (§ 3.2).

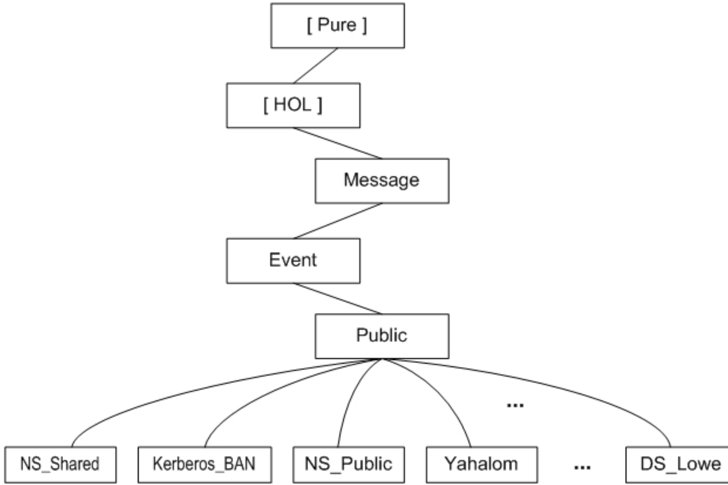

Figure 3.1: Theory Dependencies of `Auth` (part)

3.1 Basics

Before defining messages in cryptographic protocols, a free type key of cryptographic keys has to be defined.

```
types key = nat
consts invKey ::  "key => key"
```

A key is a natural number. The function invKey maps an encryption key to the corresponding decryption key, and vice versa. That is to say, for asymmetric keys, the inverse of a public key is the corresponding private key, and vice versa. And for symmetric keys, we have $K^{-1} = K$. The set of symmetric keys is defined as symKeys.

```
constdefs
   symKeys ::  "key set"
   "symKeys == K. invKey K = K"
```

Because the protocols we discuss and analyze in this thesis are shared key protocols, a symmetric key setting is assumed for the follow part of the thesis. Since then, each agent shares a long-term symmetric key with the server. The function shrK is defined to map an agent name to the corresponding shared key.

```
consts shrK ::  "agent => key"
```

Obviously all shared keys are symmetric. And it is assumed that no two agents can correspond to the same shared key. In the symmetric key environment, every cryptographic key is either a long-term shared key or a short-term session key.

The Isabelle datatype definition is used to define three basic types for modelling a cryptographic protocol, namely agent, message and event.

3.1.1 Agent

The datatype agent represents all the principals on the network. The definition is shown below.

```
datatype agent = Server | Friend nat | Spy
```

One of the agents is the Server, which is a trusted third party required by most key distribution protocols. The Spy is the malicious agent on the network, that is, the active attacker. Since nat is the type of natural numbers, any number of friendly agents is allowed. These friendly agents are indexed by natural numbers.

An agent is *compromised* if his own shared key is already perceived by the spy from the start of the protocol. The set of compromised agents is defined as bad. In particular, the spy belongs to the set bad, since it knows his own key. But server is not in bad, because it is set to be secure.

```
consts bad ::  "agent set"
specification (bad)
   Spy_in_bad [iff]:  "Spy ∈ bad"
   Server_not_bad [iff]:  "Server ∉ bad"
```

3.1.2 Message

Then the form of messages in cryptographic protocol is defined as datatype msg, which introduces seven constructors.

```
datatype msg = Agent agent
```

```
             | Number nat
             | Nonce nat
             | Key key
             | Hash msg
             | MPair msg msg
             | Crypt key msg
```

A message can be an agent's name, an ordinary number, an unguessable nonce, a cryptographic key, a hashed message, an encrypted message or a pair of message. Compound message is recursively defined. The notation $\{|X_1, \ldots, X_{n-1}, X_n|\}$ expresses a compound message, as an abbreviated form for MPair X_1 (. . . (MPair X_{n-1} X_n)).

CryptKX expresses the ciphertext that the message X is encrypted by key K. In respect that constructors of datatype are injective, we have the theorem

$$\text{Crypt } K\ X = \text{Crypt } K'\ X' \Longrightarrow K = K' \ \&\ X = X'.$$

That is to say, a ciphertext can only be decrypted by only one certain key, and the resulting plaintext is unique. Moreover, as we have discussed in setion 2.5.1, the encryption is assumed to be perfect that encrypted messages are not able to be read or transformed by any agent unless he has the right key. Moreover, it is assumed that the type of each message component is clearly specified, so it is impossible to confuse different types of message components.

Number represents the guessable natural number, which is normally used for modelling timestamps. It was later added [6] in order to apply the inductive approach to protocols with timestamps.

3.1.3 Event

In the inductive approach, the history of actions on the network is specified as a *trace* of events. Then, an *event* is a single action that forms the trace. Sending message is the most important and straightforward event, but it is not everything. The datatype event is defined as follows, which contains three constructors.

```
datatype event = Says agent agent msg
             | Gets agent msg
             | Notes agent msg
```

Obviously, the constructor Says expresses sending a messages. Says A B X states that A attempts to send a message X to B.

The Gets constructor, explicitly expressing message reception, is an extension to event by G.Bella [5] in order to model agent's knowledge via message reception. Gets B X expresses B's reception of a message X. It only indicates that the message was previously sent, but B does not have any knowledge about who is the sender. In Paulson's work [26], this case is expressed as Says A' B X without the Gets constructor. In this way, the sender is stated as A' which is not used elsewhere, because B cannot get to know the real sender of the message X. However, the extension of Gets can improve the readability of the specifications of protocols and security properties.

Notes A X expresses that the agent A internally stores X which is a part of the message he received. This means the event is only known by A himself, as long as A is uncompromised. Otherwise, if A is a bad agent, the event Notes A X is also visible to the spy.

3.1.4 Event Traces

As mentioned above, the history of agents' actions is formalized as a list of events, namely a *trace*. When an event has taken place, the existing trace is then extended with the new event. A trace is a list of events in reverse order. So the new event is inserted to the head of the list. In Isabelle syntax, $ev\#evs$ expresses the trace evs is extended with the new event ev.

Moreover, given a trace evs, set evs represents the set of all events in the trace, that is, all the events that have occurred. According to Isabelle's Logic, the function set maps a list to the set that consists of all elements from the list. Therefore, given $ev \in$ set evs, it is indicated that the event ev has taken place on trace evs.

3.2 Agent's Knowledge

In order to model agents' knowledge inductively, the initial knowledge of agents is required to be specified.

3.2.1 Initial Knowledge

To specify the initial knowledge of agents before the protocol starts, the function initState is declared as follows.

```
consts initState ::  "agent => msg set"
```

The function initState maps an agent's name to the set of messages that are known by the agent before the running of the protocol. So the notation initState A expresses the initial knowledge of the agent A. In a symmetric key setting, for each type of agents, the initial knowledge of the agent is defined respectively as follows.

```
primrec
initState_Server:
   "initState Server = (Key ‘ range shrK)"
initState_Friend:
   "initState (Friend i) = Key (shrK(Friend i))"
initState_Spy:
   "initState Spy = (Key ‘ shrK ‘ bad)
```

According to Isabelle's logics, a function's range is the set of values that the function can take on, in other word, the image of the universal set under that function. Therefore, range shrK represents the set of all shared keys. And the notation `f ‘` A represents the image of the set A under the function `f` [25]. In this way, the above definition of agents' initial knowledge turns to be clear. The server's initial knowledge contains the shared keys of all the agents on the network, that is, all long-term keys on the network. The initial knowledge of a friendly agent is his own long-term key that is shared with the server. And the spy's initial knowledge contains shared keys of all compromised agents, including his own shared key, of course.

3.2.2 Modelling Agent's Knowledge

The function knows is introduced to describe an agent's knowledge on some trace.

```
consts knows ::  "agent => event list => msg set"
```

The notation knows A evs represents the set of message that the agent A can

obtain based on the trace *evs*. This knowledge is defined as follows.

```
primrec
  knows_Nil:  "knows A [] = initState A"
  knows_Cons:
    "knows A (ev # evs) =
      (if A = Spy then
        (case ev of
          Says A' B X => insert X (knows Spy evs)
        | Gets A' X => knows Spy evs
        | Notes A' X =>
           if A'∈bad then insert X (knows Spy evs) else knows Spy evs)
      else
        (case ev of
          Says A' B X =>
           if A'=A then insert X (knows A evs) else knows A evs
        | Gets A' X =>
           if A'=A then insert X (knows A evs) else knows A evs
        | Notes A' X =>
           if A'=A then insert X (knows A evs) else knows A evs))"
```

- At the beginning, any agent's knowledge is just his initial knowledge.
- About the spy's knowledge, he knows every message sent over the network no matter who is the sender or the receiver. And the spy also knows what is noted by every compromised agent. Notice that a Gets event for message reception does not extend the spy's knowledge because the Says event for sending that message did so already.
- For an agent other than the spy, he knows every message sent by himself, and he knows every message he received. Every note by that agent on the trace is also included in the agent's knowledge.

The modelling of agents' knowledge was introduced by G. Bella [5]. In Paulson's original version [26] of the inductive approach, only the spy's knowledge was modelled. A function spies is used to specify the spy's knowledge. The notation spies *evs* represents the set of message that the spy can see based on the trace *evs*.

$$\begin{aligned}
&\mathsf{spies}[\,] \triangleq \mathsf{initState\ Spy}\\
&\mathsf{spies}((\mathsf{Says}\,A\,B\,X)\#evs) \triangleq \{X\} \cup \mathsf{spies}\,evs\\
&\mathsf{spies}((\mathsf{Notes}\,A\,X)\#evs) \triangleq \begin{cases} \{X\} \cup \mathsf{spies}\,evs & \text{if } A \in \mathsf{bad}\\ \mathsf{spies}\,evs & \text{otherwise}\end{cases}
\end{aligned}$$

In the current logic library of Isabelle 2005 [1], the basic theories for protocol verification have already been updated with G. Bella's extension [5] of agents' knowledge and message reception. However, the use of spies is still retained for compatibility, as the syntax for spies stays in the theory. For simple key-distribution protocols, Paulson's original syntax would be sufficient for analysis of the security properties. In this thesis, we adopt the original syntax in modelling and analyzing the protocol, and then update the protocol model with message reception and agent's knowledge.

3.3 Operators

Operators express the operations on messages. These operations are used to describe the capabilities of attackers and specify the security properties. The following three operators are all defined inductively. Each of them maps a set of messages to another set of messages.

3.3.1 The Function parts

Suppose H is a set of messages. Then parts H is inductively defined as follows.

```
consts parts ::  "msg set => msg set"
inductive "parts H"
   intros
      Inj [intro]:  "X ∈ H ==> X ∈ parts H"
      Fst:  "{|X,Y|} ∈ parts H ==> X ∈ parts H"
      Snd:  "{|X,Y|} ∈ parts H ==> Y ∈ parts H"
      Body:  "Crypt K X ∈ parts H ==> X ∈ parts H"
```

According to the above definition, parts H consist of every part of each message in H. Formally speaking, for any form of message X, the parts of message X contain the message itself; for a compound message, the parts of the message contain all the elements that form the compound message; for an encrypted message Crypt K X, the parts of the message contain the decrypted plaintext X, while the key K does not belong to parts of the message unless K appears in X.

Basic lemmas about parts can be derived from the definition. Two straightforward examples are its idempotence and monotonicity. The operator parts is idempotent, because we have

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

And parts is monotonic since we have

$$G \subseteq H \Longrightarrow \mathsf{parts}\, G \subseteq \mathsf{parts}\, H$$

3.3.2 The Function analz

The function analz is used to describe what the spy can obtain by analyzing a set of messages. The inductive definition of analz is shown below.

```
consts analz ::  "msg set => msg set"
inductive "analz H"
 intros
  Inj [intro,simp] :  "X ∈ H ==> X ∈ analz H"
  Fst:  "{|X,Y|} ∈ analz H ==> X ∈ analz H"
  Snd:  "{|X,Y|} ∈ analz H ==> Y ∈ analz H"
  Decrypt [dest]:
   "[|Crypt K X ∈analz H; Key(invKey K): analz H|] ==> X∈analz H"
```

The definition is similar with parts, the only difference is the last rule for encrypted message. For an encrypted message Crypt K X in set H, the decrypted plaintext X is included in analz H only if the appropriate decryption key is available in analz H. For symmetric key settings, the decryption key is the same as the encryption key, namely, $\mathsf{Key}(\mathsf{invKey}\, K) = \mathsf{Key}\, K$.

