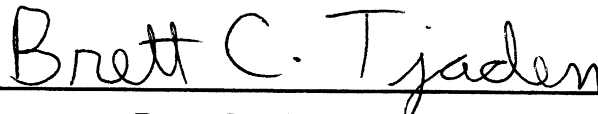
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

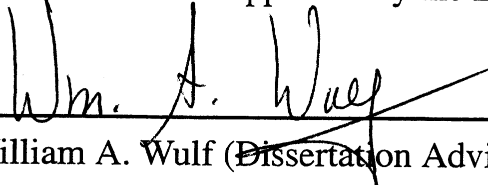
requirements for the degree of

Doctor of Philosophy (Computer Science)

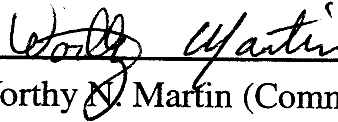


Brett C. Tjaden (Author)

This dissertation has been read and approved by the Examining Committee:

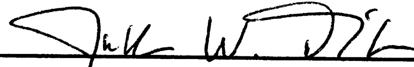


William A. Wulf (Dissertation Advisor)

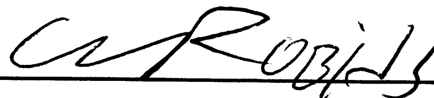


Worthy N. Martin (Committee Chairman)

Donald A. Brown (Committee Member)

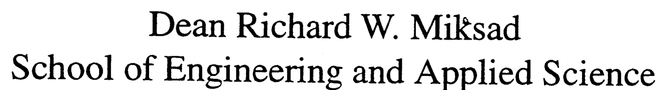


Jack W. Davidson (Committee Member)



Gabriel Robins (Committee Member)

Accepted for the School of Engineering and Applied Science:



Dean Richard W. Miksad
School of Engineering and Applied Science

May, 1997

A Method For Examining Cryptographic Protocols

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Brett C. Tjaden

May, 1997

© Copyright by

Brett C. Tjaden

All Rights Reserved

May, 1997

To my family:

Thank you Mom, Dad, and Brian, for a lifetime of love. I will never be able to repay you for everything you have done for me.

Acknowledgments

I want to thank Bill Wulf, my advisor, for the encouragement, guidance, and insight he has given me during my time here. Working with you, Bill, has been both enjoyable and highly beneficial, and I will always feel lucky to have had you as my advisor.

I would also like to thank all of my colleagues, past and present, who have made my time at the University both enjoyable and rewarding. In particular, I would like to acknowledge David Cooper, Russell Haddleton, Darrell Kienzle, Sally McKee, Michael Nahas, Chris Oliver, Ramesh Peri, Anne Stinehart, Chenxi Wang, Dee Weikle, Alec Yasinsac, and Tongtong Zhang.

Abstract

In this dissertation we present a powerful, general, extensible, and formal methodology that automatically examines cryptographic protocols. Our approach is to specify a protocol, its assumptions, and a statement of failure in a notation that allows us to give a formal semantic definition of the protocol and its failure conditions. We then use deductive program synthesis to try to automatically modify the protocol so that the failure condition is achieved. If this last step succeeds, we have found a weakness in the protocol and we can then give a step-by-step description of an attack to the user. Given an adequate set of axioms and enough time, our method will find any attack that exists for a given protocol and failure condition. Even if our methodology does not discover a flaw in the amount of time it is given to run, we can make a concrete statement about the minimum length of a constructive proof for any attack that might exist on the protocol (for the given failure condition and axiom set) as a result of its analysis. A preliminary implementation of our methodology has had great success in finding both known and previously unknown flaws in a significant number of published protocols. This system also helped us to discover and demonstrate an important new class of attacks based on the interaction of two or more cryptographic protocols.

Chapter 1

Introduction

1.1 Computer Security

In the days of stand-alone computers the field of computer security was concerned with protecting the system's resources from misuse and controlling the information that was stored on the computer. System resources were protected by ensuring that only specific users were permitted the use of certain resources and that no user could abuse any resource. For example, many operating systems used disk quotas and processor scheduling algorithms to apportion storage space and CPU cycles fairly among the system's users. Information was controlled by managing the rights that each user had to it and the manner in which users could communicate information to one another. Many methods have been proposed to address these two problems, but most of them are based on the access-control matrix model proposed by Butler Lampson in [LAM74].

Lampson's *access-control matrix* model consists of a set of *objects* that are to be protected and a set of *subjects* which perform actions and are themselves objects in the system. Objects can also be resources such as CPUs, memory segments, disk drives, terminals,

or printers, or they can be information such as processes, files, databases, or semaphores. Each operation that can be performed on an object corresponds to a *right*. The objects are then protected in the following straight-forward manner. If a subject has a right to an object, he is permitted to perform that operation on the object; otherwise he is not. An access-control matrix has a row for each user and a column for each object. Each entry in the matrix specifies what subset of the rights for the object a particular user has. An example of an access-control matrix is given below.

We define S, the set of *subjects*, as follows:

S = {Subject A, Subject B, Subject C}

We define O, the set of *objects*, as follows:

O = {Object 1, Object 2, Subject A, Subject B, Subject C}

The access-control matrix, M, that defines what *rights* each subject has to each object is given in Figure 1.1 below.

	Object 1	**Object 2**	**Subject A**	**Subject B**	**Subject C**
Subject A	Read		Read Write	Write	
Subject B		Execute		Read Write Execute	
Subject C	Read Write				Read Write

Figure 1.1 : An Access-Control Matrix

When subject *X* requests to perform operation *Y* on object *Z*, we simply need to refer to row *X*, column *Z* of the matrix to determine whether or not *Y* is one of the rights *X* has to *Z*:

Allow subject *X* to perform operation *Y* on object *Z* if $Y \in M\left[X\ Z\right]$

Disallow subject *X* to perform operation *Y* on object *Z* if $Y \notin M\left[X\ Z\right]$

In practice, an access-control matrix is often large and sparse, so it makes sense to store only the non-empty elements in it. There are two ways to accomplish this. One is to make a list, indexed by the objects, and record which subjects have which rights to that object. This is commonly referred to as an *access control list*. The access control list that corresponds to the matrix above is given in Figure 1.2.

Object 1: (A, Read), (C, Read, Write)

Object 2: (B, Execute)

Subject A: (A, Read, Write)

Subject B: (A, Write), (B, Read, Write, Execute)

Subject C: (C, Read, Write)

Figure 1.2 : The Matrix Expressed as an Access Control List

The other way of partitioning the matrix of Figure 1.1 is by rows. Using this strategy we obtain a list for each subject of the objects that may be accessed and the rights to that object. This is called a *capability list* and is illustrated in Figure 1.3.

Subject A: (Object 1, Read), (Subject A, Read, Write), (Subject B, Write)

Subject B: (Object 2, Execute), (Subject B, Read, Write, Execute)

Subject C: (Object 1, Read, Write), (Subject C, Read, Write)

Figure 1.3 : The Matrix Expressed as a Capability List

Access control mechanisms play an important role in securing information stored on computer systems, but they are not a complete solution. That is because there are other ways for an unauthorized user to gain access to information. For example, if Alice does not have permission to read a file but Bob does, Alice might be able to convince Bob to read that file and tell her its contents. In this scenario, none of the access control policies have been violated, but this still represents a breach of security, since Alice was not authorized to have access to the contents of the file. Issues like these are addressed by what are called *information flow* security policies, defined by Denning [DEN76] as follows: "*information is said to flow from object a to object b if the value of object b some how depends on the value of object a.*"

The most widely-used information flow policy is Multi-Level Security(MLS). In this policy, every entity in the system is assigned a level from a partially ordered set, *L*, of security levels:

L = {Unclassified, Classified, Secret, Top Secret}

The most common partial ordering is that:

$$Unclassified \leq Classified \leq Secret \leq Top\ Secret$$

Each object in the system is assigned a security level from L:

Subject A: Secret

Subject B: Unclassified

Subject C: Classified

Object 1: Classified

Object 2: Top Secret

A typical Multi-Level Security policy used by the U.S. military is:

$$\Box(\forall X, Y \cdot ((X \rightarrow Y) \Rightarrow (X \leq Y)))$$

Figure 1.4 : An Information Flow Policy

where

$$X \rightarrow Y$$

is an uninterpreted predicate that represents the fact that information flows from X to Y. The policy in Figure 1.4 states that if information flows from X to Y then X must be at the same level or lower than Y. An implementation of this policy might allow information to flow from Object 2 to either Subject A or Subject C, but it would be a violation of the security policy to allow information to flow from Object 2 to Subject B since the level of Subject B (Unclassified) is lower than the level of Object 2 (Top Secret). Implementations of the access-control model or Multi-Level Security policy have been very popular for protecting computers in a stand-alone environment.

1.2 Network Security

The move from stand-alone computer systems to distributed systems produced many new possibilities in computing, but also a host of new security concerns - notably the vulnerability of communications. This is due to the ease with which an intruder can tap into the network and observe messages in transit. By eavesdropping on the network, an intruder

could learn the contents of the messages that he sees, or he could draw additional conclusions by observing which users are communicating with each other, when they transmit their messages, and how often they communicate. For example, knowing that Carol is communicating regularly with members of an Alcoholics Anonymous group may allow an intruder to posit some information about Carol, even if he can't read the messages she is sending or receiving. Likewise, a substantial increase in the number of transmissions at an air force base may indicate that some maneuver or attack is imminent.

Entities that observe messages on the network but do not interfere with them are called *passive* intruders. In dealing with passive intruders we typically try to hide the source, destination, and contents of messages from them. Cryptography, the scrambling of messages so that they will be unreadable to anyone but the intended recipient, is often used to protect network communications from passive intruders.

In addition to passive intruders there may be *active* intruders on the network. Active intruders have all the capabilities of passive intruders, and in addition, they can interfere with messages by creating, modifying, destroying, retransmitting, delaying, or misdirecting them. A possible attack by an active intruder might be to use an ATM to withdraw money from one's bank account, and then to delete the debit message from the ATM to the bank after receiving the money. One could also make a deposit and then either modify the resulting message to credit the account with a greater amount than was actually deposited or replay the unmodified message several times.

It is sometimes possible to detect active intruders, but as in the case of passive intruders, many of the strategies used to defend against them are designed to mitigate or render their actions innocuous. For example, one branch of cryptography deals with *one-way hash functions* (discussed in the next section) that can be used to make it very difficult to modify a message without being detected. Timestamps can help the recipient of a message determine whether the message has been delayed or replayed. As in the case of

passive intruders, cryptographic techniques can provide valuable safeguards against some attacks by active intruders. Since cryptography has become such an important component of network security, we present a brief overview in the next section.

1.3 Cryptography

Cryptography is the art of creating and using *cryptosystems,* which are methods of disguising messages so that only certain people can see through the disguise. The origins of cryptography are normally traced back to the dawn of human history. According to Kahn [KAH67], cryptography may be as old as the written word since writing may have originally been used to transform a spoken message which anyone within earshot would be able to hear and understand into a form that would be less public and understandable only to a select group of people. As alphabets were standardized and more people became literate, cryptographers developed new and more cunning means to camouflage their messages.

1.3.1 A Classic Cryptosystem

Perhaps the most widely-known cryptosystem is the Caesar cipher. Legend has it that Julius Caesar disguised the messages he sent to his generals by replacing every ‘A’ in the message with a ‘D’, every ‘B’ with ‘E’, every ‘C’ with ‘F’, and so on through the alphabet. So if Caesar wanted to tell his generals to “ATTACK AT DAWN”, he would send them the message “DWWDFN DW GDZQ”. Although the enemy might capture this message in transit he would not have been able to understand what it said. When one of Caesar’s generals received this message, he knew that replacing each letter in the message with the letter appearing three places before it in the alphabet would reveal the intended message.

This simple cryptosystem can be used to define many of the basic terms in cryptology. The original message, “ATTACK AT DAWN”, is called the *plaintext*, and the disguised message, “DWWDFN DW GDZQ” is called the *ciphertext*. We call the procedure for converting the plaintext into ciphertext *encryption*. In the case of the Caesar cipher, we encrypt a message by replacing each letter with the third letter after it in the alphabet. The procedure for converting the ciphertext into plaintext is called *decryption* and is accomplished in the Caesar cipher by replacing each letter with the third letter before it in the alphabet.

Cryptographers have usually chosen to make the algorithms for encryption and decryption public but to make their results depend upon some value known as a *key*. This is done to allow other cryptographers to examine proposed cryptosystems and convince themselves that the protection offered by a cryptosystem is based on the secrecy of the key and not the secrecy of the cryptographic algorithm. If this is the case, then the only factor that determines whether or not someone can decrypt a message is whether they know the proper key. If we define the encryption procedure for the Caesar cipher as “shift forward by n” and the decryption procedure as “shift backwards by n” then n is the key for the cipher. Caesar supposedly used n=3, but any value from the set {1, 2, ..., 25} could be used as a key. This set of usable keys is usually referred to as a cryptosystem’s *keyspace*. The problem with having a keyspace with so few elements is that someone who doesn’t know which key was used can try decrypting the ciphertext with each possible key until a message that makes sense (and is almost certainly the plaintext) is produced.

1.3.2 Codes and Ciphers

Most cryptographic algorithms can be classified as either codes, ciphers, or a combination of the two. We have already seen an example of a cipher in the preceding section on the Caesar cipher. In a *cipher*, some procedure is used to transform a block of plaintext into a block of ciphertext. A *block* is the fixed-size unit on which a cryptosystem

operates. It could be a single character, as in the Caesar cipher, or two or more characters, as in some other ciphers. Usually the plaintext blocks are transformed into blocks of ciphertext by substitution and/or transposition. As we saw in the Caesar cipher, *substitution* is performed by applying some function to the plaintext block and the key in order to produce a block of ciphertext which replaces the block of plaintext. *Transposition* does not involve changing the plaintext blocks but instead shuffles the blocks into a new order that depends on the key and possibly the plaintext. A simple transposition scheme would be to enter the plaintext into a matrix with n columns and one letter per box.Ciphertext could then be created by reading down the columns from left to right. If we were to encipher the message "ATTACK AT DAWN" in this way we could start by choosing n=5 for the key. Next, we would create a matrix with five columns and enter the plaintext as shown below.