Similarly, the operator analz is idempotent and monotonic.

$$\mathsf{analz}(\mathsf{analz}\, H) = \mathsf{analz}\, H.$$
$$G \subseteq H \Longrightarrow \mathsf{analz}\, G \subseteq \mathsf{analz}\, H$$

Further lemmas between the two operators can be easily derived from their definitions. Here are some examples.

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

3.3.3 The Function synth

The operator synth is used to describe what the spy can build up by synthesizing a set of messages. It is defined in Isabelle syntax as follows.

```
consts synth ::  "msg set => msg set"
inductive "synth H"
  intros
   Inj [intro]:  "X ∈ H ==> X ∈ synth H"
   Agent [intro]:  "Agent agt ∈ synth H"
   Number [intro]:  "Number n ∈ synth H"
   Hash [intro]:  "X∈synth H ==> Hash X∈synth H"
   MPair [intro]:  "[|X∈synth H; Y∈synth H|] ==> {|X,Y|}∈synth H"
   Crypt [intro]:  "[|X∈synth H; Key(K)∈H|] ==> Crypt K X∈synth H"
```

According to the above definition, synth H contains all its element messages, all the agent names and any guessable numbers. Because nonces and keys are assumed to be not guessable, they are then not included in synth H, except for those already in H. Besides, available messages in synth H can be hashed, combined to form pairs, and can be encrypted using available keys in H.

Similarly, this operator synth is also idempotent and monotonic.

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

Among the above three operators, there are nine possible combination of two operators. An notable combination is synth $\circ$ analz. Recall the notion of spy's knowledge, then analz(knows Spy evs) specifies the set of messages that the spy can extract from his knowledge, and synth(analz(knows Spy evs)) specifies the set of fake messages that the spy can fabricate based on his knowledge.

3.4 Other Useful Functions

3.4.1 The Function used

A necessary operator used is required to express freshness. The function maps a trace of events to the set of all message components mentioned in the trace and in all agents' initial knowledge. The definition in Isabelle syntax is shown below.

```
consts used ::  "event list => msg set"
primrec
   used_Nil:  "used [] = (⋃ B. parts (initState B))"
   used_Cons:  "used (ev # evs) =
                      (case ev of
                         Says A B X => parts {X} ∪ used evs
                       | Gets A X => used evs
                       | Notes A X => parts {X} ∪ used evs)"
```

According to this definition, used evs includes parts of all initial knowledge by every agent and parts of all past messages on the trace evs, namely, parts(knows Spy evs). Thus, a message X is considered to be fresh on trace evs, if X is not in the set used evs.

In particular, this operator is much useful in modelling cryptographic protocols with nonce. It is used to formalize the agent's behavior that an agent generates a fresh nonce. In this way, Nonce $N \notin$ used evs specifies the freshness of nonce N on trace evs.

3.4.2 The Function keysFor

Similar as the aforementioned operators, the function keysFor also expresses operation on messages. This function maps a set of messages to the set of keys that can be used to decrypt any message in the set. Its definition is simple and straightforward, as follows.

```
constdefs
   keysFor ::  "msg set => key set"
   "keysFor H == invKey ‘ K. ∃X. Crypt K X ∈ H"
```

For shared-key protocols, all the keys are symmetric, no matter long-term keys or short-term keys. In this setting, given a message set H, keysFor H represents the set of keys that were used to seal those encrypted messages in H. It is mostly used to state the lemma `new_keys_not_used` (ses section 5.1) in the form of keysFor (parts (spies evs)) , which represents the set of keys that have been used to encrypt any message components that appear on traffic.

Chapter 4

Modelling the Protocol

This chapter presents how a cryptographic protocol is modelled by Paulson's inductive approach. We apply the approach to the modified version of Denning-Sacco shared-key protocol [11] proposed by Gavin Lowe [18]. This protocol uses symmetric key cryptography to encrypt messages and employs both nonce and timestamps to guarantee freshness (§ 4.1). At the beginning, modelling of timestamps was not contained in Paulson's original inductive approach [26]. It was proposed when BAN Kerberos was verified by the inductive approach [6]. We introduces the method for modelling timestamps (§ 4.2), and then describes how Lowe's modified Dennning-Sacco shared key protocol is modelled and formally specified in Isabelle (§ 4.3). We also include some discussions about the inductive model of a protocol (§ 4.4). The protocol is first modelled following Paulson's original approach [26] (§ 4.3). Then the model is updated with Bella's extensions of message reception and agent's knowledges [5] (§ 4.5).

4.1 Lowe's Modified Denning-Sacco Shared-key Protocol

Consider the Needham-Schroeder shared-key protocol [23] (see section 2.4.1). As mentioned above, it has weakness that replay attacks against this protocol is possible if old session keys may be compromised occasionally (see section 2.5.2). Considering the possibility that the session keys may be lost, Denning and Sacco [11] proposed to employ timestamps in key distribution protocols in order to prevent replays of compromised old session keys. In this way, they proposed a solution to enhance Needham-Schroeder shared-key protocol using timestamps. Their enhanced version with timestamps is thus.

(1) $A \rightarrow S : A, B$
(2) $S \rightarrow A : \{B, Kab, T, \{Kab, A, T\}_{Kb}\}_{Ka}$
(3) $A \rightarrow B : \{Kab, A, T\}_{Kb}$

Figure 4.1: The Denning-Sacco Shared-key Protocol

This is known as the Denning-Sacco shared-key protocol. In step (2), the server marks the message with the timestamp T which gives the current time. This timestamp thus presents the time when session key Kab was issued. A verifies the message by checking that the interval between current time and timestamp T is less than the lifetime of a session key. The timestamp T is then forwarded to B in step (3). Similarly, B verifies the message in the same way. As long as the session key lifetime is adjusted to be less than the time interval since the last protocol execution, the use of timestamp will achieve its goal. Of course, this kind of timestamp-based protocols require a global synchronized clock system.

Denning and Sacco also claimed that, by adding timestamps properly in this way, the last two steps of the Needham-Schroeder shared-key protocol can be replaced. They believed that timestamps have the additional benefit of replacing the two-step handshake [11]. However, Lowe [18] pointed out a subtle flaw in Denning-Sacco shared-key protocol. The possible attack is shown below.

(1) $A \rightarrow S : A, B$
(2) $S \rightarrow A : \{B, Kab, T, \{Kab, A, T\}_{Kb}\}_{Ka}$
(3) $A \rightarrow B : \{Kab, A, T\}_{Kb}$
(3′) $Spy(A) \rightarrow B : \{Kab, A, T\}_{Kb}$

The spy can impersonate A and replay the message of step (3) immediately. In this way, B believes that A is requesting for two sessions with him, while A only set up one session with B. The reason why this attack succeeds is that the timestamps provide only partial authentication of A: they show that A is

currently trying to establish a session, but they don't show how many [18].

Therefore, the nonce handshake of the last two steps in Needham-Schroeder shared-key protocol is still necessary for the timestamp-based version. Then Lowe modified version of Denning-Sacco shared-key protocol is shown in Figure 4.2.

$$
\begin{array}{ll}
(1) & A \rightarrow S : A, B \\
(2) & S \rightarrow A : \{B, Kab, T, \{Kab, A, T\}_{Kb}\}_{Ka} \\
(3) & A \rightarrow B : \{Kab, A, T\}_{Kb} \\
(4) & B \rightarrow A : \{Nb\}_{Kab} \\
(5) & A \rightarrow B : \{dec(Nb)\}_{Kab}
\end{array}
$$

Figure 4.2: The Lowe's Modified Denning-Sacco Shared-key Protocol

Generally, a session key is considered *expired* if the interval between current time and the issue time of the key is longer than the session key's lifetime. Particularly in this protocol, the timestamp T stands for the issue time of session key Kab. A checks the timestamp T he received by step (2), and he will send the message of step (3) only if the session key Kab issued at time T is not expired. Similarly, B also verifies the timestamp T before he sends the message of step (4).

Compared with the Needham-Schroeder shared-key protocol, this version would be considered as an extension with timestamps. It can solve the weakness in Needham-Schroeder's version, and avoid the multiplicity attack on Denning-Sacco's original version. In this thesis, we will formalize and analyze this protocol using the inductive approach.

4.2 Modelling Timestamps

To model timestamp-based cryptographic protocols using inductive approach, it is assumed that there is a network-wide accurate clock for all the agents. As mentioned above, in the inductive approach, a cryptographic protocol is formalized as a set of traces. Each trace is a list of events that present a possible history of the occurred events on network. Therefore, Bella and Paulson [6] define current time as the current length of a trace. In this way, the current time of an empty trace is zero, and the current time of a trace including n events is n. This formalization is simple and straightforward, and also sufficient to express the time as the clock in the formal protocol model.

Timestamps are then modelled as natural numbers, and their values are possible to guess. The constructor Number defined in datatype msg was introduced for timestamps (see section 3.1.2). Thus, a timestamp T is formally specified as Number T in a protocol model. Since it is guessable, the spy should be able to synthesize timestamps. So the definition of operator synth contains Number n $\in$ synth H.(see section 3.3.3)

The function CT maps a trace to the current time on that trace, which is, in fact, the length of the list of events. The definition below is included in our theory.

```
syntax CT ::  "event list => nat"
translations "CT" == "length "
```

The natural number SesKeyLife is defined to represent the lifetime of session keys.

```
consts SesKeyLife ::  nat
```

Agents check the timestamps in the messages they received. The messages with fresh timestamps are accepted, and the messages with expired timestamps are dropped. This behaviour of checking is modelled by the function Expired, which states whether the session key is expired on a given trace. Its definition is shown below.

```
syntax Expired ::  "[nat, event list] => bool"
translations "Expired T evs" == "SesKeyLife + T < CT evs"
```

Formally speaking, if Expired T evs is ture, it expresses that the interval between timestamp T and current time of trace evs is longer than the valid life of this timpstamp. In other words, it states the timestamp T is expired on trace evs. Since, in this protocol, the only timestamp is used to record the time of issue of the session key, the predicate Expired T evs expresses that the session key issued at time T is expired on trace evs.

4.3 Formalizing the Protocol by Induction

4.3.1 Inductive Rules

As mentioned above, the protocol is formalized as a set of all possible traces. The set of traces is defined by induction. The induction is based on an empty trace of the set. Then each protocol step is formalized as an inductive rule that specify all possible extensions to a given trace with the new event concerning the protocol step. As the protocol we are going to model comprises five steps, there will be five inductive rules in the formal specification. We transcribe the five steps respectively as follows.

(1) If *evs* is a trace of the set, then *evs* may be extended with the event
Says A $Server$ $\{\!|$ Agent A, Agent B $|\!\}$.

(2) If *evs* is a trace containing an event of the form
Says A' $Server$ $\{\!|$ Agent A, Agent B $|\!\}$,
and Kab is a *fresh* symmetric key, then *evs* may be extended with the event
Says $Server$ A $\{\!|$ Agent B, Key KAB, Number Tk,
$\{\!|$ Key KAB, Agent A, Number Tk $|\!\}_{Kb}$ $|\!\}_{Ka}$
where timestamp Tk records the current time on trace *evs*.

Here the server is not able to judge who is the real sender by receiving the message in assumption, so the sender is denoted as A', which is not used elsewhere in the rule.

(3) If *evs* is a trace containing two events
Says A $Server$ $\{\!|$ Agent A, Agent B $|\!\}$ and
Says S A $\{\!|$ Agent B, Key K, Number Tk, X $|\!\}_{Ka}$,
and the timestamp Tk is *not* expired on trace *evs*, and A is and agent distinct from the server, then *evs* may be extended with the event
Says A B X.

Here the message component encrypted with B's key is simply denoted as X because it is unreadable to A. But A may forward this ciphertext to B. Moreover, in the first message in the assumption, the sender's name is just A because A is able to know whether he has sent such a message sometime before. S is used to indicate the sender of the second message because A cannot know who is the real sender of this message.

(4) If *evs* is a trace containing the event of the form
Says A' B $\{\!|$ Key K, Agent A, Number Tk $|\!\}_{Kb}$,

and key K is symmetric, and Tk is *not* expired on evs, and nonce NB is *fresh*, then evs may be extended with the new event
Says B A $\{\!|$ Nonce NB $|\!\}_K$.

Similarly, B cannot know who really send the message, so the sender's name is shown as A'.

(5) If evs is a trace containing the two events
Says B' A $\{\!|$ Nonce NB $|\!\}_K$ and
Says S A $\{\!|$ Agent B, Key K, Number Tk, X $|\!\}_{Ka}$,
and K is a symmetric key, then evs may be extended with the new event
Says A B $\{\!|$ Nonce NB, Nonce NB $|\!\}_K$.

Similarly, here B' and S are used to represent the senders, because A cannot identify the real senders of the two messages he received. Note that it is not necessary to decrease the nonce NB for this step in the inductive model. Instead, we let A to send NB twice. We believe this way provides equivalent effect of the protocol but keeps our modelling easier. It does not require the extension of the spy's capability with decrement. And if the spy is formalized to have the capability of increment and decrement, he would be able to fake any nonce.

In fact, there is still an unspecified last step that B decrypts the message to check NB and then confirms the session. However, the implicit step is not necessary to be formalized for this protocol. Our proof (see chapter 5) shows that the protocol model with the five explicit steps is adequate to prove the mutual authentication. Further discussion on this topic can be found in section 4.4

4.3.2 A Model for the Protocol

For Lowe's modified Denning-Sacco shared-key protocol, we declare a constant ds_lowe as the set of traces that formalizes the protocol. The formal specification of this protocol is shown in Figure 4.3.

```
consts ds_lowe ::  "event list set"
inductive "ds_lowe"
 intros

  Nil:  "[] ∈ ds_lowe"

  Fake:  "[| evsf ∈ ds_lowe; X ∈ synth (analz (spies evsf)) |]
```

```
       ==> Says Spy B X # evsf ∈ ds_lowe"

DS1:  "[| evs1 ∈ ds_lowe |]
       ==> Says A Server {| Agent A, Agent B |} # evs1 ∈ ds_lowe"

DS2:  "[| evs2 ∈ ds_lowe; Key KAB ∉ used evs2; KAB ∈ symKeys;
         Says A' Server {| Agent A, Agent B |} ∈ set evs2 |]
       ==> Says Server A
             (Crypt (shrK A)
               {| Agent B, Key KAB, Number (CT evs2),
                 (Crypt (shrK B)
                   {| Key KAB, Agent A, Number (CT evs2) |}) |})
             # evs2 ∈ ds_lowe"

DS3:  "[| evs3 ∈ ds_lowe; A ≠ Server;
         Says S A (Crypt (shrK A)
             {| Agent B, Key K, Number Tk, X |}) ∈ set evs3;
         Says A Server {| Agent A, Agent B |} ∈ set evs3;
          ~ Expired Tk evs3 |]
       ==> Says A B X # evs3 ∈ ds_lowe"

DS4:  "[| evs4 ∈ ds_lowe; Nonce NB ∉ used evs4;
         K ∈ symKeys;
         Says A' B (Crypt (shrK B)
             {| Key K, Agent A, Number Tk |}) ∈ set evs4;
         ~ Expired Tk evs4 |]
       ==> Says B A (Crypt K (Nonce NB)) # evs4 ∈ ds_lowe"

DS5:  "[| evs5 ∈ ds_lowe; K ∈ symKeys;
         Says B' A (Crypt K (Nonce NB)) ∈ set evs5;
         Says S A (Crypt (shrK A)
                {| Agent B, Key K, Number Tk, X |}) ∈ set evs5 |]
        ==> Says A B (Crypt K {| Nonce NB, Nonce NB |})
             # evs5 ∈ ds_lowe"

Oops:  "[| evso ∈ ds_lowe;
         Says Server A (Crypt (shrK A)
             {| Agent B, Key K, Number Tk, X |}) ∈ set evso;
         Expired Tk evso |]
       ==> Notes Spy {| Number Tk, Key K |} # evso ∈ ds_lowe"
```

Figure 4.3: Specification of Lowe's modified Denning-Sacco Shared-key Protocol in Isabelle

Recall that set *evs* represents the set of all events in the trace. Then an event $ev \in \mathsf{set}\, evs$ indicates that the event *ev* has occurred on trace *evs*. The base case is an empty trace representing the initial state that the protocol has not been executed. The *Nil* rule admits the empty trace. Then all the following rules inductively extend a given trace. The *Fake* rule models the spy's capability that he can send fake messages which are fabricated based on his analysis of past traffics. These two rules are always required in modelling all protocols. The next five rules from *DS1* to *DS5* formalize the five protocol steps as aforementioned.

Note that the last rule *Oops* is indispensable for modelling key-distribution protocols. The *Oops* rule allows the leak of session keys to the spy. In particular, for this timestamp-based protocol, the *Oops* rule can be less permissive that only expired session keys may be compromised. This is reasonable and realistic, because generally the risk of losing a session key increases over time. Here the premise of this *Oops* rule is that the server has issued the session key K at time Tk, and the time Tk has been expired at the current time. The conclusion then gives the expired session key K and its time of issue to the spy.

In the rule *Fake*, the notation $evsf$ is used to denote the trace, and $evs1$ is used in rule *DS1*, and so forth. This is for better clarity in the proving stage. Most theorems are proved by induction, and the goal is split to several subgoals that each corresponding to an inductive rule. In this way, the subgoal with $evs2$ can be easily identified that the last event is introduced by rule *DS2*.

In addition, the lifetime of the session key for this protocol should be assigned a proper value. We have the specification for SesKeyLife in the theory as follows.

```
specification (SesKeyLife)
  SesKeyLife_LB [iff]:  "3 <= SesKeyLife"
    by blast
```

This states that a session key of this protocol should remain valid within at least three events after the issue of the session key, because the protocol has three steps to go from the issue of a session key to the successful establishment of the session.

The full theory DS_Lowe for Lowe's modified Denning-Sacco shared-key protocol can be found in Appendix A. The theory includes the formal specification of the protocol and the proof for security properties.

4.4 Discussions

About the Inductive Model

As mentioned above, each inductive rule extends a given trace with new events. Most rules extend traces with Says events. However, the rule itself does not indicate that the message sent in this step can be received. The message might be received. This reception is identified when the inductive rule of the next protocol step is applied, which contains the reception of the message in the previous step as a premise. As A attempts to send a message M to B, the inductive model allows M to be lost, and allows that M is ignored or rejected by B.

Moreover, the inductive model allows interleaving of protocol sessions. An agent is not force to respond to the newly received message immediately and it is not forced to respond to any message. An agent may ignore messages, or respond to a message for several times. It is also allowed to respond to old messages. In this way, the inductive model is much permissive. Because of the same reason, however, the inductive approach is not applicable to the analysis about *denial-of-service*.

The Implicit Step

Consider our description for the Needham-Schroeder shared-key protocol [23] (see section 2.4.1). The protocol consists of five steps (see Figure 2.5), while our description contains six steps. This is because, by receiving the message sent in step (5), the agent still has to decrypt it and verify the content. This is then described as an additional implicit step. In general, such implicit step also exists for other protocols.

Similarly, for Lowe's modified Denning-Sacco protocol, the message sent in the last step has the same form as the Needham-Schroeder protocol. Thus there is also a similar implicit step for this protocol. At the end of section 4.3.1, we mentioned that the implicit step at the end is not necessary to be formalized for this protocol. This is because the inductive rules never explicitly express agent's behaviours such as decryption and verifying message content. But this is not the whole thing. For timestamp-based protocols, agents need to check the freshness of timestamps. The behaviour of checking timestamps is, however, explicitly expressed in inductive rules by functions such as Expired. In this case, if the implicit step at the end of a protocol includes verifying a timestamp, then

this implicit step needs to be formalized.

Consider the original Denning-Sacco shared-key protocol [11] as an example (see Figure 4.1). This protocol only contains the first three steps of Lowe's modified version. In the third step of this protocol, or say the last explicit step, A sends B the message $\{Kab, A, T\}_{Kb}$. By receiving this message, B decrypts it and check the freshness of the timestamp T. If T is fresh, then he will accept the corresponding session key Kab. This course is considered as the implicit step of this protocol. To model this protocol using inductive approach, this implicit step should be formalized as an additional inductive rule, as follows.

```
D4:  "[| evs4 ∈ ds_lowe; Kab ∈ symKeys;
        Says A' B (Crypt (shrK B) {| Key Kab, Agent A, Number Tk |})
           ∈ set evs4;
        ~ Expired Tk evs4 |]
     ==> Notes B (Crypt (shrK B) (Key Kab)) # evs4 ∈ ds_lowe"
```

In this way, if the timestamp is verified to be fresh, the above rule extends the trace of the model with a Notes event stating that B internally stores the session key.

The *Oops* Rule

As we have mentioned above, the *Oops* rule formalizes the loss of session keys, and it is indispensable for the modelling of a protocol. In general, the Oops rule is intended to model the loss of session keys by any means [28]. Its premise is that the sever has distributed the key for some session, and its conclusion gives this session key to the spy by a Notes event. If we follow this general way, the Oops rule for this protocol model would be simply like this:

```
Oops:  "[| evso ∈ ds_lowe;
           Says Server A (Crypt (shrK A)
              {| Agent B, Key K, Number Tk, X |}) ∈ set evso |]
        ==> Notes Spy {| Number Tk, Key K |} # evso ∈ ds_lowe"
```

On the other hand, although we have not moved to the analysis stage, we can infer that confidentiality and authentication theorems must rely on that no Oop event occurs. Consider the *session key secrecy theorem* [26] which will be mentioned in section 5.6.2. This crucial confidentiality theorem states that if the two participants are uncompromised agents and the session key has not been accidentally leaked by any Oops events, then the session key issued by the server

remains secret to the spy. It is obvious that if the participant is compromised, then the session key is also compromised, because the agent's long-term key has been used to encrypt the session. And if an Oops event leaks the session key, it is not confidential of course. These two assumptions of the theorem are also indispensable assumptions for other confidentiality and authentication theorems, because the session key will be compromised if any of them doesn't hold.

However, these two assumptions are not able to be verified by honest agents. An honest agent is not able to know whether other agents (or even the agent himself) are compromised or not, and he is not able to check whether the accidental loss of session key has occurred. Therefore, these two assumptions of the theorem forms the *minimal trust* [5] named by Bella. Since the agents running the protocols are not able to verify these assumptions, they could only *trust* that they are preserved.

However, for timestamp-based protocols, perhaps allowing the leaking of any session key at any time makes the model too permissive [5]. Bella [6] suggested refining the model with a less permissive Oops rule in his verification of BAN Kerberos. For a similar consideration, we adopt the less permissive Oops rule for our model ds_lowe (see section 4.3.2), by adding another condition Expired $Tk\ evs$. To be specific, the refined Oops rule states that if the server has issued the session key K at time Tk, and Tk has been expired, then K may be leaked to the spy. This is reasonable and realistic, since the risk of losing a session key increases over time.

Moreover, for the refined protocol model, agents are able to check whether the Oops event has occurred. Since the agents can get the timestamp that records the issue time of session key, they can check this timestamp. A verified fresh timestamp can assure the agent that no Oop event occurred. In this way, agents can verify this assumption rather than simply trust it. Therefore, the minimal trust required by confidentiality and authentication theorems becomes lower for the refined protocol model [5].

4.5 Update the Model

As mentioned in section 3.1.3 and 3.2.2, Bella [5] introduced message reception and agent's knowledge to the inductive approach. The definition of agent's knowledge and message reception has been released with the distribution of Isabelle. This extension enhances the inductive approach and prepares it for analyzing new hierarchies of protocols [5], for example, non-repudiation proto-

cols and e-commerce protocols. For basic key-distribution protocols, Paulson's original inductive approach is adequate for the modelling and verification. However, Bella's extension may also improve the readability of the specifications of such protocols and their security properties.

We updated the existing model with the event `Gets` and function knows, and then completed the verification of the new model. The new specification and proof script are included in Appendix B.

In general, to update a model with message reception, a rule named *Reception* should be added. For our model ds_lowe, the *Reception* rule is stated as follows.

```
Reception:  "[| evsr ∈ ds_lowe; Says A B X ∈ set evsr |]
           ==> Gets B X # evsr ∈ ds_lowe"
```

This rule extends a trace of the protocol model with the event Gets $B\,X$, if an event Says $A\,B\,X$ appears on the trace. It allows the reception of a message since it has been sent. Like other inductive rules of the model, *Reception* is not forced to extend a trace, which states that the reception of a message that was sent can not be guaranteed. A Gets event only states the message and the receiver. The sender is not referred at all, since the claim about sender is not reliable on a vulnerable network.

The existing inductive rules are also required for updates. If the inductive rule has an Says event with uncertain sender as a premise, then it could be updated with the corresponding Gets event. In the original inductive approach, Says $A'\,B\,X$ is used to express that the message X reaches B but the sender is uncertain. Here A' which denotes the sender does not appear elsewhere in the rule. In the new model, this case is explicitly expressed with Gets $B\,X$. Moreover, the function spies which states the spy's knowledge should be updated with knows *Spy*.