A	T	T	A	C
K		A	T	
D	A	W	N	

Figure 1.5 : Entering the Plaintext into the Matrix

We could then read the ciphertext, "AKDT ATAWATNC " from the matrix. To decrypt the ciphertext we would enter it into a matrix with three columns (the length of the message divided by the key) and recover the plaintext by reading down the columns from left to right. Many ciphers include both substitution and transposition operations and perform several rounds of each to produce the final ciphertext.

A	K	D
T		A
T	A	W
A	T	N
C		

Figure 1.6 : Entering the Ciphertext in the Matrix

The other class of cryptosystems are called *codes*. Codes rely on a *codebook* that specifies one or more *codewords* for each word that might be used in a message. Codewords can be random numbers, strings of characters, or other symbols, but each codeword should map to only one plaintext word. It is assumed that the sender and receiver each have a copy of the codebook and that the sender creates the ciphertext by replacing each word in the plaintext with a corresponding codeword. The recipient can then translate the encrypted message back to plaintext by referring to codewords in the codebook. Obviously, if an adversary obtains a copy of the codebook they will be able to decode all of the messages that utilize that code.

1.3.3 Modern Cryptosystems

In this century, astounding advances in communication and computing power have served as a catalyst for the development of new cryptosystems. Not only has the speed and ease with which computers manipulate symbols made these new and highly complex cryptographic algorithms feasible, but it has been matched by scholarly theoretical work on the strengths and weaknesses of the cryptosystems. As a result, this century has witnessed the evolution of cryptography from an art to a science, as well as the beginnings of standardization normally associated with a scientific field along with some dissemination

and use of the technologies which have been produced. We can expect the spread of cryptography to continue well into the next century when it will almost certainly have a direct influence on the everyday lives of a large number of people. In this section we give a brief overview of some of the most important modern cryptosystems and refer interested readers to more detailed sources for additional information.

In 1977, the National Bureau of Standards (NBS), now renamed the National Institute of Standards and Technology (NIST), established a Federal Information Processing Standard (FIPS) for data encryption. Often referred to as the Data Encryption Standard, or simply DES, the algorithm was a descendant of an encryption scheme named "Lucifer" that had been developed at IBM. After consulting with the National Security Agency (NSA) and making a few changes, the IBM team submitted their Lucifer variant to NBS and it was accepted. DES is still a FIPS, having been recertified by NIST in 1993.

DES is a symmetric-key cipher that breaks a message into 64-bit blocks and uses a 56-bit key to encrypt them. The term *symmetric-key* alludes to the property that the same key used to encrypt the plaintext must be used to decrypt the ciphertext. Symmetric-key cryptosystems are sometimes also called *secret-key* systems since the communicating parties must share a key that is kept secret from everyone else. The algorithm is performed in *rounds* with each plaintext block going through one substitution operation followed by one transposition operation sixteen times to produce a 64-bit block of ciphertext. DES is usually referred to as a *bulk encryption* algorithm since it can encrypt a large amount of plaintext relatively quickly due to the simplicity of the operations used to substitute and transpose bits.

DES has been extensively studied since its adoption by the U.S. government and is currently the most well-known and widely used cryptosystem in the world. Most researchers agree that DES is very hard to cryptanalyze with its most often cited "weakness" being that the keyspace contains only 2^{56} elements, making a brute-force

search for the key feasible but still rather expensive. More recently developed symmetric-key block ciphers use keys of length 80 (Skipjack) or 128 (IDEA) bits, making an exhaustive search for the correct key impossible.

In addition to DES and the other secret-key algorithms mentioned above there are a number of important *public-key* algorithms. These types of cryptosystems were first proposed in 1976 [DH76] and are based on *trap-door one-way functions*. A function, $f(x) = y$, is *one-way* if, given x, it is easy to compute y, but given y, it is difficult to determine x. We often describe this one-way property by saying that the function is easy to compute in the forward direction but difficult to compute in the backward direction. However, a *trap-door* in a one-way function allows anyone who knows the trap-door to compute in the backward direction easily. Public-key cryptosystems are based on supposed trap-door one-way functions that enable anyone who knows the trap door to perform the function easily in both directions, while anyone lacking the trap door can perform the function only in the forward direction. Using this type of function a cryptosystem can be defined in which the forward direction is used for encryption and the inverse direction is used for decryption. The *public key* in such a system gives information about the particular instance of the one-way function, and the *private key* gives information about the trap door. For a public-key cryptosystem to work properly, a user's public-key information must be widely publicized so that anyone can encrypt a message to that user, but the user should keep the private-key information secret so that only he can decrypt messages encrypted with his public key. Other properties that we would like a public-key system to possess include:

- The public/private key pair is unique to the user
- Encrypting any message, M, with the public key and then decrypting the result with the private key yields M
- Deriving the private key from the public key is as hard as reading M

- The encryption, decryption, and key-generation routines are easy to compute

With these conditions in place, a public-key cryptosystem usually operates as follows. If Alice wants to send a coded message to Bob, she encrypts the message using Bob's public key and sends the ciphertext to Bob. When Bob receives the ciphertext he can use his private key to decrypt it and recover the plaintext. Note that anyone can send an encrypted message to Bob, since we assume that everyone has his public key. Only Bob, however, can read messages encrypted with his public key because he alone knows his private key. One advantage of public-key cryptosystems is that Alice and Bob do not have to agree on a key prior to communicating, as is the case with symmetric-key algorithms. This is a valuable property since private keys do not ever need to be transmitted or revealed to anyone, whereas in secret-key systems, there is always a chance that an enemy could discover the key while it is being transmitted or by compromising the other communicating party.

Another major advantage of public-key cryptosystems is that they can provide a method for digital signatures. A *digital signature* can be thought of as proof of authorship of a document or at least agreement with its contents. For a digital signature to be useful it must have the following properties:

- Authenticity - the signer deliberately signed the document
- Unforgability - only the signer can produce his signature
- Nonreusability - a signature cannot be moved to another document
- Nonrepudiation - the signer cannot reasonably claim that they didn't sign a document bearing their signature

In a public-key system, a user produces a digital signature by encrypting a document (or a hash of it) with her private key. This signature can then be verified by anyone using the signer's public key. By contrast, a secret-key system requires the sharing of some key and

sometimes requires trust of a third party as well. A sender can then repudiate a previously signed message by claiming that the shared key was somehow compromised by one of the parties sharing the secret.

The main disadvantage of public-key cryptography is speed. Public-key systems tend to be from 100 to 1000 times slower than secret-key systems, depending upon whether or not they are both implemented in hardware or software. This is due to the greater complexity of the enciphering operations that are performed in most public-key systems. For this reason, public and secret key cryptosystems are often used together with the public-key system used for digital signatures and to transmit a key that will then be used by a secret-key algorithm to encrypt subsequent communications.

The most famous public-key algorithm, RSA [RSA78], is named for its three inventors: Rivest, Shamir, and Adleman. It is based on arithmetic operations being performed on very large integers. RSA, like all practical public-key cryptosystems proposed to date, is based on functions that are believed to be one-way, but have not been proven to be so. This means that it is theoretically possible that an algorithm will be discovered that can compute the inverse function easily without knowledge of a trap door, rendering any cryptosystem based on that one-way function insecure and useless. In the case of RSA, the one-way nature of the functions is based upon the belief that the large integers used must be factored to break the cryptosystem and that factoring large integers like the ones used in RSA is intractable. Both of these conjectures have never been proven, but neither has there been any success in disproving them. As with DES, RSA is currently in wide use and thought to be a strong encryption algorithm.

1.4 Cryptographic Protocols

While cryptographic algorithms are valuable tools they do not provide a complete solution to the network security problem. In section 1.2 we discussed a number of ways in which intruders can interfere with communications on a network including such things as:

- Intercepting messages and attempting to understand their contents
- Inferring information from messages based on their source, destination, frequency, routing, or timing
- Introducing new messages into the network or resending a copy of an old message
- Modifying messages or deleting them altogether
- Misdirecting or delaying messages

Simple utilization of a good cryptosystem will not protect users communicating over a network from all of the hazards on this list.

In addition to the concerns listed above, there are some higher-level issues that usually fall under the heading of network security. Most importantly, large-scale distributed systems have seldom designated a central authority to control all of the resources in the system and enforce a single system-wide security policy. This was a role filled by a single trusted operating system in the case of stand-alone computer systems, but that may not be viable or desirable in distributed systems. Without this authority, many fundamental issues that have a direct impact on system security become much more difficult to resolve. For instance:

- Who is this entity that has asked for my checking account number and is claiming to represent my bank?
- How can Alice and I agree on a new secret key so that we can have a secure conversation over the network?

• How can I make sure that Bob doesn't deposit my digital payment until I receive a copy of the software he sold me and vice versa?

Answering questions such as these will probably require bringing some cryptography to bear on the problem, but clearly something else is needed as well.

1.4.1 What is a Cryptographic Protocol?

Part of the solution to these problems has been the development of network security protocols. A *protocol* is an agreed upon sequence of actions performed by two or more entities in order to accomplish some mutually desirable goal. Many of the protocols that have been proposed to address issues like those listed above make use of cryptographic techniques and are referred to as *cryptographic protocols*. The first such protocols were proposed in 1978 to achieve several different types of authenticated communication over a computer network.

One of these early protocols dealt with interactive communications between two principals using machines that were physically separated but that were linked by a large network. The goal of this protocol was to enable the communicating parties to prove their identities to one another and to agree on a session key that would be used to encrypt all subsequent communications. These two tasks are often called *authentication* and *key-distribution*, respectively. The following specification of the protocol is taken directly from [NS78].

$$
\begin{array}{ll}
A \rightarrow AS: & A, B, I_{A1} \\
AS \rightarrow A: & \{I_{A1}, B, CK, \{CK, A\}KB\}KA \\
A \rightarrow B: & \{CK, A\}KB \\
B \rightarrow A: & \{I_B\}CK \\
A \rightarrow B: & \{I_B - 1\}CK
\end{array}
$$

Figure 1.7 : The Needham and Schroeder Private-Key Protocol

This protocol involves three entities: the two users *A* and *B*, and AS, an authentication server that they both trust. *KA* is a secret key shared between *A* and *AS*, and *KB* is a secret key shared between *B* and *AS*. *CK* is a session key that the authentication server generates for A and B to use to encrypt their communications after they are done authenticating each other. I_{A1} and I_B are nonces generated by *A* and *B* respectively. A *nonce* is an unpredictable value that is generated by a principal, used once in a protocol, and then never used again.

In the first step of the protocol *A* sends a plaintext message to the authentication server. The message contains *A*'s name, *B*'s name, and a nonce generated by *A*. In the second step, the authentication server replies to *A* with a message encrypted using the key *KA*. In this message the authentication server includes the nonce that *A* sent, *B*'s name, a session key that the *AS* created, and something that Needham and Schroeder call a *ticket*. The ticket contains the session key and *A*'s name encrypted under *KB*. Upon receipt of this message *A* can remove the outer layer of encryption by decrypting it with *KA*. *A* then checks to see that the nonce is the same one which was sent in step one. If so, *A* then knows that the message is *fresh* because it must have been generated after the time when *A* created the nonce. Since *A* could be attempting to establish communication sessions with a number of agents simultaneously, *A* also checks for *B*'s name to make sure that this is a reply to her request to talk to *B*. From this same message *A* also learns *CK*, the conversation key created by the authentication server, and the ticket. Since *A* does not know *KB* she cannot read the contents of the ticket, but she can blindly forward it to *B* which she does in step three.

When *B* receives the message *A* sent in step three, he decrypts it and discovers that *A* wishes to talk to him and that the session key is *CK*. In step four, *B* generates a nonce, encrypts it under the session key, and sends the result back to *A*. This is sometimes referred to as a *challenge* since *B* is challenging *A* to prove that she knows the session key. In the final step of the protocol, *A* receives *B*'s message, decrypts it, subtracts one from the nonce, encrypts that value with the session key, and sends the result to *B*. After decrypting this

reply and checking that *A* has indeed returned one less than the nonce sent in step four, *A* has demonstrated her knowledge of the session key to *B*. So after successfully completing this protocol *A* and *B* each believe that they share a conversation key known only to them and the authentication server whom they trust. Furthermore, they have *authenticated* each other - *A* believes that the agent she will be talking to with *CK* is *B* because only *B* could have decrypted the ticket to find out the conversation key, and *B* believes that the agent he will be talking to with *CK* is *A* because only *A* could have decrypted the message from the authentication server that contained the ticket.

1.4.2 Problems with Cryptographic Protocols

Three years after it was proposed, another group of researchers discovered a way for an intruder to deceive an agent using the Needham and Schroeder Private-Key Protocol into believing that she was communicating with one of the other principals when she was actually communicating with an intruder. In [DS81], Denning and Sacco point out that the reason that principals should use conversation keys is so that any damage done by the compromise of a key would be limited to a single session. A conversation key might be compromised through cryptanalysis or by breaking into the *AS* or into *A*'s or *B*'s computer to steal a key. Conversation keys are intended to discourage intruders from even attempting these types of attacks by making them expensive, risky, and time-consuming enough so that the value of breaking a conversation key is not worth the effort. However, it may be worthwhile for an enemy to compromise a conversation key if it jeopardizes more than a single past session.

Denning and Sacco assume that some intruder, named *I*, has recorded a run of the Needham and Schroeder Private-Key Protocol by agents *A* and *B* and subsequently compromised the conversation key, *CK*. The key may not be compromised for many months or years after the session between the two had ended, but from that point on the intruder can

pass himself off as *A* to *B* at will.

$$I \rightarrow B: \quad \{CK, A\}_{KB}$$

$$B \rightarrow I: \quad \{I'_B\}_{CK}$$

$$I \rightarrow B: \quad \{I'_B - 1\}_{CK}$$

Figure 1.8 : Denning and Sacco's Attack on the Needham and Schroeder Protocol

The attack, illustrated in Figure 1.8, begins with the intruder sending *B* a copy of the old message from *A* suggesting *CK* as a conversation key. *B* has no way of knowing that this isn't a valid connection request from *A* or that he has already used *CK* as a session key in the past. Adhering to the protocol, *B* replies with a new nonce challenge for *A* which the intruder intercepts and is able to decrypt. The intruder can then generate the proper response and convince *B* that he is talking with *A* using the key *CK*.

This attack was interesting for several reasons. Most importantly, the protocol had been studied for quite some time and was actually being used in a popular network application, so it was surprising and disturbing that the flaw had gone undetected. Secondly, the nature of the flaw was interesting because there was no problem with the cryptographic algorithm (DES). Rather, there was a flaw in the protocol which is rather short and was thought to be very simple.

This pattern of protocol design, scrutiny, implementation, and flaw discovery has been repeated an alarming number of times in the years that have followed Denning and Sacco's discovery. This has been the main problem with cryptographic protocols: too many of them have had flaws revealed later rather than earlier, and this has caused many people to become skeptical of cryptographic protocols in general.

1.4.3 The Importance of Protocol Correctness

When flawed security protocols are implemented and used there can be grave consequences. Cryptographic protocols are already being used for electronic funds transfers, and voting protocols have been proposed and may be used for elections in the near future. If an unscrupulous person is the first to discover a flaw in one of these protocols, he could exploit it to steal a large amount of money or influence the results of an election. Since the payoff for breaking some of these protocols is so large, we should expect these protocols to be subject to serious, sophisticated, and well-funded attacks. Withstanding these attacks will require a variety of analysis techniques to reduce the number of flaws in protocols and to try to ensure that any flaws that may remain will not be easy to find.

There are also less tangible costs that can be attributed to flawed protocols. For example, while internet use has grown exponentially in the past decade, commerce over the internet has not expanded at nearly the same rate. Well-publicized flaws in several protocols for internet commerce have convinced most people that they cannot trust the security mechanisms that are in place, and commerce on the internet has suffered accordingly. In order to begin to realize the vast potential of commerce on the internet, the research community will have to convince the general public that the best available methods are being used to design and validate the protocols.

1.5 Our Approach to Protocol Analysis

As important as correctness is for cryptographic protocols, we must acknowledge that there is a fundamental limitation on what we can hope to achieve through analysis. Protocols may be short and somewhat simple computer programs, but they are programs nonetheless, and we know that we cannot prove their correctness in the general case.

Another problem with trying to prove protocol correctness is the difficulty in specifying the correctness criteria. Given protocol P with goal G, it is not sufficient to show that P achieves G. What must really be shown is that for all possible sequences of actions by all possible intruders, P achieves G. That can be a very hard specification to reason about, particularly since we assume that intruders are intelligent agents who are actively trying to subvert the protocol.

For the reasons given above, our approach is intended to help designers reduce the number of flaws in their protocols, and especially to point out some of the flaws that would be easiest for an adversary to identify and exploit.

1.5.1 Overview of Our Approach

A more detailed description of our method is given in Chapter 5, but our basic strategy is to specify the protocol(s) to be analyzed in a formal language that is designed to express cryptographic protocols. In this language, the assumptions, actions, and goals of each protocol are stated explicitly. We then negate the goals to establish a failure condition for each protocol. Since there is a formal semantics defined for the language we are using, we can give a formal semantic definition of what it means for a protocol to fail. This definition of failure can be represented as a logical theorem, the proof of which indicates that the protocol is vulnerable to an attack.

Our approach is to employ an automatic theorem prover to attempt to find a constructive proof of the theorem and then modify the protocol by adding or deleting valid statements according to the proof. If the proof succeeds, we have not only proven that the protocol is flawed, but we have also generated code that implements the attack and exploits the flaw. Unfortunately, if the theorem prover cannot prove the theorem in some reasonable amount of time, we cannot say that the protocol is sound. In Chapter 5 we discuss some

weaker conclusions that we can draw when we cannot prove that the protocol is flawed.

1.5.2 Advantages of our Approach

One of the main strengths of our methodology is its power. Given an adequate set of axioms and enough time, our approach can discover any attack that exists for a given protocol and failure condition. For example, our approach has been able to discover a previously unknown type of attack based on the interactions among different protocols. Even if our methodology does not discover a flaw in the amount of time it is given to run, we can make a useful statement about the minimum length of a constructive proof for any attack that might exist on the protocol (for the given failure condition and axiom set) as a result of its analysis.We will revisit each of these points in Chapter 7 after we have presented our method and results in Chapters 5 and 6, respectively.

1.6 Outline of this Dissertation

The organization of this dissertation is as follows. In Chapter 2 we review the work that has already been done on cryptographic protocol design and analysis. In Chapters 3 and 4, we review the practical and theoretical background material upon which our approach is built, followed by a detailed presentation of our methodology in Chapter 5. We demonstrate our method in Chapter 6 by presenting the results of our system's analysis of a number of cryptographic protocols from the literature. Finally, in Chapter 7 we offer our conclusions and note some possible extensions to our work that may be undertaken in the future.

Chapter 2

Related Work

In this chapter we review the various cryptographic protocol analysis techniques that have been developed to date. These methods can be roughly divided into four categories: ad-hoc strategies, general-purpose specification and verification systems, special-purpose state-based approaches, and logical analysis. Each of the next four sections examines one of these categories and presents, in roughly chronological order, the methods that comprise it.

2.1 Ad-Hoc Strategies

When we consider all of the techniques that have been used to analyze cryptographic protocols in the past twenty years, informal methods have probably been the most commonly used and the most successful at identifying and eliminating flaws. This can be attributed to the relative youth of the field, as researchers have only recently moved from informal towards more formal approaches, and many of these formal methods have not yet had their full impact. The move to formality is, of course, necessary in this field, and we can expect many of the formal systems that we will discuss in later sections to overtake and

replace these ad-hoc strategies in the near future.

2.1.1 Denning and Sacco

The flaw (see Figure 1.8) in the Needham and Schroeder Private-Key Protocol [NS78] found by Denning and Sacco [DS81] is the earliest example of informal analysis uncovering a flaw in a protocol. Denning and Sacco realized that while the ticket sent to *B* in step 3 of the protocol (see Figure 1.7) must have been good when it was generated by the authentication server, there is nothing about the ticket that tells *B* how long ago the *AS* may have generated it. If the ticket is very old, it is probably not good anymore since an intruder might have had time to compromise the session key contained in it. Denning and Sacco suggest that this weakness can be avoided if a timestamp is placed in the *AS*'s reply to *A* and in the ticket for *B*. The use of timestamps eliminates the need for nonces to prove the freshness of messages and thereby reduces the number of messages in the protocol from five to three.

$$A \rightarrow AS: \quad A, B$$

$$AS \rightarrow A: \quad \{B, CK, T, \{A, CK, T\}_{KB}\}_{KA}$$

$$A \rightarrow B: \quad \{A, CK, T\}_{KB}$$

Figure 2.1 : Denning and Sacco's version of the Needham and Schroeder Protocol

In the above specification, *T* represents a *timestamp* - an explicit statement of the local time at which the message was generated. Denning and Sacco assume that the clocks of *A*, *B*, and the *AS*, are loosely synchronized so that *A* and *B* can check, respectively, that the message from the *AS* and the ticket were not created too far in the past.

2.1.2 Simmons

Another example of informal analysis is found in [SIM85], in which Simmons pre-

sents the results of his analysis of the TMN Protocol [TMN89]. The TMN Protocol is a key distribution scheme by which a pair of ground stations in a mobile communication system obtain a common session key, through the mediation of a trusted server. The protocol assumes that the server has a public/private key pair and that the server's public key is known to all of the ground stations. A pseudocode specification of the TMN Protocol is given below.

$$A \rightarrow S: \quad A, B, \{r1\}S_{Public}$$
$$S \rightarrow B: \quad A$$
$$B \rightarrow S: \quad \{r2\}S_{Public}$$
$$S \rightarrow A: \quad \{r2\}r1$$

Figure 2.2 : The TMN Protocol

In step 1 of the protocol, user *A* notifies the server that she wishes to communicate with *B*. She sends her name, *B*'s name, and a random number encrypted with *S*'s public key. When the server receives this message, it decrypts the random number using its private key and stores it as the key-encryption key. In step 2, the server notifies *B* that *A* wishes to talk to him. User *B* then generates a random number, encrypts it with *S*'s public key, and sends the result to *S* in step 3. The server decrypts *B*'s reply, encrypts *B*'s random number using *A*'s random number as a key, and sends this value to *A*. After decrypting the message from the server, *A* will know *B*'s random number which can then be used as a session key to encrypt subsequent communications.

Simmons identified two security flaws in this protocol. The first results from the fact that no secure authentication is used between parties, so an intruder, *I*, can cause the server to convince *A* that a key generated by him was generated by *B*. He can accomplish this by intercepting *B*'s reply to the server in step three and replacing it with the encryption under *S*'s public key of a random number he has generated himself. The server will decrypt this

message, encrypt *I*'s random number with *A*'s key-encryption key, and send the result to *A*. When *A* receives this message and decrypts it she will think that the intruder's random number is the session key she should use to converse privately with *B*. This attack is illustrated in Figure 2.3.

$$
\begin{array}{ll}
A \rightarrow S: & A, B, \{r1\}S_{Public} \\
S \rightarrow B: & A \\
B \rightarrow I: & \{r2\}S_{Public} \\
I \rightarrow S: & \{r3\}S_{Public} \\
S \rightarrow A: & \{r3\}r1
\end{array}
$$

Figure 2.3 : Simmons' First Attack on the TMN Protocol

The second flaw in the TMN protocol is slightly more complex because it relies on two properties of the cryptosystems used in the TMN Protocol. For the public-key algorithm used, it is the case that encryption distributes over multiplication:

$$\{r1\}S_{Public} \times \{r2\}S_{Public} = \{r1 \times r2\}S_{Public} \tag{2.1}$$

For the secret-key algorithm we have:

$$\{X\}Y = \{Y\}X \tag{2.2}$$

Given these two axioms about the cryptosystems, the attack proceeds as follows. There are two dishonest ground stations: the *cheater*, *C*, and his *partner*, *P*. Before they attempt to attack the protocol each generates a random number, stores it, and sends a copy to the other. The number created by the cheater is called the *cheater key* and the other is referred to as the *partner key*. After they have exchanged these values they wait until station *A* decides it needs to talk with station *B*. At this time *A* performs the first step of the TMN protocol and send a message to *S*.

$$A \rightarrow S: \qquad A, B, \{r1\}S_{Public}$$

The cheater sees this message in transit and makes a note of the value $\{r1\}S_{Public}$ for later use. The cheater then encrypts the cheater key under the server's public key and multiplies it by the value she has just learned. The result is sent as the third field in the cheater's next message, which is a request to the server for a conversation with cheater's partner.

$$C \rightarrow S: \qquad C, P, (\{cheaterkey\}S_{Public} \times \{r1\}S_{Public})$$

The server receives the cheater's request, decrypts it, and notifies P that C wants to talk to him. The partner replies with the partner key encrypted under the server's public key.

$$P \rightarrow S: \qquad \{partnerkey\}S_{Public}$$

The server receives the message, decrypts it, encrypts the partner key using the value sent in the cheater's first message, and sends the result back to the cheater.

$$S \rightarrow C: \qquad \{partnerkey\}(\{\{cheaterkey\}S_{Public} \times \{r1\}S_{Public}\}S_{Private})$$

After receiving this message the cheater decrypts it using the partner key and divides the result by the cheater key which gives:

$$\frac{\{\{partnerkey\}(\{\{cheaterkey\}S_{Public} \times \{r1\}S_{Public}\}S_{Private})\}_{partnerkey}}{cheaterkey}$$

By applying axiom 2.1 we know that this value is equivalent to:

$$\frac{\{\{partnerkey\}(\{\{cheaterkey \times r1\}S_{Public}\}S_{Private})\}_{partnerkey}}{cheaterkey}$$

By simplifying $\{\{cheaterkey \times r1\}S_{Public}\}S_{Private}$ to $cheaterkey \times r1$ we get:

$$\frac{\{\{partnerkey\}(cheaterkey \times r1)\}_{partnerkey}}{cheaterkey}$$

We can then apply axiom 2.2 to transform $\{partnerkey\}(cheaterkey \times r1)$ into $\{cheaterkey \times r1\}_{partnerkey}$ and obtain:

$$\frac{\{\{cheaterkey \times r1\}_{partnerkey}\}_{partnerkey}}{cheaterkey}$$

This can be simplified to:

$$\frac{cheaterkey \times r1}{cheaterkey}$$

which can be further simplified to:

$$r1$$

So the cheater has been able to deduce *A*'s key-encryption key and when the server sends *B*'s session key to *A*, at the end of *A* and *B*'s run of the protocol, the cheater will be able to decrypt that message and learn the session key. With knowledge of the session key the cheater can decrypt and read every message between *A* and *B* for that session. It is both surprising and a testament to Simmons' abilities that such a complex attack was discovered by informal analysis of the protocol.

2.1.3 Gong

In [GON92], Gong describes a scenario where a clock synchronization failure renders a protocol vulnerable to an attack even after the faulty clock has been resynchronized. Gong's key observation here is that when a party's clock is ahead of other clocks, its messages are postdated. Postdated messages that are sent out while the clock is out of synchro-

nization can be intercepted by an intruder and stored until the timestamp on the message becomes current, at which time the intruder can replay the message. This replay could occur even after the faulty clock has been brought back into synchronization with the other clocks in the system.

Gong illustrates the vulnerability of the Kerberos Authentication Protocol to this type of attack if a clock falls out of synchronization, as the Kerberos system assumes they can't:

> Suppose a client had obtained all necessary credentials to use a file server. Also suppose that, later, the clock on the client's workstation was five hours ahead of the clock at the file server when the client tried to initiate a connection with the server by composing a request message, which included a local timestamp to indicate that the request was current. The client had now generated a postdated authenticator. An adversary blocked this request message from reaching the server. The client got no response and thought that an omission or performance failure had occurred. Exactly five hours later, when the client had already left, the adversary replayed the suppressed message from the same workstation (with the same network address) and established a connection in the client's name.

Gong calls this type of attack a *suppress-replay attack* and notes that it cannot be detected unless the recipient of the message can be notified before the attack can be mounted. In order for these warnings to do any good, it is necessary not only to detect loss of clock synchronization early, but also to ensure that the messages warning of the attack get through quickly.

2.1.4 Abadi and Needham

The final piece of work we discuss in this section on ad-hoc methods is the well-known paper entitled "Prudent Engineering Practice for Cryptographic Protocols" [AN94]. In this paper, Abadi and Needham present principals for the design of cryptographic protocols. While they do not claim that following these guidelines will guarantee correctness of the protocols developed, they do argue that their principals are helpful and that "adherence to them would have avoided a considerable number of published errors."