In our model ds_lowe, the protocol rules *DS2*, *DS3*, *DS4* and *DS5* each contain such Says events in their premises, and thus need to be updated. And the *Fake* rule that contains the function spies is updated with knows *Spy*. The updated model is shown below.

```
consts ds_lowe ::  "event list set"
inductive "ds_lowe"
 intros

  Nil:  "[] ∈ ds_lowe"
```

```
Fake:  "[| evsf ∈ ds_lowe; X ∈ synth (analz (knows Spy evsf)) |]
        ==> Says Spy B X # evsf ∈ ds_lowe"

Reception:  "[| evsr ∈ ds_lowe; Says A B X ∈ set evsr |]
            ==> Gets B X # evsr ∈ ds_lowe"

DS1:  "[| evs1 ∈ ds_lowe |]
        ==> Says A Server {| Agent A, Agent B |} # evs1 ∈ ds_lowe"

DS2:  "[| evs2 ∈ ds_lowe; Key KAB ∉ used evs2; KAB ∈ symKeys;
           Gets Server {| Agent A, Agent B |} ∈ set evs2 |]
        ==> Says Server A
              (Crypt (shrK A)
                {| Agent B, Key KAB, Number (CT evs2),
                  (Crypt (shrK B)
                    {| Key KAB, Agent A, Number (CT evs2) |}) |})
              # evs2 ∈ ds_lowe"

DS3:  "[| evs3 ∈ ds_lowe; A ≠ Server;
           Gets A (Crypt (shrK A)
               {| Agent B, Key K, Number Tk, X |}) ∈ set evs3;
           Says A Server {| Agent A, Agent B |} ∈ set evs3;
            ~ Expired Tk evs3 |]
        ==> Says A B X # evs3 ∈ ds_lowe"

DS4:  "[| evs4 ∈ ds_lowe; Nonce NB ∉ used evs4;
           K ∈ symKeys;
           Gets B (Crypt (shrK B)
               {| Key K, Agent A, Number Tk |}) ∈ set evs4;
           ~ Expired Tk evs4 |]
        ==> Says B A (Crypt K (Nonce NB)) # evs4 ∈ ds_lowe"

DS5:  "[| evs5 ∈ ds_lowe; K ∈ symKeys;
           Gets A (Crypt K (Nonce NB)) ∈ set evs5;
           Gets A (Crypt (shrK A)
                  {| Agent B, Key K, Number Tk, X |}) ∈ set evs5 |]
         ==> Says A B (Crypt K {| Nonce NB, Nonce NB |})
              # evs5 ∈ ds_lowe"

Oops:  "[| evso ∈ ds_lowe;
           Says Server A (Crypt (shrK A)
               {| Agent B, Key K, Number Tk, X |}) ∈ set evso;
           Expired Tk evso |]
        ==> Notes Spy {| Number Tk, Key K |} # evso ∈ ds_lowe"
```

Figure 4.4: Updating the Model with Message Reception

Furthermore, since the new rule *Reception* is introduced, the trace representing an ideal protocol execution requires the inductive rules to be applied in turns as follows.

DS1 - Reception - DS2 - Reception - DS3 - Reception - DS4 - Reception - DS5

Each application of a rule above extends the trace with an new event. In this way, this ideal trace is longer than the one of the old model because of the join of those Gets events which are introduced by the *Reception* rule. Recall that the current time on a trace is formalized as the length of the trace. Therefore, the formalized lifetime of session keys should be doubled. The specification of SesKeyLife should be updated as follows.

```
specification (SesKeyLife)
  SesKeyLife_LB [iff]:  "6 ≤ SesKeyLife"
     by blast
```

It indicates that a session key should be considered fresh within at least six events after the issue of the session key.

At the verification stage, since the model is updated, the theorems and their proofs are also required to be adjusted. This will be discussed in section 5.8. The updated theory DS_Lowe_2 for the protocol can be found in Appendix B.

CHAPTER 5

Verifying the Protocol

This chapter describes how Lowe's modified Denning-Sacco shared-key protocol is verified using the inductive approach. We analyze the expected security goals for this inductive model. These security properties are then specified as several theorems. These theorems are proved using the interactive theorem prover Isabelle [25], and thus the security properties are verified.

The main security properties that we have proved about this protocol are presented, including reliability (§ 5.1), regularity (§ 5.3), unicity (§ 5.4), authenticity (§ 5.5), confidentiality (§ 5.6) and authentication (§ 5.7). In particular, confidentiality and authentication properties are analyzed based on *viewpoints* of individual agents. Some necessary lemmas are also introduced and proved, as they are required for proving the main security goals. At last, we also update the theorems and their proofs with message reception. (§ 5.8)

5.1 Proving Reliability Lemmas

In general, reliability lemmas do not directly indicate any security related properties. However, they may be used to confirm that the formal model we have constructed for a cryptographic protocol is suitable to represent the real protocol. Verification of its reliability is indispensable because further theorem

proving is based on the model.

The *possibility property* [26] is always the first lemma to prove for any protocol. It assures that there exist traces that could reach the last step of the protocol. If the possibility property cannot be proved, it means that the model does not have any trace which allows a complete protocol execution. In this case, the formalization of the protocol must be incorrect, and the resulting formal model is inappropriate for further verification.

The possibility property of our model ds_lowe is specified and proved as follows.

```
lemma "[| A ≠ Server; Key K ∉ used []; K ∈ symKeys |]
   ==> ∃ N. ∃ evs ∈ ds_lowe.
         Says A B (Crypt K {| Nonce N, Nonce N |}) ∈ set evs"

apply (cut_tac SesKeyLife_LB)
apply (intro exI bexI)
apply (rule_tac [2] ds_lowe.Nil [THEN ds_lowe.DS1,
                   THEN ds_lowe.DS2, THEN ds_lowe.DS3,
                   THEN ds_lowe.DS4, THEN ds_lowe.DS5])
apply (possibility, simp add:  used_Cons)
done
```

For a fresh symmetric key K, there exists a nonce N and a trace *evs* on which the event concerning the last step of the protocol occurs. The proof is straightforward, combining all protocol rules. The method `possibility` is provided in theory `Public`, and specialized for proving possibility theorems.

Another reliability lemma states that the server only sends well-formed messages [6]. The lemma is specified and proved as follows.

```
lemma Says_Server_message_form:
  "[| Says Server A (Crypt K' {| Agent B, Key K, Number Tk, X |})
       ∈ set evs; evs ∈ ds_lowe |]
  ==> K ∉ range shrK &
      X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |}) &
      K' = shrK A"

apply (erule rev_mp)
apply (erule ds_lowe.induct)
apply auto
```

```
done
```

The statement of the property is straightforward and the proof is simple. Induction is applied, and it splits the case to several subgoals that each corresponds to an induction rule. Then the method `auto` simplifies all the subgoals.

A lemma named `new_keys_not_used` states that agents can never use non-existent keys. It's specified and proved as follows.

```
lemma new_keys_not_used:
   "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ ds_lowe|]
   ==> K ∉ keysFor (parts (spies evs))"
apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply simp_all
apply (force dest!:  keysFor_parts_insert)
apply blast+
done
```

Recall that used *evs* represents a message set consists of all message components mentioned in trace *evs* and in all agents' initial knowledges. And $K \notin$ keysFor (parts (spies *evs*)) indicates that the key K was not used to encrypt any message components that appear on traffic. In this way, this lemma states that if a symmetric key K does not belong to any agent's initial knowledge and does not appear on traffic, then K cannot ever be used to encrypt any message component that appears on trace *evs*.

The above three theorems cannot guarantee entire reliability of the protocol model, but can assure basic availability that further theorem proving can be proceeded based on the model. Moreover,the lemma `Says_Server_message_form` is also useful in proving some secrecy lemma (see section 5.6), since it states the standard form of the message portion which is not accessible by A. And the lemma `new_keys_not_used` is useful in proving authentication theorem (see section 5.7).

5.2 Proving Forwarding Lemmas

Forwarding lemmas concern the forward of message components. When an inductive rule of the model formalizes that an agent forwards an unreadable portion of the message, a forwarding lemma would be useful for reasoning about this rule. In step (3) of the protocol, the agent A decrypts the outer layer of encryption, and then forwards the unknown message portion to agent B, which is encrypted by B's shared key and thus unreadable to A. Then the forwarding lemma for *DS3* states that the message portion X is in parts of what the spy can see. It is trivial, since the spy can see every message on traffic. The forwarding lemma for *DS3* is specified and proved simply by method `blast` as follows.

```
lemma DS3_msg_in_parts_spies:
   "Says S A (Crypt KA {| B, K, Timestamp, X |}) ∈ set evs
   ==> X ∈ parts (spies evs)"

by blast
```

We also proved a forwarding lemma for the *Oops* rule, as it breaks a layer of encryption of the server's key-distribution message and forward the session key K to the spy. The lemma is specified and proved as follows.

```
lemma Oops_parts_spies:
   "Says Server A (Crypt (shrK A) {| B, K, Timestamp, X |}) ∈ set evs
   ==> K ∈ parts (spies evs)"

by blast
```

5.3 Proving Regularity Lemmas

Regularity lemmas concern occurrences of a particular item X as a possible message component [26]. The statements always have the form of $X \in \mathsf{parts}(\mathsf{spies}\, evs) \rightarrow \ldots$.

A basic regularity lemma states that if an agent is not compromised, then its long-term shared key will not be any component of the messages that can be seen by the spy. If an agent is compromised, the spy holds its long-term key from scratch. Otherwise, the spy does not know a good agent's long-term key at the beginning and cannot learn it by observing message traffics, because the

protocol never allows any agent to include their long-term key in the messages. An agent's long-term shared key may only appear in the message of *Fake* case, and if this happens, it implies the agent is compromised. The basic regularity lemma is specified and proved by induction as follows. The proved forwarding lemma for *DS3* is used in the proving this basic regularity lemma.

```
lemma Spy_see_shrK:
  "evs ∈ ds_lowe ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"

apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply (simp_all)
apply (blast+)
done
```

To explicitly specify that the spy can never get hold of a good agent's shared key, the operator analz is used to express the above lemma. Since we have analz $H \subseteq$ parts H, the proof can be trivial and simply obtained by method `auto` based on the above lemma. This is shown below.

```
lemma Spy_analz_shrK:
  "evs ∈ ds_lowe ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"

by auto
```

5.4 Proving Unicity Theorems

Cryptographic protocols usually require agents to generate fresh session keys or nonces. Fresh means that the same session keys or nonces have not been issued before, that is to say, a certain fresh components can be issued only once. Thus, a fresh session key or nonce can be used to uniquely identify the message that issues it. The *unicity theorem* states that if two events issue the same fresh components, then the values of all other components in the message should be identical as well.

For Lowe's modified Denning-Sacco shared-key protocol, the sever is required to generate a fresh session key and issue it at the second protocol step. The unicity theorem for this protocol states that a certain session key uniquely identifies the the server's message. Given a fresh session key K, its issue time Tk and the

two participants of that protocol execution are determined, as well as all other components of the server's message carrying this session key. This theorem is specified and proved as follows.

```
lemma unique_session_keys:
   "[| Says Server A (Crypt (shrK A)
          {| Agent B, Key K, Number Tk, X |}) ∈ set evs;
       Says Server A' (Crypt (shrK A')
          {| Agent B', Key K, Number Tk', X' |}) ∈ set evs;
       evs ∈ ds_lowe |]
   ==> A = A' & B = B' & Tk = Tk' & X = X'"

apply (erule rev_mp, erule rev_mp, erule ds_lowe.induct)
apply (simp_all)
apply (blast+)
done
```

The proof is proceeded by induction and simplification. The unicity theorem states the unicity and freshness of session keys, and it is also very useful in proving secrecy and authentication lemmas.

5.5 Proving Authenticity Guarantees

As mentioned above, when an agent receives a message, it may not be originated with the sender as claimed, and the agent does not know who is the real sender. Authenticity guarantees can assure the agent that the session key he received is authentic and really originated with the server. In other words, the authenticity guarantees state the security goals of integrity.