2.1.4.1 Abadi and Needham's Explicit Naming Principle

One of Abadi and Needham's principles is the following:

> Principle 3 - Explicit Naming
>
> The identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

Abadi and Needham use a key exchange protocol proposed by Denning and Sacco [DS81] to demonstrate the dangers of not following this guideline. The protocol is given below:

$$A \rightarrow S: \quad A, B$$
$$S \rightarrow A: \quad CA, CB$$
$$A \rightarrow B: \quad CA, CB, \{\{K_{AB}, T_A\}A_{Private}\}B_{Public}$$

Figure 2.4 : Denning and Sacco Key Exchange Protocol

In the first two messages of this protocol, *A* obtains from *S* certificates *CA* and *CB* that prove that K_a and K_b are the public keys of *A* and *B* respectively. The exact form of *CA* and *CB* is not important for Abadi and Needham's purposes. In the third message *A* sends these certificates to *B* along with a session key, K_{ab}, which will be used to encrypt subsequent communication between *A* and *B*, and a timestamp, T_a. The third field of this last message is signed with *A*'s private key (to prove that *A* sent it) and encrypted with *B*'s public key (to keep the contents secret). Denning and Sacco intend that B should know that the third message of the protocol was intended for him because it is encrypted with his public key, but Abadi and Needham demonstrate how this might not be the case. If we assume that an intruder has a valid certificate for his public key from *S* and that *A* engages in the protocol with that intruder, then *A* will send the following message to the intruder in step 3:

$$A \rightarrow I: \quad CA, CI, \{\{K_{AI}, T_A\}A_{Private}\}I_{Public}$$

The intruder can then remove the outer layer of encryption and re-encrypt with *B*'s public key. The intruder can then start a conversation with B pretending to be A. The final message that the intruder sends to B is:

$$I \rightarrow B: \qquad CA, CB, \{\{K_{AI}, T_A\}A_{Private}\}B_{Public}$$

Upon receipt of this message, *B* will believe that the message is from *A*, and *B* might then send out sensitive information intended for *A,* encrypted under K_{AI} which is known by the intruder. According to Abadi and Needham, the problem with Denning and Sacco's protocol is that the Explicit Naming principle has not been followed. They say that "[t]he intended meaning of Message 3 [of the Denning and Sacco protocol] is roughly 'At time T_a, *A* says that K_{ab} is a good key for communication between *A* and *B*.'" Abadi and Needham suggest that the name of the principal for whom message three is intended, be mentioned explicitly as in:

$$A \rightarrow B: \qquad CA, CB, \{\{B, K_{AB}, T_A\}A_{Private}\}B_{Public}$$

With this modification, the protocol is no longer vulnerable to the given attack.

2.1.4.2 Abadi and Needham's Signing Encrypted Data Principle

Another recommendation that Abadi and Needham make is that secret data be signed before it is encrypted for privacy. They state this principle as follows:

> Principle 5 - Signing Encrypted Data
>
> When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. On the other hand, it is proper to infer that the principal that signs a message and then encrypts it for privacy knows the content of the message.

To illustrate the importance of this principle, Abadi and Needham examine a protocol from the CCITT X.509 standard that encrypts secret data and then signs it. The protocol of interest is the simple one-message protocol that is intended for signed, secure

communication between two principals, assuming that each knows the public key of the other.

$$A \rightarrow B: \qquad A, \{T_A, N_A, B, X_A, \{Y_A\} B_{Public}\} A_{Private}$$

Figure 2.5 : CCITT X.509 One-Message Protocol

Here T_A is a timestamp, N_A is a nonce, and X_A and Y_A are user data. The protocol is intended to ensure B that A sent X_A and Y_A and to guarantee the privacy of Y_A. However, as Abadi and Needham point out "although Y_A is transferred in a signed message, there is no evidence to suggest that the sender is actually aware of the data sent in the private part of the message. This corresponds to a scenario where some [intruder] intercepts a message and removes the existing signature while adding his own, blindly copying the encrypted section within the signed message." This weakness can be avoided by applying Abadi and Needham's principle and signing the secret data before it is encrypted for privacy as shown below.

$$A \rightarrow B: \qquad A, \{T_A, N_A, B, X_A, \{\{Y_A\} A_{Private}\} B_{Public}\} A_{Private}$$

2.2 General-Purpose Formal Specification and Verification Systems

Formal specification and verification techniques have been used for some time in attempts to ensure that critical systems satisfy their requirements. Secure operating systems and safety-critical software are two areas where this approach has been applied regularly. In fact, the National Computer Security Center, which certifies systems for use in classified or other sensitive environments, requires formal specification and verification of system designs for its highest rating [DOD85].

Several objections to the formal specification and verification approach have been raised in the literature including Ken Thompson's Turing Award lecture [THO84] and DeMillo, Lipton, and Perlis' renowned paper [DLP79] entitled "Social Processes and

Proofs of Theorems and Programs". In the later work, the authors argue that "it is a social process that determines whether mathematicians feel confident about a theorem - and ... because no comparable social process can take place among program verifiers, program verification is bound to fail." They go further by stating that "scientists should not confuse mathematical models with reality - and verification is nothing but a model of believability." Many researchers in formal methods respond to this criticism as Kemmerer did in [KEM89] that "although there is some validity to these arguments, formal specification and verification techniques should not be abandoned until there is a better method to replace them." Indeed, several researchers have had notable success using general-purpose formal specification and verification systems to analyze cryptographic protocols.

2.2.1 Kemmerer

In [KEM89] and [KMM94], Kemmerer describes how cryptographic protocols can be analyzed using an existing machine-aided formal verification technique. His approach makes use of the Formal Development Methodology (FDM), which describes a system as a state-machine that can be in any one of a number of states. According to Kemmerer, "one state is differentiated from another by the values of state variables, and the values of these variables can be changed only via well-defined state transitions." Kemmerer chooses to represent the properties of the network as state constants and variables with the protocol defining state transitions. Assumptions about the cryptographic algorithms used by the protocol can be represented as axioms, and the goals of the protocol are specified as state invariants.

Given the above specification, the verification system can automatically generate a set of theorems that must be proven to guarantee that the invariants always hold. The verification system includes an automated theorem proving component which assists the user in proving generated theorems and a facility for testing the formal specification by executing it symbolically. In [KMM94], Kemmerer demonstrates his approach by using it

to analyze the TMN protocol and reproduce the Simmons flaw that we described in Section 2.1.2.

2.2.2 Merritt and Toussaint

Another approach to formal analysis of cryptographic protocols has been to use algebras to reason about knowledge. The protocol is modeled as an algebraic system which expresses the state of the participants and the intruder's knowledge about the protocol. This technique was used by Toussaint in [TOU91] to demonstrate how attacks can be detected by a principal's seeing an inconsistency between messages received and its state of knowledge of the words used in the protocol. Merritt used a similar approach in [MER83] to define *hidden automorphisms* which express an intruder's lack of knowledge about the contents of a message:

> Suppose, for example, that a principal views a message e(k,m) (denoting the encryption of m with k), where that principal does not know k. Suppose furthermore that we define an automorphism h of the space of words such that h(m)=n for some n, but all other words are left invariant. Then the set of messages known by the principal is invariant under h, (in particular h(e(k,m)) = e(k,m)). Thus effects of the automorphism are invisible to the principal, and can be used to define formally the principal's ignorance of m. [MEA94]

Merritt uses these hidden automorphisms to prove results about secrecy that are "considerably more subtle than the simple secrecy of words; for example, he is able to prove that the correspondence between votes and individual voters in a voting protocol is unknown, even when all the voters and all votes are public."

2.2.3 Lowe

Another formal method that has had success analyzing cryptographic protocols is the Failure Divergences Refinement Checker (FDR), a model checker for Communicating Sequential Processes (CSP). FDR takes as input two CSP processes, a specification and an implementation, and tests whether the implementation refines the specification. FDR has been used to analyze many types of systems, including communications protocols, distributed databases, and puzzles. In [LOW96], Lowe uses FDR to uncover a previously

unknown flaw in a reduced version of Needham and Schroeder's Public-Key Authentication Protocol shown below.

$$
\begin{aligned}
A \rightarrow B: &\quad A,B,\{N_A,A\}B_{Public} \\
B \rightarrow A: &\quad B,A,\{N_A,N_B\}A_{Public} \\
A \rightarrow B: &\quad A,B,\{N_B\}B_{Public}
\end{aligned}
$$

Figure 2.6 : Needham and Schroeder's Public-Key Protocol

The complete Needham and Schroeder Public-Key Protocol presented in [NS78] involves seven steps, but for his analysis Lowe considers only the last three. The omitted steps deal with each agent requesting and receiving the other's public key from the server. Lowe notes that these steps can be omitted if we assume that each agent already knows the other's public key. As an aside, Lowe notes that the full seven-step protocol suffers from a weakness that allows an intruder to replay old, compromised public key messages from the server, since these messages contain no proof of freshness. This flaw is well known and can be repaired easily by having the agents include nonces in their requests and then having the server return the nonces in its replies.

Lowe's analysis proceeds as follows. The two agents taking part in the protocol are modeled as CSP processes. Also part of the model is an intruder who can interact with the two legitimate principals and intercept their messages to each other. FDR is then used to test whether the protocol correctly achieves its goal of authenticating the two honest principals to one another. The result of this analysis is the discovery of an attack which allows the intruder to convince agent B that he is agent A during a run of the protocol. The attack scenario is shown in Figure 2.7.

$$
\begin{aligned}
A \rightarrow I: &\quad A,I,\{N_A,A\}I_{Public} \\
I \rightarrow B: &\quad A,B,\{N_A,A\}B_{Public} \\
B \rightarrow I: &\quad B,A,\{N_A,N_B\}A_{Public} \\
I \rightarrow A: &\quad I,A,\{N_A,N_B\}A_{Public}
\end{aligned}
$$

$$A \to I: \qquad A, I, \{N_B\} I_{Public}$$
$$I \to B: \qquad A, B, \{N_B\} B_{Public}$$

Figure 2.7 : Lowe's Attack on the Needham and Schroeder Public-Key Protocol

The attack starts when agent A begins a run of the protocol with the intruder by sending the nonce N_A encrypted under the intruder's public key. In step two, the intruder decrypts the nonce and encrypts it using B's public key, forming part of a message he will send to B claiming to be A. This is usually called a *parallel-session attack* since the intruder is taking part in two or more runs of the protocol simultaneously and uses information learned in one session to attack in another. In step three of the attack, agent B replies to the message from the intruder by encrypting N_A and a new nonce, N_B, under A's public key and sending the result back to the intruder. Since the intruder doesn't know A's private key, he cannot decrypt this message. In step four, the intruder sends the message he has just received from B to A who is waiting for I's reply in their run of the protocol. In step five agent A decrypts the message and returns N_B to the intruder encrypted using I's public key. This is exactly the value the intruder needs to construct message six and convince B that he has successfully authenticated agent A.

2.3 Special-purpose State-Based Approaches

Both Kemmerer and Lowe's work are described as *state-based approaches* since the system is always viewed as being in a distinct state[1] with the protocol and the abilities of the intruder defining state transitions. The system is then analyzed to determine whether any insecure state is reachable. Both Kemmerer and Lowe utilize existing, general-purpose formal methods for specification and analysis of protocols. Another group of researchers has chosen to follow the state-based approach as well, but they have developed special-purpose systems for dealing with cryptographic protocols.

---

1. This implicitly assumes a blocking semantics for send and receive operations.

2.3.1 Dolev and Yao

In [DY83], Dolev and Yao propose several algorithms to analyze restricted classes of cryptographic protocols. The protocols that Dolev and Yao's method applies to are limited to using public-key cryptography only, and their technique does not address any properties of the cryptosystem that the protocol uses except in so far as:

- Encryption using the public key is cancelled out by decryption with the private key and vice versa
- It is impossible to read a message encrypted with a public key unless the private key is known

Dolev and Yao define the two classes of protocols that they will consider as cascade and name-stamp protocols. In a *cascade protocol* the users can apply any number of public-key encryption or decryption operations to form the messages that they send to each other. In a *name-stamp protocol* the users are allowed to append, delete, and check names encrypted with plaintext. As in a cascade protocol, any number of public-key encryption or decryption operations can be applied in a name-stamp protocol. Dolev and Yao give the following as an example of a simple cascade protocol.

$$A \rightarrow B: \quad A, \{M\}B_{Public}, B$$
$$B \rightarrow A: \quad B, \{M\}A_{Public}, A$$

Figure 2.8 : A Simple Cascade Protocol

Dolev and Yao then give efficient algorithms that determine whether a given cascade or name-stamp protocol is secure. By secure, they refer only to the privacy of values that have been encrypted. There is no attempt to reason about the freshness of a value or the identity of the agent who sent it. For cascade protocols, they state that a protocol is secure if and only if both of the following conditions are satisfied:

- The messages transmitted between A and B always contain some layers of

encryption (under either *A* or *B*'s public key)

- In generating a reply message, each participant never applies the decrypt operator without subsequently applying the encrypt operator.

Dolev and Yao show how the cascade protocol in Figure 2.8 does not meet the first of these two requirements, and they give the following attack. The intruder intercepts *A*'s opening message to *B*. In step two, he sends this message to *B*. In step three, *B* replies with a message encrypted under the intruder's public key. The intruder can then decrypt this message and learn the value of *M*. This attack is shown in Figure 2.9 below.

$$A \rightarrow I: \quad A, \{M\}B_{Public}, B$$
$$I \rightarrow B: \quad I, \{M\}B_{Public}, B$$
$$B \rightarrow I: \quad B, \{M\}I_{Public}, I$$

Figure 2.9 : Attack on the Cascade Protocol

Although Dolev and Yao's algorithms are not applicable to a large number of cryptographic protocols, they are noteworthy for being among the earliest formal methods used to analyze security protocols and for pioneering the state-transition model for protocol analysis. The work we discuss in the next three sections was heavily influenced by the Dolev and Yao approach and has aimed to extend the state-transition method to a larger class of operators and message formats, to broaden the types of security that can be reasoned about, and to supply computer support during analysis.

2.3.2 Millen

One of the earliest systems to extend the Dolev and Yao approach was the Interrogator [MIL87] [MIL95] developed by Millen. The Interrogator is a Prolog program that takes a protocol specification and a compromise objective and then performs a search of the state space in an attempt to find a path from an initial state to a state in which the

compromise objective is satisfied. Early versions of the Interrogator were fully automatic; there was no user intervention after specifying the protocol and compromise objective.

More recent versions of the Interrogator allow user interaction during the search. For example, the Interrogator might be considering several possible state transitions at some point in the search and it will ask the user which one to try first. If the user's choice is unproductive, the program will return to the choice point and, if there are any remaining, ask for another choice. There is also a limited automatic search mode in which the program makes an arbitrary choice without asking the user, but this feature is carefully designed to ensure that the program will not fall into a loop.

The Interrogator has been able to reproduce a number of known attacks on cryptographic protocols, but it has not yet uncovered any previously unknown vulnerability in a well-known protocol. Furthermore, according to [MIL95]:

> The tool requires some training to use effectively, for two principal reasons. First, the security objective must be set up in the initial and goal state with enough information to constrain the overall penetration approach. If messages or message fields are to be replayed, for example, the user must know that and specify them in the initial state. During the interactive search phase, the attacker's use of encryption or other devices is controlled interactively, and this also requires understanding of the overall attack strategy.

2.3.3 Meadows

The NRL Protocol Analyzer [MEA91] is also based on the Dolev and Yao approach, but it takes a slightly different approach than the Interrogator. As with the Interrogator, the user specifies a protocol and an insecure state. Unlike the Interrogator, Meadows' tool is intended to help the user prove that the insecure state is unreachable from the initial state. This is done by helping the user prove that certain paths leading backwards from the insecure state go into infinite loops and therefore never reach the initial state. According to [MEA94]:

> Once these paths have been eliminated, the resulting search space is often small enough to search exhaustively. The proofs that paths lead into infinite loops are largely guided by the

user; thus the search is much less automated than in the Interrogator.

Unlike the Interrogator, the NRL Protocol Analyzer also allows an unlimited number of protocol rounds in a single path. While this makes the search space infinite, it also allows the Analyzer to discover parallel session attacks where the intruder takes part in two or more runs of the protocol simultaneously and uses information learned in one session to attack in another.

The NRL Protocol Analyzer can also be used to find flaws in protocols by generating paths to insecure states. It has been used to demonstrate several flaws that were already known to exist, including the attack on the TMN Protocol shown in Figure 2.3. More impressively, Meadows has used the Protocol Analyzer to find several previously unknown security flaws in some well-known cryptographic protocols, including an authentication flaw in Simmons' Selective Broadcast Protocol [SIM85] and a vulnerability in the Burns and Mitchell Resource Sharing Protocol [BM90].

2.3.4 Longley and Rigby

Another system for protocol analysis is that of Longley and Rigby [LR92] who developed a PROLOG program which automatically examines key management schemes for flaws. By using a simple rule-based model the package sets up a large but finite search tree with the intruder's goal as the root. Each child node in the tree represents a scenario by which the intruder could discover the information in its parent. The user of Longley and Rigby's tool can then guide the search in an attempt to eliminate leaves that correspond to impossible attacks. When a leaf is removed the search might proceed through other subtrees of the parent node. The search ends in one of two ways. If the tree has been pruned so that only the root remains, then the package reports that no attack has been found. If the search is successful and it finds some subtree that satisfies the goal at the root, the package prints out the steps needed to implement the attack.

As noted by its authors, the value of this system is that it "provides for an automatic, impartial, and exhaustive search for security-specified security loopholes." [LR92] However, Longley and Rigby go on to caution that:

> The package does not claim to be an automatic certification system for three reasons. Firstly it only tests for specified attacks, secondly there is no means of guaranteeing that all possible data, required for the attack and available to the attacker, are included in the input data, and thirdly there has been no theoretical analysis of the search technique to guarantee that it is truly exhaustive.

The Longley and Rigby system was used to find a previously unknown flaw in a draft hierarchical key management scheme designed for an electronic funds transfer point of sale (EFTPOS) network.

2.4 Logical Analysis

With more than a dozen different logics defined, logical analysis of cryptographic protocols is the most extensive of the four analytical approaches we will discuss in this chapter. According to Carnap [CAR37], a logical system is characterized by stating its formation rules and its transformation rules. The *formation rules* provide us with a list of recognized characters and a decidable means of delineating the grammatically well formed formulae. The *transformation rules* provide us with a list of axiomatic sentences and a (not necessarily decidable) means of delineating those sentences that follow from a given set of sentences called the inference rules. Many of the logics that have been developed for cryptographic protocol analysis have their roots in the various logics that had been developed previously to reason about the knowledge and beliefs of agents in a distributed system. These logics can be divided into two categories: epistemic and doxastic. Logics that deal with the knowledge set of principals are called *epistemic logics*, and those which deal with the belief set of agents are called *doxastic logics*. All epistemic logics assume that principals cannot know something that is false and so they always include the axiom "If agent X knows Y then Y is true," but no doxastic logic has a comparable axiom for belief.

Therefore, in a doxastic logic it can be a true statement that some principal believes a statement that is false.

For a cryptographic protocol to be analyzed using logic it must first be translated from its procedural representation into the language of the logic. This process is called *idealization* and has been the topic of some debate in the research community. The difficulties with idealization are two-fold. First, due to the informal manner in which protocols are usually specified, idealization itself is often an informal procedure. Without agreed-upon rules governing idealization, different researchers have sometimes idealized the same protocol in slightly different ways, leading to vastly different results during analysis. It is not always the case that the research community can agree on which idealization was "correct." The second difficulty in idealization is the need for global knowledge. According to Burrows, Abadi, and Needham, "the idealized form of each message cannot be determined by looking at a single protocol step by itself. Only knowledge of the entire protocol can determine the essential logical components of the message" [BAN89]. As we will see later, there have been many suggestions aimed at improving or eliminating the idealization procedure.

In [MEA94], Meadows gives the following overview of the use of logic in analyzing cryptographic protocols:

> In an analysis of a protocol, an initial set of beliefs [and/or knowledge] is assumed. One then uses the inference rules to determine what beliefs can be derived from the initial beliefs and the beliefs gained from participating in the protocol. If the set of beliefs [and/or knowledge] is adequate, according to some predefined notion of adequacy, then the protocol is assumed to have been proven correct. If the set of beliefs [and/or knowledge] is not adequate, then it may lead to the discovery of a security flaw in the protocol.

In the following sections we present a number of logics that have been designed to reason about cryptographic protocols. We demonstrate how they are used and compare some of their various strengths and weaknesses. Where possible, we try to discuss some of the results that analysis with these logics has yielded.

2.4.1 Burrows, Abadi, and Needham

Perhaps the best known and most widely-used logic for cryptographic protocol analysis is that of Burrows, Abadi, and Needham, commonly known as BAN logic. In [BAN89], Burrows, Abadi, and Needham describe BAN as a "simple logic [that allows] us to describe the beliefs of trustworthy parties involved in authentication protocols and the evolution of these beliefs as a consequence of communication."

2.4.1.1 The Logic

BAN logic is a many-sorted modal logic that distinguishes between different types of objects including principals, encryption keys, and formulas (also called statements). The only propositional connective in BAN logic is conjunction, which is denoted by a comma. An explanation of the basic notation of BAN logic from [BAN89] is given below. In this list, the symbols *P*, *Q*, and *R*, range over the set of principals, the symbols *X* and *Y* range over the set of statements, and the symbol *K* ranges over the set of encryption keys.

P **believes** *X*: *P* would be entitled to believe *X*. In particular, the principal *P* may act as though *X* is true.

P **sees** *X*: Someone has sent a message containing *X* to *P*, who can read and repeat *X* (possibly after doing some decryption).

P **said** *X*: The principal *P* at some time sent a message including the statement *X*. It is not known whether the message was sent long ago or during the current run of the protocol, but it is known that *P* believed *X* then.

P **controls** *X*: The principal *P* is an authority on *X* and should be trusted on this matter.

fresh(*X*): The formula *X* is fresh, that is, *X* has not been sent in a message at any time before the current run of the protocol.

$P \overset{K}{\leftrightarrow} Q$: *P* and *Q* may use the shared key *K* to communicate. The key *K* is good, in that it will never be discovered by any principal except *P* or *Q*, or a principal trusted by either *P* or *Q*.

$\overset{K}{\rightarrow} P$: *P* has *K* as a public key. The matching secret key (denoted K^{-1}) will never be discovered by any principal except *P*, or a principal trusted by *P*.

$P \overset{X}{\Leftrightarrow} Q$: The formula *X* is a secret known only to *P* and *Q*, and possibly to principals trusted by them. Only *P* and *Q* may use *X* to prove their identities to one another.

$\{X\}_K$: This represents the formula *X* encrypted under the key *K*. Formally, $\{X\}_K$ is a convenient abbreviation for an expression of the form $\{X\}_K$ from *P*. We make the

realistic assumption that each principal is able to recognize and ignore his own messages; the originator of each message is mentioned for this purpose.

$<X>_Y$: This represents X combined with the formula Y; it is intended that Y be a secret, and that its presence prove the identity of whoever utters $<X>_Y$.

The logical postulates given in [BAN89] that are used in proofs are the following.

- The *message-meaning rules* concern the interpretation of messages. Two of the three concern the interpretation of encrypted messages, and the third concerns the interpretation of messages with secrets. They all explain how to derive beliefs about the origin of messages. For shared keys, we postulate:

$$\frac{\text{P believes } \left(P \overset{K}{\leftrightarrow} Q\right), \text{P sees } \{X\}_K}{\text{P believes Q said X}}$$

That is, if P believes that the key K is shared with Q and sees X encrypted under K, then P believes that Q once said X. For this rule to be sound, we must guarantee that P did not send the message himself: it suffices to recall that $\{X\}_K$ stands for a formula of the form $\{X\}_K$ from R, and to require that $R \neq P$. Similarly, for public keys, we postulate:

$$\frac{\text{P believes } \left(\overset{K}{\rightarrow} Q\right), \text{P sees } \{X\}_{K^{-1}}}{\text{P believes Q said X}}$$

For shared secrets, we postulate:

$$\frac{\text{P believes } \left(P \overset{Y}{\Leftrightarrow} Q\right), \text{P sees } <X>_Y}{\text{P believes Q said X}}$$

That is, if P believes that the secret Y is shared with Q and sees $<X>_Y$, then P believes that Q once said X. This postulate is sound because the rules for **sees** (given below) guarantee that $<X>_Y$ was not just uttered by P himself.

- The *nonce-verification rule* expresses the check that a message is recent, and hence that the sender still believes in it:

$$\frac{\text{P believes fresh(X)}, \text{P believes Q said X}}{\text{P believes Q believes X}}$$

That is, if P believes that X could have been uttered only recently (in the present) and that Q once said X (either in the past or in the present), then P believes that Q believes X. For the sake of simplicity, X must be "cleartext," that is, it should not include any subformula of the form $\{X\}_K$.

- The *jurisdiction rule* states that if P believes that Q has jurisdiction over X then P trusts Q on the truth of X:

$$\frac{\text{P believes Q controls X,}\quad \text{P believes Q believes X}}{\text{P believes X}}$$

- If a principal sees a formula then he also sees its components, provided he knows the necessary keys:

$$\frac{\text{P sees (X,Y)}}{\text{P sees X}} \qquad \frac{\text{P sees } \langle X \rangle_Y}{\text{P sees X}}$$

$$\frac{\text{P believes}\left(P \overset{K}{\leftrightarrow} Q\right)\text{, P sees}\{X\}_K}{\text{P sees X}}$$

$$\frac{\text{P believes}\left(\overset{K}{\rightarrow} P\right)\text{, P sees}\{X\}_K}{\text{P sees X}}$$

$$\frac{\text{P believes}\left(\overset{K}{\rightarrow} Q\right)\text{, P sees}\{X\}_{K^{-1}}}{\text{P sees X}}$$

Recall that $\{X\}_K$ stands for a formula of the form $\{X\}_K$ from *R*. As a side condition, it is required that $R \neq P$, that is, $\{X\}_K$ is not from *P* himself. A similar condition applies to $\{X\}_{K^{-1}}$.

The fourth rule is justified by the implicit assumption that if *P* **believes** that *K* is his public key, then *P* knows the corresponding secret key K^{-1}.

Note that if *P* sees *X* and *P* **sees** *Y* it does not follow that *P* **sees** (*X*,*Y*), since this means that *X* and *Y* were uttered at the same time.

If one part of a formula is fresh, then the entire formula must also be fresh:

$$\frac{\text{P believes fresh(X)}}{\text{P believes fresh(X,Y)}}$$

2.4.1.2 Analyzing a Protocol Using BAN Logic

One of the authentication protocols analyzed in [BAN89] is the Andrew Secure RPC Handshake. The protocol is intended to enable a client, A, to obtain from a server, B, a new session key, K'_{AB}, given that they already share a key, K_{AB}. The protocol is given in the standard cryptographic protocol pseudocode below.

$$\begin{aligned}
A \rightarrow B&: \quad A, \{N_A\}_{K_{AB}} \\
B \rightarrow A&: \quad \{(N_A + 1), N_B\}_{K_{AB}} \\
A \rightarrow B&: \quad \{N_B + 1\}_{K_{AB}}
\end{aligned}$$

$$B \rightarrow A: \quad \{K'_{AB}, N'_B\}_{K_{AB}}$$

Figure 2.10 : Pseudocode representation of the Andrew Secure RPC Handshake

N_A and N_B are nonces created by A and B respectively, and N'_B is an initial sequence number that will be incremented and used to number each message in the session that follows the authentication protocol. In the first message, A sends a nonce to B. In the second message, B returns A's nonce, incremented by one, along with a nonce of his own. The client, A, returns B's nonce incremented by one, and then B sends A the new session key and the beginning sequence number.

Burrows, Abadi, and Needham idealized the protocol into the following BAN logic representation.

$$A \rightarrow B: \quad \{N_A\}_{K_{AB}}$$
$$B \rightarrow A: \quad \{N_A, N_B\}_{K_{AB}}$$
$$A \rightarrow B: \quad \{N_B\}_{K_{AB}}$$
$$B \rightarrow A: \quad \left\{\left(A \overset{K'_{AB}}{\leftrightarrow} B\right), N'_B\right\}_{K_{AB}}$$

Figure 2.11 : The Andrew Secure RPC Handshake expressed in BAN Logic

Next the protocol's assumptions are listed:

A **believes** $A \overset{K_{AB}}{\leftrightarrow} B$

B **believes** $A \overset{K_{AB}}{\leftrightarrow} B$

A **believes** $(B$ **controls** $A \overset{K}{\leftrightarrow} B)$

B **believes** $A \overset{K'_{AB}}{\leftrightarrow} B$

A **believes fresh**(N_A)

B **believes fresh**(N_B)

B **believes fresh**(N'_B)

The first two assumptions indicate that A and B initially share a good key, K_{AB}. The third assumption states that A trusts B to generate good keys, and the fourth indicates that B has generated a new key that he considers to be good. The final three assumptions indicate that each agent believes the nonces it has generated to be fresh.