For this protocol, A receives the session key by the second message of the protocol. The guarantee for A states that if A is not compromised and a message in the form

$$\mathsf{Crypt}\ (\mathsf{shrK}\ A)\ \{|\ \mathsf{Agent}\ B,\ \mathsf{Key}\ K,\ \mathsf{Number}\ Tk,\ X\ |\}$$

appears on traffic, then the message is originated with the server as an instance of the second step of the protocol. Provided that A is not compromised, the spy does not have A's shared key, and thus not able to fake such message. In this case, by receiving the message, A can confirm that the session key K was issued at time point Tk by the server. If the timestamp Tk is not expired

at current time, A can confirm the freshness of the key and then accept it. The authenticity guarantee for A is specified and proved as follows. It is proved by induction, and the aforementioned forwarding lemma is applied in the proof.

```
lemma A_trusts_K_by_DS2:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          ∈ parts (spies evs);
       A ∉ bad; evs ∈ ds_lowe |]
   ==> Says Server A (Crypt (shrK A)
          {| Agent B, Key K, Number Tk, X |}) ∈ set evs"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply (auto)
done
```

Similarly, B gets hold of the session key by the third message of the protocol. The guarantee for B states that if B is not compromised and a message in the form

Crypt (shrK B) {| Key K, Agent A, Number Tk|}

appears on traffic, then this message is originated with the server. Note that the message portion carrying the session key is originated with the server and then forwarded to B by A. The agent B has the same assurance that the session key he received is issued by the server at time Tk, provided that B is not compromised. The authenticity guarantee for B is specified and proved in a similar way, as follows.

```
lemma B_trusts_K_by_DS3:
   "[| Crypt (shrK B) {| Key K, Agent A, Number Tk |}
         ∈ parts (spies evs);
       B ∉ bad; evs ∈ ds_lowe |]
   ==> Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk,
         Crypt (shrK B) {| Key K, Agent A, Number Tk|}|})∈ set evs"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply (auto)
done
```

5.6 Proving Confidentiality Theorems

For any key-distribution protocol, the confidentiality of session keys is always crucial. To prove the secrecy of session keys, some required lemmas are proved in advance. A typical such lemma is known as the *session key compromise theorem*.

5.6.1 Session Key Compromise Theorem

Paulson's *session key compromise theorem* [26] expresses that the loss of a session key does not compromise the other session keys. To be specific, the theorem states that if a session key K can be extracted from another session key K' and past messages on traffic, then either K and K' are equivalent, or K can be analyzed from past messages without K'. The theorem is denoted in the form

$$K \in \mathsf{analz}\ (\{K'\} \cup (\mathsf{spies}\ evs)) \Longleftrightarrow \\ K = K' \vee \mathsf{Key}\ K \in \mathsf{analz}\ (\mathsf{spies}\ evs).$$

To prove this session key compromise theorem, the lemma in a generalized form has to be proved inductively. The general lemma states that if a session key K can be obtained by analyzing a set KK of session keys and past traffic, then either K is already in the set KK, or K can be analyzed from past traffic alone.

$$K \in \mathsf{analz}\ (KK \cup (\mathsf{spies}\ evs)) \Longleftrightarrow \\ K \in KK \vee \mathsf{Key}\ K \in \mathsf{analz}\ (\mathsf{spies}\ evs)$$

This general lemma is specified and proved by induction as follows.

```
lemma analz_image_freshK:
  "evs ∈ ds_lowe ==>
    ∀ K KK. KK ⊆ - (range shrK) -->
      (Key K ∈ analz (Key‘KK ∪ (spies evs))) =
      (K ∈ KK ∨ Key K ∈ analz (spies evs))"

apply (erule ds_lowe.induct)
apply (drule_tac [8] Says_Server_message_form)
apply (erule_tac [5] Says_S_message_form [THEN disjE])
apply (analz_freshK, spy_analz)
apply (blast+)
done
```

The proof is not such simple because confidentiality lemmas are stated in terms of the operator analz which does not have rich and convenient rewriting rules. However, two specialized methods can be applied in this proof. The method `analz_freshK` is specialized for proving session key compromise theoroms, and the method `spy_analz` is specialized for proving the **Fake** case when analz is involved [1]. The lemma `Says_S_message_form` has to be proved previously. Its proof is omitted here. The full proof script is included in Appendix A.

With the above lemma, the session key compromise theorem can be proved easily. The specification and proof is shown below.

```
lemma analz_insert_freshK:
   "[| evs ∈ ds_lowe; KAB ∉ range shrK |]
   ==> (Key K ∈ analz (insert (Key KAB) (spies evs))) =
        (K = KAB ∨ Key K ∈ analz (spies evs))"

apply (simp only:  analz_image_freshK analz_image_freshK_simps)
done
```

Since Lowe's modified Denning-Sacco shared-key protocol never uses a session key to encrypt other session keys, the loss of a session key will not compromise other session keys. The session key compromise theorem confirms this point, and it helps in proving the *session key secrecy theorem*.

5.6.2 Session Key Secrecy Theorem

Paulson's *session key secrecy theorem* [26] indicates that if the two participants are uncompromised agents, then the session key issued by the server remains secret to the spy, provided that the session key has not been accidentally leaked by any *Oops* events. In other words, the protocol steps never disclose the session key to the spy.

In particular, the session key secrecy theorem of our model for Lowe's modified Denning-Sacco shared-key protocol is subject to the following conditions:

- The trace *evs* belongs too the model ds_lowe.
- A and B are not compromised agents.
- The server has sent an instance of protocol step (2) that issued the session key K, namely the following event has occurred on trace *evs*:

$$\text{Says } Server\ A(\text{ Crypt (shrK } A)\{\!|\text{ Agent } B, \text{ Key } K, \text{ Number } Tk,$$
$$\text{Crypt (shrK } B)\{\!|\text{ Key } K, \text{ Agent } A, \text{ Number } Tk|\!\}\ |\!\}\ ).$$

- The session key is not expired at current time. (Since the *Oops* rule only reveals expired session keys, the freshness of session key K can prevent accidental loss of K.)

If the above assumptions hold, the theorem concludes the secrecy of session key K to the spy:

$$\text{Key } K \notin \text{analz (spies } evs).$$

This session key secrecy theorem is formalized as follows.

```
lemma secrecy_lemma:
   "[| Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk,
                      Crypt (shrK B) {| Key K, Agent A, Number Tk |}|})
          ∈ set evs;
       A ∉ bad; B ∉ bad; evs ∈ ds_lowe |]
   ==> ~ Expired Tk evs -->
              Key K ∉ analz (spies evs)"
```

Although it is not in the standard form of a session key secrecy theorem, this alternative form would be convenient for proving following theorems.

The proof for this theorem is long and thus omitted. It is proved by induction, and several aforementioned lemmas assist this proof, such as the session key compromise theorem, authenticity lemmas and the unicity theorem. The full proof script can be found in Appendix A.

Although the session key secrecy theorem constitutes the main confidentiality result, it is still not directly applicable by the agents [5]. The assumption that the server has sent the key-distribution message can only be checked by the server itself, but cannot be verified by the two participants A and B, since an agent cannot verify the events that occurred on other peers of the network. However, every agent participating the protocol running expects some available guarantees to confirm the confidentiality of session keys. For this consideration, further confidentiality lemmas are introduced for each agent based on the session key secrecy theorem, and then be proved.

Confidentiality to the Server

If A and B are not compromised, and a trace evs of model ds_lowe contains an event in the form

Says $Server$ A (Crypt K' {| Agent B, Key K, Number Tk, X |})

where the timestamp Tk is *not* expired on evs, then the session key K is safe from the spy, namely

Key $K \notin$ analz (spies evs).

This theorem expresses the confidentiality of the session key from the server's point of view. The event that the server sent such a message can be verified by the server, and since the server knows Tk, its freshness can be checked by the server as well. This theorem is specified and proved as follows. The proof is simple with the help of the session key secrecy theorem.

```
lemma Confidentiality_S:
   "[| Says Server A (Crypt K' {| Agent B, Key K, Number Tk, X |})
         ∈ set evs;
      ~ Expired Tk evs;
      A ∉ bad; B ∉ bad; evs ∈ ds_lowe |]
   ==> Key K ∉ analz (spies evs)"

apply (blast dest:  Says_Server_message_form secrecy_lemma)
done
```

Confidentiality to A

If A and B are not compromised, and for a trace evs of model ds_lowe, a message in the form

Crypt (shrK A){| Agent B, Key K, Number Tk, X|}

appears on traffic, and the timestamp Tk is not expired on evs, then the session key K is safe from the spy, namely

Key $K \notin$ analz (spies evs).

This theorem expresses the confidentiality of the session key from A's point

of view. Upon his reception of the second message of the protocol, A is able to confirm the condition that such a message appears on traffic. And since A also obtains Tk, he can verify if it is expired.

This theorem is specified and proved as follows. The proof is helped with the confidentiality theorem for the server and also the authenticity lemma `A_trusts_K_by_DS2`.

```
lemma Confidentiality_A:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          ∈ parts (spies evs);
       ~ Expired Tk evs;
       A ∉ bad; B ∉ bad; evs ∈ ds_lowe|]
   ==> Key K ∉ analz (spies evs)"

apply (blast dest!:  A_trusts_K_by_DS2 Confidentiality_S)
done
```

Confidentiality to B

If A and B are not compromised, and for a trace evs of model ds_lowe, a message component in the form

$$\text{Crypt (shrK B) } \{\!|\text{ Key } K, \text{ Agent } A, \text{ Number } Tk|\!\}$$

appears on traffic, and the timestamp Tk is not expired on evs, then the session key K is safe from the spy, namely

$$\text{Key } K \notin \text{analz (spies } evs).$$

This theorem expresses the confidentiality of the session key from B's point of view. Upon his reception of the third message of the protocol, B is able to confirm the condition that such a message component appears on traffic. And since B also obtains Tk, he can verify its freshness. This theorem is specified and proved as follows. The proof is helped with the confidentiality theorem for the server and also the authenticity lemma `B_trusts_K_by_DS3`.

```
lemma Confidentiality_B:
   "[| Crypt (shrK B) {| Key K, Agent A, Number Tk |}
          ∈ parts (spies evs);
       ~ Expired Tk evs;
```

```
      A ∉ bad; B ∉ bad; evs ∈ ds_lowe|]
  ==> Key K ∉ analz (spies evs)"

apply (blast dest!:  B_trusts_K_by_DS3 Confidentiality_S)
done
```

5.7 Proving Authentication Theorems

Besides the confidentiality of session keys, it is also significant to verify the *mutual authentication* of the protocol. For Lowe's modified Denning-Sacco shared-key protocol, the initiator A should gain the authentication of the responder B upon his reception of the fourth message of the protocol, and similarly the fifth message should authenticate A to B. In order to achieve the authentication, the session key which encrypts these two messages is required to keep its secrecy. Therefore, the conditions of the confidentiality theorems should be covered in the corresponding authentication theorems as assumptions.

Authentication of B to A

If A and B are not compromised, and for a trace evs of model ds_lowe, two messages

> Crypt (shrK A)$\{|$ Agent B, Key K, Number $Tk, X|\}$ and
> Crypt K(Nonce NB)

appear on traffic, and the timestamp Tk is not expired on evs, then the trace evs contains

> Says B A(Crypt K(Nonce NB)),

that is to say, the fourth message is really originated with B.

The above theorem stated the authentication of B to A. According to the aforementioned confidentiality theorem, from A's point of view, if the second message of the protocol appears on traffic, and the timestamp Tk he received is not expired, then A can confirm that the session key K is confidential, provided

that A and B are not compromised. In this case, upon his reception of the fourth message which is encrypted with the safe session key K, A confirms that the message could only be originated with B. In this way, A gets the evidence that it is B who is communicating with him and sharing the session key K with him.

The authentication theorem of B to A is specified as follows. Its proof appeals to the confidentiality theorem for A, and also the unicity theorem, the authenticity lemmas and the forwarding lemmas. The proof is long and thus omitted here.

```
lemma Authentication_B_to_A:
   "[| Crypt K (Nonce NB) ∈ parts (spies evs);
       Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          ∈ parts (spies evs);
       ~ Expired Tk evs;
       A ∉ bad; B ∉ bad; evs ∈ ds_lowe |]
   ==> Says B A (Crypt K (Nonce NB)) ∈ set evs"
```

Authentication of A to B

If A and B are not compromised, and for a trace evs of model ds_lowe, two messages

> Crypt (shrK B)$\{|$ Key K, Agent A, Number $Tk|\}$ and
> Crypt $K\{|$ Nonce NB, Nonce $NB|\}$

appear on traffic, and the timestamp Tk is not expired on evs, then the trace evs contains

> Says A B(Crypt $K\{|$ Nonce NB, Nonce $NB|\}$),

that is to say, the message of the last protocol step is really originated with A.

This theorem stated the authentication of A to B, and it is reasoning from B's point of view. According to the confidentiality theorem for B, if the third message of the protocol appears on traffic, and the timestamp Tk sealed in this message is not expired, then B can confirm that the session key K he received is safe from the spy, provided that A and B are not compromised. In this case, upon his reception of the fourth message sealed under the safe session key K, B

can verify the double nonce NB and thus confirm that the message could only be originated with A. In this way, B can infer that A agrees on the session key K to communicate with him.