The analysis of this protocol using BAN logic looks like this. From the assumptions we know that:

B **believes** $A \overset{K'_{AB}}{\leftrightarrow} B$

From the four messages in the protocol we know that:

A **said** N_A

B **said** (N_A, N_B)

A **said** N_B

B **said** $\left(\left(A \overset{K'_{AB}}{\leftrightarrow} B\right), N'_B\right)$

Since each message was encrypted under K_{AB} we can use the first message-meaning rule to deduce:

B **believes** A **said** N_A

A **believes** B **said** (N_A, N_B)

B **believes** A **said** N_B

A **believes** B **said** $\left(\left(A \overset{K'_{AB}}{\leftrightarrow} B\right), N'_B\right)$

By using the nonce-verification rule on the three assumptions about nonce freshness and the preceding formulas we get:

$$B \textbf{ believes } A \textbf{ believes } N_B$$

$$A \textbf{ believes } B \textbf{ believes } (N_A, N_B)$$

However, we cannot get:

$$A \textbf{ believes } B \textbf{ believes } A \overset{K'_{AB}}{\leftrightarrow} B$$

So while it has been proved that *B* believes that he and *A* share a good new session key, we cannot prove that *A* believes the same thing. This is because there is nothing in the fourth message of the protocol that *A* believes to be fresh. Burrows, Abadi, and Needham's conclusion is that:

> the protocol suffers from the weakness that an intruder can replay an old message as the last message in the protocol, and convince *A* to use an old, possibly compromised session key. In other words, an intruder may find an old session key, and he may replay the fourth message of the handshake in which that key was established; he can then impersonate *B*. The problem can be fixed simply by adding the nonce N_A to the last message, and indeed a descendant of the Andrew file system has adopted this solution.

2.4.1.3 Nessett

Although BAN is currently the most popular logic for examining cryptographic protocols, a number of researchers have identified shortcomings in it. This has lead to the development of a host of extensions to BAN and a number of new logics all designed to address BAN's weaknesses. For example, in [NES90] Nessett presents a simple key-distribution protocol and then utilizes BAN logic to prove that the protocol is "secure" in so far as it can be proven that:

$$A \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

$$B \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

$$A \textbf{ believes } B \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

$$B \textbf{ believes } A \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

and the proof of these four properties was used to demonstrate the security of a key-distribution protocol in [BAN89]. Nessett's protocol is given below.

$$A \rightarrow B: \quad \{N_A, K_{AB}\}_{K^{-1}_A}$$
$$B \rightarrow A: \quad \{N_B\}_{K_{AB}}$$

Figure 2.12 : The Nessett Protocol

In the first step of the protocol agent *A* sends to *B* a nonce and a new session key, both of which she has supposedly generated and encrypted under her private key. In the second step, *B* generates a nonce, encrypts it under the new session key, and returns the result to *A*. The problem is that *B* is able to decrypt the first message and learn the session key because he knows *A*'s public key just like everybody else. So K_{AB} is not a good session key, since anybody who sees the first message can learn K_{AB}. Nessett's BAN logic analysis of the protocol is as follows. He idealizes the protocol as:

$$A \rightarrow B: \quad \left\{N_A, A \overset{K_{AB}}{\leftrightarrow} B\right\}_{K^{-1}_A}$$

$$B \rightarrow A: \quad \left\{A \overset{K_{AB}}{\leftrightarrow} B\right\}_{K_{AB}}$$

Next the assumptions are given:

$$B \textbf{ believes } \overset{K_A}{\rightarrow} A$$

$$A \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

$$A \textbf{ believes fresh}\left(A \overset{K_{AB}}{\leftrightarrow} B\right)$$

$$B \textbf{ believes fresh}(N_A)$$

$$B \textbf{ believes } A \textbf{ controls } A \overset{K_{AB}}{\leftrightarrow} B$$

From message one of the protocol Nessett gets:

$$B \textbf{ sees } \left\{ N_A, A \overset{K_{AB}}{\leftrightarrow} B \right\}_{K^{-1}_A}$$

Using the second message-meaning rule he deduces:

$$B \textbf{ believes } A \textbf{ said } N_A, A \overset{K_{AB}}{\leftrightarrow} B$$

And, using the nonce-verification rule on this formula:

$$B \textbf{ believes } A \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

Applying the jurisdiction rule to this statement and the fifth assumption he arrives at:

$$B \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

The second message gives:

$$A \textbf{ sees } \left\{ A \overset{K_{AB}}{\leftrightarrow} B \right\}_{K_{AB}}$$

Which, with the second message-meaning rule produces:

$$A \textbf{ believes } B \textbf{ said } A \overset{K_{AB}}{\leftrightarrow} B$$

Applying the nonce-verification rule to this formula and the third assumption yields:

$$A \textbf{ believes } B \textbf{ believes } A \overset{K_{AB}}{\leftrightarrow} B$$

Nessett concludes that “the example protocol is obviously insecure, [and therefore] the BAN logic is defective.”

In [BAN90], Burrows, Abadi, and Needham argue that Nessett’s example violates one of the assumptions of the logic, namely, that principals do not divulge their secret keys. In her summary of this debate in [MEA94], Meadows notes that “Nessett’s example makes

the point that this assumption is one that needs to be verified, since keys can be leaked not only by dishonest or incompetent principals, but as a result of the protocol itself." She concludes that "since BAN does not attempt to model knowledge, it can not be used to prove results about secrecy; it can be used only to reason about authentication."

2.4.1.4 Snekkenes

In [SNE91], Snekkenes demonstrates another limitation of the BAN approach, namely, that "flaws due solely to protocol step permutation [are] undetectable by the BAN logic." Snekkenes constructs an example protocol for a real-time monitoring system in which a master process, M, requires information from a number of sensors, S_1, S_2, ..., S_n, several times a day. The master process would like to be sure that the replies it receives are not only authentic, but that they are also timely, so that they correctly reflect the sensor's state. Snekkenes suggests the following protocol for this system. Let M send a numbered query, $Q_{i,j}$, to each sensor, S_i. A sensor responds by sending a message that contains a nonce, the master's query, and the sensor's answer. This reply by the sensor is signed with its private key to convince the master that it is authentic. It is further assumed that the master process keeps track of all nonces previously received from the sensors so that it can be sure that a sensor's reply contains a fresh nonce. In the final step of the protocol, the master acknowledges receipt of the answer by returning the nonce signed with the master's private key to the sensor. This protocol is shown in standard pseudocode below.

$$M \rightarrow S_i: \quad Q_{i,j}$$
$$S_i \rightarrow M: \quad \{R_{S_i,j}, Q_{i,j}, A_{i,j}\}_{K_{S_i}^{-1}}$$
$$M \rightarrow S_i: \quad \{R_{S_i,j}\}_{K_M^{-1}}$$

Figure 2.13 : The Snekkenes Protocol

Starting with the assumptions that:

M **believes fresh**$(R_{S_i,j})$

M **believes** $\xrightarrow{K_i} S_i$

M **believes** S_i **controls** $(Q_{i,j}, A_{i,j})$

Snekkenes performs a BAN logic analysis of the protocol and concludes that:

M **believes** $(Q_{i,j}, A_{i,j})$

According to Snekkenes, the master process should not believe the answer it gets from a sensor since the protocol is vulnerable to the following attack. An intruder could pose as the master, interact with a sensor several times, and store a series of the sensor's answers:

$$\{(R_{S_i,j}, Q_{i,j}, A_{i,j})\}_{K_{S_i}^{-1}}, j = 1, ..., n$$

Then, when the master queries the sensor, the intruder can choose one of these replies and replay it to the master. Snekkenes goes on to observe that simply changing the order of messages 2 and 3 so that M generates the nonce would render the protocol immune to the above attack. Snekkenes calls this the *step permutation problem* in BAN logic since the ordering of the steps in a protocol is not represented; instead the actions of a protocol are transformed into an unordered set of formulae in the logic. As a result, BAN logic can be used to find a proof of security of a protocol if *some* ordering of the steps is correct, even though the actual ordering of the steps in the protocol's specification might be incorrect. Meadows expresses concern about BAN logic's inability to "distinguish between the ... correct version of the protocol and the ... incorrect version."[MEA94]

2.4.1.5 Moser

Another limitation of BAN logic is highlighted by Moser's [MOS89] observation that:

> several well-developed formalisms exist for describing security policies and procedures, these formalisms are based on certainty and, when faithfully implemented, aim to guarantee security with a high degree of rigor. However, certainty requires complete

> information, which is seldom available and always expensive. Real systems cannot usually achieve certainty and, thus, are only approximately, and optimistically, modeled within those formalisms.

With this in mind, Moser develops a nonmonotonic logic of knowledge and belief that she uses to analyze a key-distribution protocol. Moser's logic includes both knowledge axioms and belief axioms, whereas most other logics are strictly epistemic (e.g. KPL [SYV90], CKT5 [BIE90]) or doxastic (BAN [BAN89]).

The main contribution of Moser's logic, though, is its nonmonotonicity. A logic, like BAN, is called *monotonic* if propositions cannot be refuted once they have been proven. In BAN this means that once an agent believes something he cannot subsequently disbelieve it. A *nonmonotonic* logic like Moser's allows for a proposition to be provable from a set of axioms, but the addition of a further axiom may render the proposition false. For example, in Moser's logic an agent could believe that a key is good, but upon learning that the key has been compromised, the agent might revise her beliefs and conclude that the key is not good.

Moser's nonmonotonicity is accomplished by introducing an **unless** operator that takes two arguments that are simple beliefs $B_i(p)$ or negations, conjunctions, or disjunctions of beliefs. The predicate $B_i(p)$ **unless** $B_i(q)$ denotes that agent *i* believes *p* unless he believes *q*. In the above proposition, $B_i(p)$ is presumed to be true unless it is refuted by other evidence. According to Moser, "reasoning in a nonmonotonic logic involves applying the axioms in their entirety to determine if any of them refute the conjectured lemma or theorem." We summarize Moser's nonmonotonic logic by noting that it allows:

- Belief in a proposition in the absence of other information
- Refutation of that presumption if evidence indicates that it is unjustified
- A mechanism to model uncertainty about a proposition in the presence of contradictory information.

Nonmonotonic logic also introduces added complexity during reasoning since it allows for more than one valid solution to exist when there are cycles among various **unless** clauses. For example, $(B_i(p)$ **unless** $B_i(q))$ **and** $(B_i(q)$ **unless** $B_i(p))$ has either of two solutions: $(B_i(p)$ **and** $\neg B_i(q))$, or $(\neg B_i(q)$ **and** $B_i(p))$. Moser's explanation of this property is that "rational agents may interpret the same evidence but reach different conclusions that are equally valid." For this reason, Moser acknowledges that reasoning in a modal logic, like BAN, is more difficult than reasoning in propositional calculus, and that reasoning in a nonmonotonic logic is harder still, particularly since "natural deduction methods no longer apply."

2.4.2 Other Alternatives and Extensions to BAN Logic

As we mentioned at the beginning of this section, research on logics for reasoning about cryptographic protocols has been a very active area since the introduction of BAN. Unfortunately, many of the schemes that have been proposed as alternatives or extensions to BAN logic have been shown to have limitations of their own, and difficulty in understanding and applying these logics has undermined some of these efforts to replace BAN with more expressive logics. BAN does have a number of limitations as we have seen in earlier sections, but the fact that it has only ten inference rules and an easily-understood model of reasoning makes the logic easy to use.

Simplicity, combined with its usefulness, is probably the main reason that BAN is still so widely used despite its well-publicized limitations. In contrast, many of the logics which have been designed to address some of the issues that BAN logic ignores are extremely complex. For example, Gong, Needham, and Yahalom's extended version of BAN logic, GNY logic, contains more than fifty inference rules, many of which are complicated themselves. One of the many additions this logic makes to BAN is rules for reasoning about message recognizability. These rules enable one to reason about a principal's ability to recognize that a bit string is a meaningful message, for example.

Perhaps due to its added complexity, GNY logic has not yet achieved nearly the same widespread use as BAN logic. In the remainder of this section we will briefly discuss some of the other alternatives and extensions to BAN logic that have been suggested.

2.4.2.1 Bieber

In [BIE90], Bieber presents a logic called CKT5 to reason about communications in a hostile environment. CKT5 is a combination of epistemic and temporal logic in which the knowledge operator is indexed by agents and time. For example, in CKT5 the statement $K_{A,t}P$ means that at time t agent A knows P. CKT5 also defines two new modal operators that model how an agent's knowledge can change as a result of communication with other agents. These two operators are $S_{A,t}P$ and $R_{A,t}P$, which represent, respectively, the sending and receiving of a message by A at time t. The CKT5 logic is the basis of a formal method for describing and analyzing cryptographic protocols that is demonstrated in [CAR93].

2.4.2.2 Syverson

Another epistemic logic is Syverson's KPL [SYV90]. KPL is a propositional logic of knowledge that has a possible world semantics. A *possible world semantics* defines a set of different ways the world may be, and an accessibility relation on this set for each agent. According to Syverson, "if one world is *accessible* from another for a given individual, then that individual cannot distinguish the two worlds in her state of knowledge." Syverson uses the following example to illustrate this principle:

> if Melissa is in a world where it's raining, but there is another world accessible from it where it's not raining, then Melissa doesn't know that it's raining. (She could be in a room with no windows.)

The advantage of such a semantics is that it gives an easily understandable way of reasoning about an agent's knowledge in the real world. To do this we simply need to prove that a

given proposition about an agent engaged in a protocol is true in all worlds accessible from the real world.

2.4.2.3 Gaarder and Snekkenes

In [GS91], Gaarder and Snekkenes extend BAN logic to be able to reason about public as well as private key cryptosystems. They accomplish this by adding the following statements to the logic:

PK(K,U) : The entity *U* has a good public key *K*.

$\Pi(U)$: The entity *U* has a good private key known only to *U*.

$\sigma(X,U)$: The formula *X* is signed with the private key belonging to *U*.

The following two inference rules are added to reason about statements of this form:

- The *signed-message rule* states that in order to believe that U_j once said *X*, it is sufficient to believe that we have U_j's public key, that U_j's secret key is good, and that a message containing *X* signed with U_j's private key must have been seen:
$$\frac{U_i \text{ believes } PK(p_j,U_j), U_i \text{ believes } \Pi(U_j), U_i \text{ sees } \sigma(X,U_j)}{U_i \text{ believes } U_j \text{ once said } X}$$
- The *signed-message content rule* says that the contents of a signed message can always be made visible:
$$\frac{U_i \text{ sees } \sigma(X,U_j)}{U_i \text{ sees } X}$$

Gaarder and Snekkenes also introduce the notion of "duration" which allows a more fine-grained treatment of time than BAN's simple division of time into the "current" and "past" runs of the protocol. The statements for time are:

$(\Theta(t_1,t_2),X)$: *X* holds in the interval (t_1,t_2). The creator which uttered the time-stamped message X, claims that X is, or was, good in the time interval between (t_1,t_2).

$\Delta(t_1,t_2)$: (t_1,t_2) denotes a good interval. The local unique real time clock shows a time in the interval between (t_1,t_2).

The rule for reasoning about durations is:

- The *duration rule* states that when uttering a duration-stamped message, we commit ourselves to

believe the message for the interval specified by the duration-stamp:

$$\frac{P \text{ believes } Q \text{ believes } \Delta(t_1,t_2), P \text{ believes } Q \text{ once said } (\Theta(t_1,t_2),X)}{P \text{ believes } Q \text{ believes } X}$$

With these extensions, Gaarder and Snekkenes utilized the logic to demonstrate a flaw in a CCITT X.509 authentication protocol.

2.4.2.4 Kailar and Gligor

In [KG91], Kailar and Gligor develop a logic that models the evolution of beliefs in an authentication protocol. Like BAN logic, theirs is also a logic of belief and it is developed at about the same level of abstraction. In addition to borrowing much of BAN's notation and inference rules, Kailar and Gligor introduce additional constructs that enable principals to reason about other principal's knowledge set of beliefs. According to Kailar and Gligor, "an agent, *X*, is said to be a member of the knowledge set, *KS*, of message content *a* at message instance M_i if X can recognize the message content *a* at and after the instant when message M_i is received." They express this formally as $X \in KS(a,M_i)$.

After presenting their logic, Kailar and Gligor analyze a number of authentication protocols with BAN logic and then with their own formalism. The most important difference between the two is the ability of Kailar and Gligor's scheme to analyze protocols that do not make use of key jurisdiction properties. For example, in the BAN analysis of a Multi-Party Session protocol they conclude that agent *Y* does not believe that the communication key sent by agent *X* is good because *Y* does not believe that *X* controls the session key. However, by using *Y*'s belief about *X*'s knowledge set beliefs, the logic of Kailar and Gligor can show that *Y* can believe that the session key sent by *X* is good.

2.4.2.5 Campbell, Safavi-Naini, and Pleasants

Campbell et al. [CSP92] extend BAN logic by adding probabilities to the sentences and rules of the logic allowing them to "quantify the beliefs of principals and represent the insecurities and uncertainties of a real life situation." For example, one could assign the

probability p_1 to the assumption that A **believes** (B **controls** $A \overset{K}{\leftrightarrow} B$) in the BAN analysis of the Andrew Secure RPC Handshake given in Section 2.4.1.2. Likewise, the probability p_2 could be assigned to BAN logic's jurisdiction rule. According to Campbell et al., "this gives a useful means of quantifying the trust on the conclusions derived from formal proofs in the logic in terms of the probabilities assigned to the beliefs of the principals and the inference rules they use."

2.4.2.6 Mao and Boyd

Mao and Boyd's goal in [MB93] is to suggest a new logic which adopts the basic notational framework of BAN logic, but takes a more formal approach. Of particular concern to Mao and Boyd are the various weaknesses of BAN logic that have been noted previously. Most importantly, Mao and Boyd propose a more formal alternative to BAN's idealization process which they consider "fundamentally flawed." Although they introduce a small number of new constructs to their logic, Mao and Boyd do not regard their new technique as being more complex than idealization in BAN logic, and, in fact, intend it to be more formal and straightforward.

In explaining their alternative to BAN's idealization step, Mao and Boyd first give definitions for such things as atomic messages, challenges, replied challenges, responses, and nonsense. Next, they propose rules for what they call protocol message idealization. According to Mao and Boyd, these rules are "formally feasible, which means that, with limited human intervention, they form a guideline to correctly comprehend the authentication semantics of a security protocol." They go on to demonstrate the use of their protocol message idealization and their new logic by examining several standard protocols.

2.4.2.7 Syverson and van Oorschot

In [SvO94], Syverson and van Oorschot present a logic that represents the unification of the four members of the BAN family of logics presented in [BAN89],

[GNY90], [AT91], and [vO93]. According to Syverson and van Oorschot, the logic "captures all of the desirable features of its predecessors and more; nonetheless, it accomplishes this with no more axioms or rules than the simplest of its predecessors." Syverson and van Oorschot borrow heavily from the model of computation and semantics of the AT logic in [AT91], and they include many of the extensions found in the GNY and VO logics. Rather than simply tacking together the notation and rules from all of those logics, Syverson and van Oorschot develop an integrated approach that is designed to make their logic "as simple to use or simpler than any of those [logics] from which it was derived; yet ... more expressive than any of them."

2.4.2.8 Rubin and Honeyman

Rubin and Honeyman present a method for specifying and analyzing nonmonotonic cryptographic protocols in [RH94]. They point out that all of the other logics developed to date reason monotonically about knowledge - once something is known, it is always known. This means that there are valid formulas that cannot be derived like "*P* **knew** *X*, but *P* no longer **knows** *X*," for example. Rubin and Honeyman differentiate their logic from Moser's by pointing out that only beliefs can be reasoned about nonmonotonically in Moser's system whereas Rubin and Honeyman deal with nonmonotonicity of knowledge.

Another interesting aspect of Rubin and Honeyman's approach is that they do not require protocol idealization but rather specify protocols at a level that is close to the actual implementation. They demonstrate their technique by uncovering a known flaw in a protocol by Needham and Schroeder [NS78], and they also apply their method to their own *khat* protocol. The analysis reveals what they call a "serious, previously undiscovered flaw in our nonmonotonic protocol for long-running jobs."

2.4.2.9 Kessler and Wedel

Kessler and Wedel's AUTOLOG [KW94] is a modified version of BAN logic which is implemented in PROLOG. Kessler and Wedel consider the following modifications to the constructs, inference rules, and idealization procedure of BAN logic "useful for most protocols and [they] do not complicate the logic":

- introducing the predicate **recognize**
- replacing the predicate "*A* **believes** that *B* **believes** that ..." by the predicate "*A* **believes** that *B* **has recently said** that ..."
- enlarging the meaning of the **sees** operator
- not omitting the cleartext messages in the idealization of a protocol
- simulating an eavesdropper, *Z*
- reducing the ambiguous idealization step by introducing a special *key rule*

Kessler and Wedel demonstrate how their AUTOLOG tool can analyze a simple challenge-response protocol from the Draft International Standard [ISO93] by using their method to show a failure in this protocol.

2.5 Summary

In this chapter we have reviewed many cryptographic protocol analysis techniques. Informal analysis has uncovered a number of flaws in protocols, but this approach depends entirely on the insight of the person performing the analysis. Flaws have tended to go undetected for years before being discovered by this method, and relatively little is learned about a protocol when informal analysis fails to discover a vulnerability.

In [MEA94], Meadows suggests why cryptographic protocols are so "well suited for the application of formal methods." According to Meadows, they are "well-contained enough so that modeling and analysis should be tractable; on the other hand, they are complex enough and the flaws are counterintuitive enough so that an informal analysis may

be too prone to error to be reliable." State-based and logical methods for analyzing cryptographic protocols are fairly new, but some important results have already been obtained using them. While formal methods are not in wide use among protocol designers yet, we expect this to change in the near future as the techniques are improved and as protocols are developed for more critical applications.

Chapter 3

The Cryptographic Protocol Analysis Language

In this chapter we present a language for specifying and reasoning about cryptographic protocols that was developed in [YAS96]. This language has several important advantages over the pseudocode notation which was used in Chapter 2. Most importantly, Yasinsac's Cryptographic Protocol Analysis Language (CPAL) defines a semantics that can be used to reason formally about a protocol. After reviewing pseudocode notation and presenting CPAL, we offer a brief comparison of the two and conclude the chapter by discussing Yasinsac's analysis technique based on the formal semantics of CPAL.

3.1 Review of the Standard Pseudocode Notation

Although a number of researchers ([CAR94], [YAS96], [MIL96]) have proposed alternative methods for specifying cryptographic protocols, most protocols are still specified almost exactly as they were in Needham and Schroeder's 1978 paper. The appeal of what has come to be called *pseudocode* or *Standard Notation* is that it is simple,

compact, and well understood by the research community. As we saw in Chapter 2, the pseudocode representation of a protocol usually includes a list of messages sent during a run of the protocol with the messages listed in the order in which they are to be sent. The name of the principal sending the message and the name of the intended recipient are given explicitly for each message as in the following pseudocode for agent A sending a message intended for agent B:

$$A \rightarrow B: \qquad message$$

The values that comprise a message are listed, separated by commas. For example, if agent A sends B the nonce, N_A, the pseudocode representation would be:

$$A \rightarrow B: \qquad N_A$$

Likewise, if B sends A his own name and the encryption of N_A under K_{AB}:

$$B \rightarrow A: \qquad B, \{N_A\}_{K_{AB}}$$

While this notation allows straight-forward and compact representations of cryptographic protocols, it does suffer from some serious limitations. Most importantly, the semantics associated with this language are informal and incomplete, so protocols expressed in this manner will have to be translated into some other representation or annotated to facilitate formal analysis.

It is also clear from the examples in Chapter 2 that pseudocode does not explicitly represent all of the actions by the agents during a run of the protocol. Actions such as receiving messages, generating nonces, decrypting values, and checking timestamps are not represented in a pseudocode specification but are usually described separately using English prose. For example, consider the following pseudocode specification of a simple two-message protocol:

$A \rightarrow B$: $\{N_A, B\}_{K_{AB}}$

$B \rightarrow A$: N_A

The pseudocode representation of this protocol does not explicitly state that *B* decrypts the message he receives from *A* in the first step, but we can assume he must in order to learn the value of the nonce he returns in step two. This implies that both *A* and *B* know K_{AB}, although that assumption is not stated explicitly in the pseudocode specification. Furthermore, the pseudocode does not tell us whether *B* checks to see if the name included in message one matches his, or if *A* checks the nonce returned in the second message to make sure it matches the one she sent in message one. Whether or not these checks are performed determines what the given protocol accomplishes. However, we cannot decide if they should be performed from the pseudocode since it does not represent the intended goals of the protocol. In the next section we present a protocol specification language that seeks to remedy all of the limitations of pseudocode notation given in this section while remaining simple, compact, and easy to understand.

3.2 CPAL

Yasinsac's CPAL is a language that was designed specifically for cryptographic protocol specification and formal analysis. We begin our presentation of CPAL by describing the model of the environment in which cryptographic protocols operate according to Yasinsac.

3.2.1 The CPAL Environment

CPAL's model of the environment in which cryptographic protocols operate is very similar to those suggested by other researchers including Abadi and Tuttle in [AT91]. Each principal possesses a private address space which cannot be read or written by any other principal. An agent can use her private address space to store values and to perform computations on those values. Each principal also has an input queue which can be written to by anyone. Agents can communicate with one another by performing send and receive

operations. A *send* operation places a message into an agent's input queue, and a *receive* operation consists of a principal removing a message from the head of her input queue and storing it somewhere in her private address space. If the agent's input queue is empty when she performs a receive, it is assumed that she blocks waiting for a message to arrive. For reasons that will be explained later, it is assumed that the blocking will time-out after some fixed period and the agent will proceed as if a null message had arrived. Given this model of the environment, we are now ready to present an overview of the syntax of CPAL.

3.2.2 Syntax

For those interested in the complete definition of CPAL's syntax, it is given in Backus-Naur form in Appendix B of [YAS96]. CPAL protocol specifications are similar to those expressed in pseudocode in so far as the protocol is represented as an ordered sequence of actions by the principals involved in the protocol. In CPAL, each action is preceded by the name of the agent performing it, and the semicolon is used as an action separator. A generic CPAL protocol specification might look like:

$Agent_i$: *action*;
$Agent_i$: *action*;
$Agent_j$: *action*;
$Agent_i$: *action*;
$Agent_j$: *action*

Unlike pseudocode, there are more actions in CPAL than simply sending messages. In the following subsections we concentrate on the syntax of the various actions in CPAL while hinting at the meanings of some of these constructs. In the next section we give a complete and formal semantics to the CPAL language.

3.2.2.1 Assignment

One of the actions in CPAL is an *assignment* which has the following syntax:

$Agent_i$: *variable* **:=** *value*

As in most procedural programming languages a *variable* is merely a symbolic name for a storage location. In an assignment statement, all variables must refer to locations in the private address space of the agent performing the assignment. For example, in the statement:

A: x **:=** *y*

the variables *x* and *y* refer to locations in agent *A*'s address space. In CPAL, if we need to talk about variables from a global perspective, we always prefix the variable by the name of the principal in whose address space it resides. For example, *A.x* and *B.x* are the CPAL representations of the distinct variables, *x* in *A*'s address space, and *x* in *B*'s address space, respectively.

CPAL is an untyped language where any variable is allowed to store any type of value. Simple *values* in CPAL include integers, character strings, bit strings, booleans, and variables. More complicated *values* can be formed by applying some function to a set of values or by concatenating values together into a list. Angle brackets are used in CPAL to represent concatenated values as in:

$A\colon x := \langle value_1, value_2, \ldots, value_n \rangle$

A *dot operator* is used to extract a particular element from a catenated list:

$value_3 \equiv \langle value_1, value_2, \ldots, value_n \rangle.3$

3.2.2.2 Conditionals

CPAL also allows principals to execute a conditional action or sequence of actions. There are several variations on the conditional statement including:

A: ***if*** *condition* ***then*** *action*

and

A: ***if*** *condition* ***then*** *{action*$_1$***;*** *action*$_2$***;*** *..., action*$_n$*}*

and

A: ***if*** *condition* ***then*** *action*$_1$ ***else*** *action*$_2$

Conditions in CPAL include comparisons between values and evaluation of boolean functions.

3.2.2.3 The Receive Statement

A principal *receives* a message by removing the head element of her input queue and storing it somewhere in her private address space. A receive statement in CPAL looks like this:

A: $\leftarrow (variable)$

In this example, the message is taken off *A*'s input queue and stored in the location associated with the name *variable* in her address space.