The authentication theorem of A to B is specified as follows. Its proof appeals to the confidentiality theorem for A, and as well the unicity theorem, the authenticity lemmas and the forwarding lemmas. The proof is long and thus omitted. The full proof script for this protocol can be found in Appendix A.

```
lemma Authentication_A_to_B:
   "[| Crypt K {| Nonce NB, Nonce NB |} ∈ parts (spies evs);
       Crypt (shrK B) {| Key K, Agent A, Number Tk |}
          ∈ parts (spies evs);
       ~ Expired Tk evs;
       A ∉ bad; B ∉ bad; evs ∈ ds_lowe |]
   ==> Says A B (Crypt K {| Nonce NB, Nonce NB |}) ∈ set evs"
```

5.8 Updating Theorems and Proofs

As we have updated the protocol model with message reception and agent's knowledge (see section 4.5), the theorems and their proofs are also adjusted.

Similarly, each lemma containing Says events with uncertain sender in their premises is updated with the corresponding Gets events. For our existing lemmas, only `Says_S_message_form` and a forwarding lemma `DS3_msg_in_parts_spies` are applicable. We updates all the appearances of function spies in the theory with knows *Spy*. This is applicable to several lemmas. Our updates of the theory also include the adjustment of some theorem names. For example, `Says_S_message_form` is changed to `Gets_A_message_form`, and `DS3_msg_in_parts_spies` is changed to `DS3_msg_in_parts_knows_Spy`. The change of names is not necessary but can improve the readability.

Besides the above updates, because the protocol model is updated with message reception, it is very important to prove some basic lemmas. The following two lemmas are included in our new theory.

```
lemma Gets_imp_Says [dest!]:
   "[| Gets B X ∈ set evs; evs ∈ ds_lowe |]
   ==> ∃ A. Says A B X ∈ set evs"
by (erule rev_mp, erule ds_lowe.induct, auto)
```

```
lemma Gets_imp_knows_Spy:
   "[| Gets B X ∈ set evs; evs ∈ ds_lowe |]
   ==> X ∈ knows Spy evs"
by (blast dest!:  Gets_imp_Says Says_imp_knows_Spy)

declare Gets_imp_knows_Spy [THEN analz.Inj, dest]
```

The lemmas express that if a trace *evs* of our model ds_lowe contains the event Gets $B\, X$, then there exists an agent A that the event Says $A\, B\, X$ also appears on trace *evs*, and the message X is known by the spy. These two lemmas are necessary for protocol models with Gets events and have to be proved for each protocol separately.

Since the model and theorems have been updated, the proof for theorems needs slight modification, especially for the basic session key secrecy theorem and the possibility property. The updated theory DS_Lowe_2 containing modeling and full proof script can be found in Appendix B.

CHAPTER 6

Conclusion

Summary

The thesis concerns formal verification of security protocols and focus on Paulson's inductive approach [paulson1998iav]. We started with the review of cryptographic protocols. Although there exists some other methods for hiding information, such as *steganography* [16], most security protocols employs cryptography to protect data. In general, security protocols are expected to provide confidentiality, integrity and authentication, which can be considered as the most crucial security goals. However, security protocols may contain flaws, and informal reasoning often failed to discover those potential flaws. Several formal methods have been developed and demonstrated their success in protocol analysis. However, they also have limitations. If a protocol can pass the verification by BAN logic [9], it may still contain errors. BAN logic is useful in reasoning about freshness, but it does not attempt to prove secrecy [27]. Model Checking is effective in finding some flaws. But since it only checks limit numbers of states, it also cannot conclude the correctness of a protocol even if no attacks are detected by model checking.

In the inductive approach, security protocols are formalized as inductive models, and their security properties are verified by theorem proving. Several classical protocols have been analyzed and verified using the inductive approach, such

as Yahalom, Otway-Rees, BAN Kerberos and so forth. In this thesis, we verified Lowe's modified Denning-Sacco shared-key protocol using the inductive approach, which has not been shown in other literatures. We investigated the inductive approach including its general principles and basic constituents. Since the protocol we selected relies on timestamps, we also investigated the modelling of timestamps [6]. We formalized the protocol using the inductive approach, and discuss a couple of issues about the inductive model. Since it consists of all possible traces, the inductive model is unbounded. The inductive model is also very permissive since it never forces events to take place. In general, an Oops event is established to allow the leak of session keys by any means [28]. In particular, for this time-based protocol, the loss of a session key is only allowed if it has been expired. This makes the Oops rule less permissive but lower the agent's *minimal trust* [5] required by confidentiality and authentication theorems.

Based on the inductive model, expected security properties are analyzed and then formalized as a number of theorems. Although we concerns confidentiality, integrity and authentication, several other security related theorems are still required to be proved in advance, including reliability lemmas, regularity lemmas, unicity theorem and so on. They can be useful in proving the crucial security theorems. Authenticity guarantees, which assure the agent that the session key is authentic and really originated with the server, are equivalent to the security goals of integrity. And confidentiality and authentication theorems are expressed separately from each agent's point of view. We have completed the proof of these theorems, and thus the protocol is formally verified using the inductive approach.

For the sake of comparison, we also investigate Bella's extension with message reception and agent's knowledge. In the original inductive approach, the message reception is not explicitly expressed, and only the spy's knowledge is stated. The extension with message reception and agent's knowledge has been released with the distribution of *Isabelle*. And it enhances the inductive approach and prepares it for analyzing new hierarchies of protocols [5], such as non-repudiation protocols and e-commerce protocols. We have updated our protocol model with this extension, and subsequently the theorems as well. The proof also needs to be adjusted, and we have proved those updated theorems. Our work demonstrates the proving of confidentiality and authentication for Lowe's modified Denning-Sacco shared-key protocol in both ways. The results seem to be equivalent for this protocol.

We run Isabelle 2005 with proof general 3.7 [4] under Linux Fedora 6 environment, on a PC with 2.6GHz Intel Pentium 4. The full proof script for the original inductive model is executed in about 33 seconds. The runtime of proof script is approximately 37 seconds for the updated model with message reception and agent's knowledge.

Conlusion

The initial goal of this thesis project is to investigate Paulson's inductive approach, and apply this formal approach to a classical security protocols which has not been formally modelled and verified in this way.

We reviewed the theoretical background about cryptographic protocols and their security issues. We have investigated the inductive approach including the original version and its further extensions with modeling of timestamps, message reception and agent's knowledge. We have presented principles and basic constituents of the inductive approach and its extensions.

Lowe's modified Denning-Sacco share-key protocol has been chose, which employs both nonce and timestamps to give evidences of freshness. We have formalized this protocol using the inductive approach, and some issues on the inductive model have been discussed. We have analyzed expected security properties for this protocol model and then verified them with support by the theorem prover Isabelle. We have completed the proof of this theory, and thus this protocol has been formally verified using the inductive approach. Since the crucial security theorems have been proven, we may conclude this protocol preserves the corresponding security properties.

For the sake of comparison, we have updated the inductive model with the extension of message reception and agent's knowledge. Subsequently, the theorems have also been updated in this way. And we have completed the proof of the new theory as well.

As described, we may conclude that this thesis project has achieved its goals.

Appendix A

Verifying Lowe's Denning-Sacco Shared-key Protocol

This is our Isabelle/HOL theory for modelling and verifying Lowe's modified Denning-Sacco shared-key Protocol. This file named `DS_Lowe.thy` should be placed at `$ISABELLE_HOME/src/HOL/Auth` in order to process this theory.

```
theory DS_Lowe imports Public begin

syntax
    CT :: "event list=>nat"
    Expired :: "[nat, event list] => bool"

consts
    SesKeyLife   :: nat

specification (SesKeyLife)
  SesKeyLife_LB [iff]: "3 \<le> SesKeyLife"
    by blast

translations
   "CT" == "length "
   "Expired T evs" == "SesKeyLife + T < CT evs"

consts  ds_lowe   :: "event list set"
inductive "ds_lowe"
 intros

  Nil:  "[] \<in> ds_lowe"

  Fake: "[| evsf \<in> ds_lowe;  X \<in> synth (analz (spies evsf)) |]
        ==> Says Spy B X # evsf \<in> ds_lowe"

  DS1:  "[| evs1 \<in> ds_lowe |]
        ==> Says A Server {| Agent A, Agent B |} # evs1 \<in> ds_lowe"

  DS2:  "[| evs2 \<in> ds_lowe;
            Key KAB \<notin> used evs2;  KAB \<in> symKeys;
            Says A' Server {| Agent A, Agent B |} \<in> set evs2 |]
        ==> Says Server A
                (Crypt (shrK A)
                   {| Agent B, Key KAB, Number (CT evs2),
                      (Crypt (shrK B)
                         {| Key KAB, Agent A, Number (CT evs2) |}) |})
                # evs2 \<in> ds_lowe"

  DS3:  "[| evs3 \<in> ds_lowe;  A \<noteq> Server;
```

```
            Says S A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
               \<in> set evs3;
            Says A Server {| Agent A, Agent B |} \<in> set evs3;
            ~ Expired Tk evs3 |]
         ==> Says A B X # evs3 \<in> ds_lowe"

  DS4:  "[| evs4 \<in> ds_lowe;
            Nonce NB \<notin> used evs4;  K \<in> symKeys;
            Says A' B (Crypt (shrK B) {| Key K, Agent A, Number Tk |})
               \<in> set evs4;
            ~ Expired Tk evs4 |]
         ==> Says B A (Crypt K (Nonce NB)) # evs4 \<in> ds_lowe"

  DS5:  "[| evs5 \<in> ds_lowe;  K \<in> symKeys;
            Says B' A (Crypt K (Nonce NB)) \<in> set evs5;
            Says S  A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
               \<in> set evs5 |]
         ==> Says A B (Crypt K {| Nonce NB, Nonce NB |})
                # evs5 \<in> ds_lowe"

  Oops: "[| evso \<in> ds_lowe;
            Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
               \<in> set evso;
            Expired Tk evso |]
         ==> Notes Spy {| Number Tk, Key K |} # evso \<in> ds_lowe"


declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body  [dest]
declare Fake_parts_insert_in_Un  [dest]
declare analz_into_parts [dest]
declare image_eq_UN [simp]


(*Possibility Property*)
lemma "[| A \<noteq> Server; Key K \<notin> used []; K \<in> symKeys |]
       ==> \<exists>N. \<exists>evs \<in> ds_lowe.
               Says A B (Crypt K {| Nonce N, Nonce N |}) \<in> set evs"

apply (cut_tac SesKeyLife_LB)
apply (intro exI bexI)
apply (rule_tac [2] ds_lowe.Nil [THEN ds_lowe.DS1,
               THEN ds_lowe.DS2, THEN ds_lowe.DS3,
               THEN ds_lowe.DS4, THEN ds_lowe.DS5])
```

```
apply (possibility, simp add: used_Cons)
done



(*Forwarding lemmas*)

lemma DS3_msg_in_parts_spies:
   "Says S A (Crypt KA {| B, K, Timestamp, X |}) \<in> set evs
   ==> X \<in> parts (spies evs)"
by blast


lemma Oops_parts_spies:
   "Says Server A (Crypt (shrK A) {| B, K, Timestamp, X |}) \<in> set evs
   ==> K \<in> parts (spies evs)"
by blast


(*Regularity lemmas*)

lemma Spy_see_shrK [simp]:
   "evs \<in> ds_lowe ==>
       (Key (shrK A) \<in> parts (spies evs)) = (A \<in> bad)"

apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply simp_all
apply blast+
done


lemma Spy_analz_shrK [simp]:
   "evs \<in> ds_lowe ==>
       (Key (shrK A) \<in> analz (spies evs)) = (A \<in> bad)"
by auto


(*Nobody can have used non-existent keys!*)
lemma new_keys_not_used [simp]:
   "[|Key K \<notin> used evs; K \<in> symKeys; evs \<in> ds_lowe|]
   ==> K \<notin> keysFor (parts (spies evs))"

apply (erule rev_mp)
```

```
apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply simp_all
(* Fake *)
apply (force dest!: keysFor_parts_insert)
(* DS2, DS4, DS5 *)
apply blast+
done


(*Server only send well-formed messages*)
lemma Says_Server_message_form:
   "[| Says Server A (Crypt K' {| Agent B, Key K, Number Tk, X |})
          \<in> set evs; evs \<in> ds_lowe |]
   ==> K \<notin> range shrK &
        X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |}) &
        K' = shrK A"
by (erule rev_mp, erule ds_lowe.induct, auto)


(*Authenticity Lemmas*)

lemma A_trusts_K_by_DS2:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (spies evs); A \<notin> bad;  evs \<in> ds_lowe |]
   ==> Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
          \<in> set evs"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies, auto)
done


lemma B_trusts_K_by_DS3:
   "[| Crypt (shrK B) {| Key K, Agent A, Number Tk |}
          \<in> parts (spies evs); B \<notin> bad; evs \<in> ds_lowe |]
   ==> Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk,
          Crypt (shrK B) {| Key K, Agent A, Number Tk|}|}) \<in> set evs"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies, auto)
done
```

```
lemma cert_A_form:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (spies evs); A \<notin> bad;  evs \<in> ds_lowe |]
   ==> K \<notin> range shrK  &
       X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |})"
by (blast dest!: A_trusts_K_by_DS2 Says_Server_message_form)


lemma Says_S_message_form:
   "[| Says S A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
          \<in> set evs; evs \<in> ds_lowe |]
   ==> (K \<notin> range shrK
           & X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |}))
        | X \<in> analz (spies evs)"
by (blast dest: Says_imp_knows_Spy analz_shrK_Decrypt cert_A_form analz.Inj)


(*Unicity Theorem*)
lemma unique_session_keys:
   "[| Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
          \<in> set evs;
       Says Server A' (Crypt (shrK A') {| Agent B', Key K, Number Tk', X' |})
          \<in> set evs;
       evs \<in> ds_lowe |]
   ==> A=A' & B=B' & Tk=Tk' & X = X'"

apply (erule rev_mp, erule rev_mp, erule ds_lowe.induct)
apply simp_all
(* DS2, DS3 *)
apply blast+
done



(**************Confidentiality*****************)

lemma analz_image_freshK [rule_format (no_asm)]:
   "evs \<in> ds_lowe ==>
       \<forall>K KK. KK \<subseteq> - (range shrK) -->
           (Key K \<in> analz (Key`KK Un (spies evs))) =
           (K \<in> KK | Key K \<in> analz (spies evs))"
```

```
apply (erule ds_lowe.induct)
apply (drule_tac [8] Says_Server_message_form)
apply (erule_tac [5] Says_S_message_form [THEN disjE])
apply (analz_freshK)
apply (spy_analz)
(* DS2, DS3 *)
apply blast+;
done


(*Session Key Compromise Theorem*)
lemma analz_insert_freshK:
   "[| evs \<in> ds_lowe;  KAB \<notin> range shrK |]
   ==> (Key K \<in> analz (insert (Key KAB) (spies evs))) =
           (K = KAB | Key K \<in> analz (spies evs))"
apply (simp only: analz_image_freshK analz_image_freshK_simps)
done


(* Session Key Secrecy Theorem *)
lemma secrecy_lemma:
   "[| Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk,
          Crypt (shrK B) {| Key K, Agent A, Number Tk |} |}) \<in> set evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> ~ Expired Tk evs -->
          Key K \<notin> analz (spies evs)"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (frule_tac [7] Says_Server_message_form)
apply (frule_tac [4] Says_S_message_form)
apply (erule_tac [5] disjE)
apply (simp_all add: analz_insert_eq analz_insert_freshK
                     less_SucI pushes split_ifs)
apply (spy_analz)
(* DS2 *)
apply blast
prefer 3
(* Oops *)
apply (blast dest: unique_session_keys intro: less_SucI)
prefer 2
(* DS3 spy-subcase *)
apply (blast dest: unique_session_keys intro: less_SucI)
(* DS3 server-subcase *)
```

```
apply auto
apply (blast dest: A_trusts_K_by_DS2 Crypt_Spy_analz_bad analz.Inj
                   Says_imp_spies unique_session_keys
            intro: less_SucI)
done



(* Confidentiality of K from server's view *)
lemma Confidentiality_S:
   "[| Says Server A (Crypt K' {| Agent B, Key K, Number Tk, X |})
          \<in> set evs;
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> Key K \<notin> analz (spies evs)"
apply (blast dest: Says_Server_message_form secrecy_lemma)
done


(* Confidentiality of K from A's view *)
lemma Confidentiality_A:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (spies evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe|]
   ==> Key K \<notin> analz (spies evs)"
apply (blast dest!: A_trusts_K_by_DS2 Confidentiality_S)
done


(* Confidentiality of K from B's view *)
lemma Confidentiality_B:
   "[| Crypt (shrK B) {| Key K, Agent A, Number Tk|}
          \<in> parts (spies evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe|]
   ==> Key K \<notin> analz (spies evs)"
apply (blast dest!: B_trusts_K_by_DS3 Confidentiality_S)
done



(*************Authentication**************)
```

```
lemma lemma_B_to_A [rule_format]:
   "evs \<in> ds_lowe ==>
      Key K \<notin> analz (spies evs) -->
      Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
         \<in> set evs -->
      Crypt K (Nonce NB) \<in> parts (spies evs) -->
      Says B A (Crypt K (Nonce NB)) \<in> set evs"

apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply (analz_mono_contra)
apply simp_all
(* Fake *)
apply blast
(* DS2 *)
apply (force dest!: Crypt_imp_keysFor)
(* DS3 *)
apply blast
(* DS4 *)
apply (blast dest: B_trusts_K_by_DS3
                   Says_imp_knows_Spy [THEN analz.Inj]
                   Crypt_Spy_analz_bad unique_session_keys)
done


(* Authentication for B, from A's view *)
lemma Authentication_B_to_A:
   "[| Crypt K (Nonce NB) \<in> parts (spies evs);
       Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (spies evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> Says B A (Crypt K (Nonce NB)) \<in> set evs"
apply (blast intro: lemma_B_to_A
             dest: A_trusts_K_by_DS2 Confidentiality_S)
done


lemma lemma_A_to_B [rule_format]:
   "[| B \<notin> bad;  evs \<in> ds_lowe |] ==>
       Key K \<notin> analz (spies evs) -->
       Says Server A
       (Crypt (shrK A) {| Agent B, Key K, Number Tk,
                          Crypt (shrK B) {| Key K, Agent A, Number Tk |} |})
```

```
          \<in> set evs -->
       Crypt K {| Nonce NB, Nonce NB |} \<in> parts (spies evs) -->
       Says A B (Crypt K {| Nonce NB, Nonce NB |}) \<in> set evs"

apply (erule ds_lowe.induct, force)
apply (drule_tac [4] DS3_msg_in_parts_spies)
apply (analz_mono_contra)
apply simp_all
(* Fake *)
apply blast
(* DS2 *)
apply (blast dest!: new_keys_not_used Crypt_imp_keysFor)
(* DS3 *)
apply (blast dest!: cert_A_form)
(* DS5 *)
apply (blast dest!: A_trusts_K_by_DS2
              dest: Says_imp_knows_Spy [THEN analz.Inj]
                    unique_session_keys Crypt_Spy_analz_bad)
done


(* Authentication of A, from B's view *)
lemma Authentication_A_to_B:
   "[| Crypt K {| Nonce NB, Nonce NB |} \<in> parts (spies evs);
       Crypt (shrK B) {| Key K, Agent A, Number Tk |}
          \<in> parts (spies evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> Says A B (Crypt K {| Nonce NB, Nonce NB |}) \<in> set evs"
apply (blast intro: lemma_A_to_B
              dest: B_trusts_K_by_DS3 Confidentiality_S)
done

end
```

Appendix B

Updated Model and Theorems with Message Reception

This is our Isabelle/HOL theory for modelling and verifying Lowe's modified Denning-Sacco shared-key Protocol, which has been updated with Bella's extension of message reception and agent's knowledge [5]. This file named `DS_Lowe_2.thy` should be placed at `$ISABELLE_HOME/src/HOL/Auth` in order to process this theory.

```
theory DS_Lowe_2 imports Public begin

syntax
    CT :: "event list=>nat"
    Expired :: "[nat, event list] => bool"

consts
    SesKeyLife   :: nat

specification (SesKeyLife)
  SesKeyLife_LB [iff]: "6 \<le> SesKeyLife"
    by blast

translations
   "CT" == "length "
   "Expired T evs" == "SesKeyLife + T < CT evs"

consts  ds_lowe   :: "event list set"
inductive "ds_lowe"
 intros

  Nil:  "[] \<in> ds_lowe"

  Fake: "[| evsf \<in> ds_lowe;
            X \<in> synth (analz (knows Spy evsf)) |]
         ==> Says Spy B X # evsf \<in> ds_lowe"

  Reception: "[| evsr \<in> ds_lowe;  Says A B X \<in> set evsr |]
              ==> Gets B X # evsr \<in> ds_lowe"

  DS1:  "[| evs1 \<in> ds_lowe |]
         ==> Says A Server {| Agent A, Agent B |} # evs1 \<in> ds_lowe"

  DS2:  "[| evs2 \<in> ds_lowe;
            Key KAB \<notin> used evs2;  KAB \<in> symKeys;
            Gets Server {| Agent A, Agent B |} \<in> set evs2 |]
         ==> Says Server A
                (Crypt (shrK A)
                  {| Agent B, Key KAB, Number (CT evs2),
                     (Crypt (shrK B) {| Key KAB, Agent A, Number (CT evs2)|})|})
```

```
                # evs2 \<in> ds_lowe"

  DS3:  "[| evs3 \<in> ds_lowe;  A \<noteq> Server;
            Gets A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
               \<in> set evs3;
            Says A Server {| Agent A, Agent B |} \<in> set evs3;
            ~ Expired Tk evs3 |]
         ==> Says A B X # evs3 \<in> ds_lowe"

  DS4:  "[| evs4 \<in> ds_lowe;
            Nonce NB \<notin> used evs4;  K \<in> symKeys;
            Gets B (Crypt (shrK B) {| Key K, Agent A, Number Tk |})
               \<in> set evs4;
            ~ Expired Tk evs4 |]
         ==> Says B A (Crypt K (Nonce NB)) # evs4 \<in> ds_lowe"

  DS5:  "[| evs5 \<in> ds_lowe;  K \<in> symKeys;
            Gets A (Crypt K (Nonce NB)) \<in> set evs5;
            Gets A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
               \<in> set evs5 |]
         ==> Says A B (Crypt K {| Nonce NB, Nonce NB |})
               # evs5 \<in> ds_lowe"

  Oops: "[| evso \<in> ds_lowe;
            Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
               \<in> set evso;
            Expired Tk evso |]
         ==> Notes Spy {| Number Tk, Key K |} # evso \<in> ds_lowe"


declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body  [dest]
declare Fake_parts_insert_in_Un  [dest]
declare analz_into_parts [dest]
declare image_eq_UN [simp]


(*Possibility Property*)
lemma "[| A \<noteq> Server; Key K \<notin> used []; K \<in> symKeys |]
      ==> \<exists>N. \<exists>evs \<in> ds_lowe.
              Says A B (Crypt K {| Nonce N, Nonce N |}) \<in> set evs"

apply (cut_tac SesKeyLife_LB)
apply (intro exI bexI)
```

```
apply (rule_tac [2] ds_lowe.Nil
              [THEN ds_lowe.DS1, THEN ds_lowe.Reception,
               THEN ds_lowe.DS2, THEN ds_lowe.Reception,
               THEN ds_lowe.DS3, THEN ds_lowe.Reception,
               THEN ds_lowe.DS4, THEN ds_lowe.Reception,
               THEN ds_lowe.DS5])
apply (possibility, simp add: used_Cons)
done


(*Necessary lemmas for Gets*)

lemma Gets_imp_Says [dest!]:
   "[| Gets B X \<in> set evs; evs \<in> ds_lowe |]
    ==> \<exists>A. Says A B X \<in> set evs"
by (erule rev_mp, erule ds_lowe.induct, auto)

lemma Gets_imp_knows_Spy:
   "[| Gets B X \<in> set evs; evs \<in> ds_lowe |]
    ==> X \<in> knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

declare Gets_imp_knows_Spy [THEN analz.Inj, dest]


(*Forwarding lemmas*)

lemma DS3_msg_in_parts_knows_Spy:
   "[| Gets A (Crypt KA {| B, K, Timestamp, X |}) \<in> set evs;
       evs \<in> ds_lowe |]
    ==> X \<in> parts (knows Spy evs)"
by blast

lemma Oops_parts_knows_Spy:
   "Says Server A (Crypt (shrK A) {| B, K, Timestamp, X |}) \<in> set evs
   ==> K \<in> parts (knows Spy evs)"
by blast


(*Regularity lemmas*)

lemma Spy_see_shrK [simp]:
   "evs \<in> ds_lowe ==>
       (Key (shrK A) \<in> parts (knows Spy evs)) = (A \<in> bad)"
```

```
apply (erule ds_lowe.induct, force)
apply (drule_tac [5] DS3_msg_in_parts_knows_Spy)
apply simp_all
apply blast+
done


lemma Spy_analz_shrK [simp]:
   "evs \<in> ds_lowe ==>
       (Key (shrK A) \<in> analz (knows Spy evs)) = (A \<in> bad)"
by auto



(*Nobody can have used non-existent keys!*)
lemma new_keys_not_used [simp]:
   "[|Key K \<notin> used evs; K \<in> symKeys; evs \<in> ds_lowe|]
   ==> K \<notin> keysFor (parts (knows Spy evs))"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [5] DS3_msg_in_parts_knows_Spy)
apply (simp_all)
(* Fake *)
apply (force dest!: keysFor_parts_insert)
(* DS2, DS4, DS5 *)
apply (blast+)
done



(*Server only send well-formed messages*)
lemma Says_Server_message_form:
   "[| Says Server A (Crypt K' {| Agent B, Key K, Number Tk, X |})
          \<in> set evs; evs \<in> ds_lowe |]
   ==> K \<notin> range shrK &
       X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |}) &
       K' = shrK A"
by (erule rev_mp, erule ds_lowe.induct, auto)


(*Authenticity Lemmas*)
```

```
lemma A_trusts_K_by_DS2:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (knows Spy evs);
       A \<notin> bad;  evs \<in> ds_lowe |]
   ==> Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
          \<in> set evs"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [5] DS3_msg_in_parts_knows_Spy)
apply (auto)
done


lemma B_trusts_K_by_DS3:
   "[| Crypt (shrK B) {| Key K, Agent A, Number Tk |}
          \<in> parts (knows Spy evs);
       B \<notin> bad; evs \<in> ds_lowe |]
   ==> Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk,
          Crypt (shrK B) {| Key K, Agent A, Number Tk|}|}) \<in> set evs"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (drule_tac [5] DS3_msg_in_parts_knows_Spy, auto)
done


lemma cert_A_form:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (knows Spy evs);
       A \<notin> bad;  evs \<in> ds_lowe |]
   ==> K \<notin> range shrK  &
       X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |})"
by (blast dest!: A_trusts_K_by_DS2 Says_Server_message_form)


lemma Gets_A_message_form:
   "[| Gets A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
          \<in> set evs; evs \<in> ds_lowe |]
   ==> (K \<notin> range shrK
          & X = (Crypt (shrK B) {| Key K, Agent A, Number Tk |}))
        | X \<in> analz (knows Spy evs)"
by (blast dest: Says_imp_knows_Spy analz_shrK_Decrypt cert_A_form analz.Inj)
```

```
(*Unicity Lemma*)
lemma unique_session_keys:
   "[| Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
          \<in> set evs;
       Says Server A' (Crypt (shrK A') {| Agent B', Key K, Number Tk', X' |})
          \<in> set evs;
       evs \<in> ds_lowe |]
   ==> A=A' & B=B' & Tk=Tk' & X = X'"

apply (erule rev_mp, erule rev_mp, erule ds_lowe.induct)
apply simp_all
(* DS2, DS3 *)
apply blast+
done



(*********************Confidentiality****************)

lemma analz_image_freshK [rule_format (no_asm)]:
   "evs \<in> ds_lowe ==>
       \<forall>K KK. KK \<subseteq> - (range shrK) -->
           (Key K \<in> analz (Key`KK Un (knows Spy evs))) =
           (K \<in> KK | Key K \<in> analz (knows Spy evs))"

apply (erule ds_lowe.induct)
apply (drule_tac [9] Says_Server_message_form)
apply (erule_tac [6] Gets_A_message_form [THEN disjE])
apply (analz_freshK)
apply (spy_analz)
(* DS2, DS3 *)
apply blast+;
done


(*Session Key Compromise Theorem*)
lemma analz_insert_freshK:
   "[| evs \<in> ds_lowe;  KAB \<notin> range shrK |]
   ==> (Key K \<in> analz (insert (Key KAB) (knows Spy evs))) =
          (K = KAB | Key K \<in> analz (knows Spy evs))"
apply (simp only: analz_image_freshK analz_image_freshK_simps)
done
```

```
(* Session Key Secrecy Theorem *)
lemma secrecy_lemma:
   "[| Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk,
          Crypt (shrK B) {| Key K, Agent A, Number Tk |} |}) \<in> set evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> ~ Expired Tk evs -->
       Key K \<notin> analz (knows Spy evs)"

apply (erule rev_mp)
apply (erule ds_lowe.induct, force)
apply (frule_tac [8] Says_Server_message_form)
apply (frule_tac [5] Gets_A_message_form)
apply (erule_tac [6] disjE)
apply (simp_all add: analz_insert_eq analz_insert_freshK
                     less_SucI pushes split_ifs)
apply (spy_analz)
(* DS2 *)
apply blast
prefer 3
(* Oops *)
apply (blast dest: unique_session_keys intro: less_SucI)
prefer 2
(* DS3 spy-subcase *)
apply (blast dest: unique_session_keys intro: less_SucI)
(* DS3 server-subcase *)
apply (drule_tac Gets_imp_Says)
apply auto
apply (blast dest: A_trusts_K_by_DS2 Crypt_Spy_analz_bad analz.Inj
                   Says_imp_knows_Spy unique_session_keys
            intro: less_SucI)
done


(* Confidentiality of K from server's view *)
lemma Confidentiality_S:
   "[| Says Server A (Crypt K' {| Agent B, Key K, Number Tk, X |})
          \<in> set evs;
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> Key K \<notin> analz (knows Spy evs)"
apply (blast dest: Says_Server_message_form secrecy_lemma)
done
```

```
(* Confidentiality of K from A's view *)
lemma Confidentiality_A:
   "[| Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (knows Spy evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe|]
   ==> Key K \<notin> analz (knows Spy evs)"
apply (blast dest!: A_trusts_K_by_DS2 Confidentiality_S)
done


(* Confidentiality of K from B's view *)
lemma Confidentiality_B:
   "[| Crypt (shrK B) {| Key K, Agent A, Number Tk|}
          \<in> parts (knows Spy evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe|]
   ==> Key K \<notin> analz (knows Spy evs)"
apply (blast dest!: B_trusts_K_by_DS3 Confidentiality_S)
done


(**************Authentication*****************)

lemma lemma_B_to_A [rule_format]:
   "evs \<in> ds_lowe ==>
      Key K \<notin> analz (knows Spy evs) -->
      Says Server A (Crypt (shrK A) {| Agent B, Key K, Number Tk, X |})
         \<in> set evs -->
      Crypt K (Nonce NB) \<in> parts (knows Spy evs) -->
      Says B A (Crypt K (Nonce NB)) \<in> set evs"

apply (erule ds_lowe.induct, force)
apply (drule_tac [5] DS3_msg_in_parts_knows_Spy)
apply (analz_mono_contra)
apply simp_all
(* Fake *)
apply blast
(* DS2 *)
apply (force dest!: Crypt_imp_keysFor)
(* DS3 *)
apply blast
```

```
(* DS4 *)
apply (blast dest: B_trusts_K_by_DS3
                   Says_imp_knows_Spy [THEN analz.Inj]
                   Crypt_Spy_analz_bad unique_session_keys)
done


(* Authentication for B, from A's view *)
lemma Authentication_B_to_A:
   "[| Crypt K (Nonce NB) \<in> parts (knows Spy evs);
       Crypt (shrK A) {| Agent B, Key K, Number Tk, X |}
          \<in> parts (knows Spy evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
   ==> Says B A (Crypt K (Nonce NB)) \<in> set evs"
apply (blast intro: lemma_B_to_A
             dest: A_trusts_K_by_DS2 Confidentiality_S)
done



lemma lemma_A_to_B [rule_format]:
   "[| B \<notin> bad;  evs \<in> ds_lowe |] ==>
          Key K \<notin> analz (knows Spy evs) -->
          Says Server A
       (Crypt (shrK A) {| Agent B, Key K, Number Tk,
     Crypt (shrK B) {| Key K, Agent A, Number Tk |} |})
             \<in> set evs -->
          Crypt K {| Nonce NB, Nonce NB |} \<in> parts (knows Spy evs) -->
          Says A B (Crypt K {| Nonce NB, Nonce NB |}) \<in> set evs"

apply (erule ds_lowe.induct, force)
apply (drule_tac [5] DS3_msg_in_parts_knows_Spy)
apply (analz_mono_contra)
apply simp_all
(* Fake *)
apply blast
(* DS2 *)
apply (blast dest!: new_keys_not_used Crypt_imp_keysFor)
(* DS3 *)
apply (blast dest!: cert_A_form)
(* DS5 *)
apply (blast dest!: A_trusts_K_by_DS2
             dest: Says_imp_knows_Spy [THEN analz.Inj]
```

```
                    unique_session_keys Crypt_Spy_analz_bad)
done


(* Authentication of A, from B's view *)
lemma Authentication_A_to_B:
   "[| Crypt K {| Nonce NB, Nonce NB |} \<in> parts (knows Spy evs);
       Crypt (shrK B) {| Key K, Agent A, Number Tk |}
          \<in> parts (knows Spy evs);
       ~ Expired Tk evs;
       A \<notin> bad;  B \<notin> bad;  evs \<in> ds_lowe |]
    ==> Says A B (Crypt K {| Nonce NB, Nonce NB |}) \<in> set evs"
apply (blast intro: lemma_A_to_B
              dest: B_trusts_K_by_DS3 Confidentiality_S)
done

end
```

Bibliography

[1] Theories for Verifying Cryptographic Protocols. Also available as http://isabelle.in.tum.de/dist/library/HOL/Auth/index.html.

[2] M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[3] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. Research Report 125, Digital Equipment Corp. *System Research Center*, 1994.

[4] D. Aspinall and T. Kleymann. User Manual for Proof General 3.5. *University of Edinburgh*, 2004.

[5] G. Bella. *Inductive Verification of Cryptographic Protocols.* PhD thesis, University of Cambridge, Computer Laboratory, 2000.

[6] G. Bella and L.C. Paulson. Mechanising BAN Kerberos by the Inductive Method. *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, 1427:416–427.

[7] G. Bella and L.C. Paulson. Using Isabelle to Prove Properties of the Kerberos Authentication System. *Proc. of DIMACS*, 97, 1997.

[8] C. Bodei. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[9] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems (TOCS)*, 8(1):18–36, 1990.

[10] R.X. CCITT. 509: The Directory Authentication Framework, 1988.

[11] D.E. Denning and G.M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[12] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[13] Laboratoire Spécification et Vérification (LSV). SPORE: Security Protocols Open Repository. http://www.lsv.ens-cachan.fr/spore/.

[14] D. Gollmann. What do we mean by entity authentication. *Symposium on Security and Privacy*, pages 46–54, 1996.

[15] C.A.R. Hoare. Communicating Sequential Processes. Series in Computer Science, 1985.

[16] S. Katzenbeisser and F. Petitolas. Information Hiding Techniques for Steganography and Digital Watermaking. *EDPACS*, 28(6):1–2, 2000.

[17] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[18] G. Lowe. A family of attacks upon authentication protocols. *Proceedings of the 10th Computer Seurity Foundation Workshop (CSFW'97), Rockport, Massachusetts, USA*, 1997.

[19] L. Ma and J.J.P. Tsai. Formal verification Techniques for Computer Communication Security Protocols. *Handbook of Software Engineering and Knowledge Engineering*, 15, 2000.

[20] C. Meadows. Formal verification of cryptographic protocols: A survey. *Advances in Cryptology-Asiacrypt*, 94:133–150.

[21] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[22] R.R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[23] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[24] H.R. Nielson. Validation of Cryptographic Protocols using Static Analysis, 2007. Slides for Course 02244, Informatics and Mathematical Modelling, Technical University of Denmark.

[25] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/Hol: A Proof Assistant for Higher-order Logic*. Springer, 2005.

[26] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[27] LC Paulson. Proving security protocols correct. *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, pages 370–381, 1999.

[28] L.C. Paulson. Relations between secrets: two formal analyses of the Yahalom protocol. *Journal of Computer Security*, 9(3):197–216, 2001.

[29] S.L. Pfleeger and C.P. Pfleeger. *Security in Computing*. Prentice Hall PTR, 2003.

[30] A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. *8th IEEE Computer Security Foundations Workshop*, pages 98–107, 1995.

[31] B. Schneier. *Applied cryptography*. Wiley New York, 1996.

[32] P. Syverson. The use of logic in the analysis of cryptographic protocols. *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 156–170, 1991.

[33] T. Yasuda and K. Takahashi. Verification of Wide Mouth Frog Protocol Using Theorem Prover. 2006.