3.2.2.4 Send Statements

There are two different types of *send* statements in CPAL which place a message on a principal's input queue. They are the *secure send*, which uses the "$\Rightarrow$" operator, and the *insecure send*, which uses the "$\rightarrow$" operator. The syntax for the secure statement is:

A: $\Rightarrow B(value)$

and the syntax for the insecure send statement is:

A: $\rightarrow B(value)$

The first represents *A* securely placing a message on *B*'s input queue. That is, *A* transmits her message to *B* in such a way that the message cannot be delayed, redirected, destroyed,

modified, or observed by any other agent. A secure send operation would obviously be quite expensive to implement. The insecure send statement corresponds to *A* sending a message to *B* in an insecure manner. The message might eventually appear on *B*'s input queue, but may also be delayed, redirected, destroyed, modified, or observed by another agent while in transit.

3.2.2.5 The Reject Statement

The *reject* statement halts a principal's participation in a protocol run. Any subsequent actions that the principal would have performed in that protocol run will not be executed. The syntax for the reject statement is:

A: ***reject***

3.2.2.6 Assumptions

Though not technically an action, CPAL allows principals to explicitly state what *assumptions* they make during a run of the protocol. CPAL expresses assumptions as follows:

A: ***assume****(condition)*

3.2.2.7 Assertions

CPAL also includes an *assert* statement:

A: ***assert****(condition)*

which can be used to explicitly state goals at any point in the protocol. Note that while pseudocode specifications typically use prose to describe many of the actions as well as all of the assumptions and goals of a protocol, CPAL represents all of these things explicitly as part of the protocol's specification. Specifically, CPAL states where and when each goal should be satisfied in the protocol.

3.2.2.8 Summary of CPAL Constructs

In the table below, we give a brief description of each of the CPAL constructs just discussed. The table also contains an example for each construct to illustrate its syntax.

CPAL Construct	Example	Description
Assert Statement	A: assert(A.kab == B.kab)	Specify a goal of the protocol.
Assume Statement	B: assume(Inverse(B.k, B.k')	Specify an assumption of the protocol.
Insecure Send Statement	A: -> B (x)	Send a message to an agent. While in transit the message can be seen, modified, redirected, delayed, or destroyed by dishonest agents.
Secure Send Statement	A: => B (y)	Send a message to an agent. While in transit the message cannot be seen, modified, redirected, delayed, or destroyed by anyone.
Receive Statement	B: <- (z)	Receive a message that has been securely or insecurely sent and store it in the recipient's private address space. If no message has been sent, block and wait for a message to arrive.
Reject Statement	A: reject	Halt a principal's participation in a protocol run
Assignment Statement	B: a := b	Perform an assignment from one location in an agent's private address space to another address in that agent's private address space. A function may be applied to the value on the right-hand side of the assignment.
Conditional Statement	A: if (c<d) then m := c else m := d	Conditionally execute a statement.
Statement Concatenation	A: => B (t); B: <- (u)	Execute two constructs serially.

Table 3.1: CPAL Constructs

3.2.3 Semantics

In the preceding section we sketched an operational model for CPAL which included the language's syntax and some informal semantics. In this section we present Yasinsac's formal semantics of CPAL which are based on work by both Hoare and Dijkstra. Yasinsac has defined a formal method which uses these semantics to analyze a cryptographic protocol specified in CPAL.

3.2.3.1 Hoare's Precondition/Postcondition Reasoning

In [HOA69], Hoare suggested giving a functional semantics for a program segment, *S*, using a boolean expression called a *Hoare triple*: $Q \{S\} R$, where *Q* and *R* are boolean expressions called the precondition and postcondition, respectively. We say that $Q \{S\} R$ holds if program segment *S*, beginning with *Q* satisfied, is guaranteed to establish *R*, <u>if *S* terminates</u>.

For example, in order for the postcondition "$y = 2$" to be true after the statement "$y := \sqrt{x}$" is executed, the precondition "$x = 4$" must hold before the assignment. The preceding semantics for the statement "$y := \sqrt{x}$" can be represented by the Hoare triple:

$$x = 4 \{y := \sqrt{x}\} y = 2$$

The above Hoare triple holds under the normal definitions of the "$\sqrt{\ }$" and ":=" operators. Now let us consider the statement "$y := x^2$". Using standard definitions for the operators, the following Hoare triple holds:

$$x = 2 \{y := x^2\} y = 4$$

However, note that the following Hoare triples hold as well:

$$x = -2 \{y := x^2\} y = 4$$

$$(x = -2) \vee (x = 2) \ \{y := x^2\} y = 4$$

The precondition $(x = -2) \vee (x = 2)$ is said to be *more general* or *weaker* than the other two preconditions because it is logically implied by each of them. In the next subsection we describe Dijkstra's notion of weakest precondition, which is an extension to Hoare's precondition/postcondition reasoning that Yasinsac uses to define the formal semantics of CPAL.

3.2.3.2 Dijkstra's Weakest Precondition Reasoning

In [DIJ76], Dijkstra defines a weakest precondition, written *wp(S,R)*, as the weakest solution to the equation $Q : Q \{S\} R$. Recall that a weakest solution to a boolean equation is a solution that is implied by all solutions, so in the case of $Q : Q \{S\} R$, for any solution Q, *Q implies wp(S,R)* holds. According to Dijkstra, the weakest precondition for a program statement can also be viewed as a "predicate transformer" since it is "a rule telling us how to derive for any postcondition R the corresponding weakest precondition for the initial state such that activation will lead to a properly terminating activity that leaves the system in a final state satisfying R."

3.2.3.3 Yasinsac's Weakest Preconditions for CPAL Constructs

The table below gives Yasinsac's definition of the weakest preconditions for each CPAL construct.

CPAL Construct	Weakest Precondition Predicate
Assert Statement	wp("**assert**(X)", R) = $X \wedge R$
Assume Statement	wp("**assume**(X)", R) = $X \supset R$
Assignment Statement	wp("Y := Z", R) = $R\|_Z^Y$
Insecure Send Statement	Not defined. (See below)
Secure Send Statement	wp(" $\Rightarrow A(M)$ ", R) = $R\|_{A\text{'s input queue}}^{M}$
Receive Statement	wp(" $\leftarrow (M)$ ", R) = $R\|_{M}^{\text{receiving agent's input queue}}$
Conditional Statement	wp("**if** (C) **then** S1 **else** S2", R) = $(C \supset wp(S1, R)) \wedge (\neg C \supset wp(S2, R))$
Statement Concatenation	wp(S1;S2, R) = wp(S1, wp(S2, R))

Table 3.2: Weakest Preconditions for CPAL Constructs

We explain these weakest precondition definitions in more detail below.

The Assign Statement. Assignments in CPAL are always made within a single principal's address space. A value, which may be the contents of another variable or may be a more complex expression, is copied to the location referenced by the destination identifier. Yasinsac's semantics of the assign statement given in the table above state that for a predicate, R, to be true after the statement $x := y$ is executed, the predicate R with each instance of the variable x textually replaced by y must be true before the statement is executed. For example, if the postcondition is $(x = 3)$ then the weakest precondition of the statement $x := y$ is $(y = 3)$. This would be represented in Dijkstra's notation as:

$(y = 3) = wp(\text{“}x := y\text{”}, (x = 3))$.

Conditionals. In a conditional statement of the form "**if** *(C)* **then** *S1* **else** *S2*" only one of the two substatments, *S1* and *S2*, is executed. Which of the two substatements is executed depends on the evaluation of the condition, C. If the condition is true *S1* will be executed and if the condition is false *S2* will be executed. Yasinsac defines the semantics of the conditional statement by noting that if R is the desired postcondition for the whole statement, then the truth of the condition, C, must imply the weakest precondition of *S1* with R as the postcondition. Likewise, the falsity of C must imply $wp(S2, R)$. The conjunction of these two requirements is given as the weakest precondition predicate for conditional statements in Table 3.2 above.

The Receive Statement. A receive statement is similar to an assign statement except that the source of the assignment is implicit, namely, the input queue of the agent performing the receive operation. The value is removed from the head of the input queue[1] and stored at the specified location. So the receive statement "A: $\leftarrow (X)$" is the same as the assignment statement, "*A: X := head of A's input queue*", with the first element in *A*'s input

1. If the agent's input queue is empty when she performs a receive, it is assumed that she blocks for some fixed amount of time waiting for a message to arrive. If no message arrives in that time, it is assumed that she unblocks and proceeds as if a null message had arrived.

queue being deleted after the assignment. The weakest precondition definition of the receive statement in Table 3.2 reflects this description.

The Secure Send Statement. A secure send statement places a message on the input queue of the intended recipient. The message cannot be delayed, redirected, destroyed, modified, or observed while in transit. Yasinsac does not state how secure sends are to be carried out, but acknowledges that such an operation would be very expensive and cannot be the normal mode of communication in cryptographic protocols. As with the receive statement, the semantics of a secure send statement are similar to that of an assignment. In this case, a value is copied from one agent's address space to the input queue of another principal (see Table 3.2).

The Insecure Send Statement. In an insecure send, a principal sends out a message that is intended for another agent, but the message can be delayed, redirected, destroyed, modified, or observed by the all-powerful intruder. To make these abilities of the intruder explicit, tools that use CPAL always preprocess protocol specifications by changing all insecure sends into secure sends to the intruder. Once the intruder has received a message, he could choose to send it on unmodified to its intended destination, or he could choose any of the actions mentioned above. Since all insecure sends are removed from protocol specifications and replaced by secure sends to the intruder, no semantic definition is given for the insecure send statement.

The Assert Statement. An assert statement in CPAL does not specify any action by a principal but rather provides a mechanism to represent an agent's goals in the protocol. The assert statement:

A: **assert**(P)

declares that the proposition, P, must be true at this point in the protocol if the protocol is to work correctly. Since no action is taken by the assert statement, its postcondition, R, must

be satisfied before the assert statement, in addition to the proposition, P, represented in the assert statement. This is expressed in Yasinsac's definition of the semantics of the assert statement in Table 3.2: wp("A: **assert**(P)", R) $= P \wedge R$.

The Assume Statement. The assume statement is similar to the assert statement in that it does not specify any actions by the principals involved in the protocol. An assume statement explicitly states an assumption that is being made by the protocol designer, and these assumptions can be used later in the analysis of the protocol. Since the assumption, A, can be used to prove the theorem, R, Yasinsac defines the weakest precondition for the assume statement to be: wp("X: **assume**(A)", R) $= A \supset R$ as seen in Table 3.2.

Statement Concatenation. CPAL uses the semicolon to concatenate two statements together. This means that if *S1* and *S2* are CPAL statements, then *S1;S2* denotes the execution of *S1* followed by the execution of *S2*. More formally, Yasinsac defines the semantics of the semicolon as follows: wp("*S1;S2*", R) = wp(*S1*, wp(*S2*, R)). This definition, along with the blocking semantics of the send and receive operations, allows us to designate a strict sequencing of the actions in a protocol. This, in turn, allows us to apply Dijkstra's weakest precondition reasoning to define the formal semantics for protocols specified in CPAL. As we saw in Chapter 2, the lack of such sequencing is a known weakness of many of the logics used for cryptographic protocol analysis.

3.2.4 Analyzing Protocols Using CPAL

Defining a weakest precondition for each CPAL statement not only gives a formal semantics for every protocol specified in CPAL, but also allows us to analyze the protocol based on these semantics. In Appendix A we demonstrate Yasinsac's cryptographic protocol analysis technique which utilizes CPAL and its weakest precondition definitions given above.

The basic steps in Yasinsac's approach are:

- to manually translate the protocol into CPAL
- to automatically compute the weakest precondition for the protocol
- to automatically simplify the weakest precondition for the protocol
- to manually attempt to prove the simplified weakest precondition for the protocol

If the final step succeeds, then Yasinsac has shown that a particular trace of the protocol satisfies the protocol's stated goals. This does not imply that all valid traces of the protocol will satisfy the goals, and it is clearly impossible to perform this type of analysis on every possible trace of the protocol since there are infinitely many of them. However, this approach is still valuable because it provides us with a formal method for determining whether or not a trace of the protocol meets the protocol's goals. In later chapters we propose extensions to Yasinsac's approach that result in a more robust protocol analysis technique.

3.3 Summary

In this chapter we have reviewed the standard pseudocode notation used to express cryptographic protocols in the literature. We noted some important limitations of pseudocode including:

- its lack of a formal semantics, and
- its imprecision about the assumptions, actions, and goals that comprise a protocol

We then described Yasinsac's Cryptographic Protocol Analysis Language (CPAL) which remedies these two weaknesses. We presented the syntax and semantics of CPAL and gave a brief overview of how Yasinsac uses CPAL to perform an analysis of a protocol trace. A detailed example of Yasinsac's technique is given in Appendix A.

In the next chapter we present a brief review of the field of automatic program synthesis. We do this to prepare the reader for the discussion of our methodology (in Chapter 5) which builds on Yasinsac's work by having the user specify a protocol, *P*, in CPAL and a statement of failure, *F*, for the protocol. Using the statement of failure as a postcondition for the protocol, we can then use Yasinsac's technique to compute the weakest preconditions for $P = S_1;S_2;S_3; \ldots ; S_N$.

$wp(S_1,\ldots)$

$S_1;$

.

.

.

$wp(S_{N-2},wp(S_{N-1},wp(S_N,F)))$

S_{N-2}

$wp(S_{N-1},wp(S_N,F))$

$S_{N-1};$

$wp(S_N,F)$

$S_N;$

F

However, proof of any of these weakest preconditions actually demonstrates that the protocol is insecure, since *F*, the postcondition, is a statement of failure. We then use an automatic theorem prover to attempt to find a constructive proof of one of these weakest preconditions and, if it succeeds, use the proof to modify the protocol so that the failure condition is achieved. This last step is closely related to automatic program synthesis, as we demonstrate in the next chapter.

Chapter 4

Automatic Program Synthesis

In this chapter we give a broad overview of the field of automatic program synthesis, which seeks to map a problem specification (usually given in some very high-level, non-procedural language) to an implementation (usually a program in some programming language). We present this information because our cryptographic protocol evaluation system employs a simple automatic programming strategy that generates attacks from specifications derived from the formal semantics of the protocols.

4.1 Background

Automatic programming has been a goal of computer science and artificial intelligence almost since their inception. Writing programs for computers is a time-consuming and error-prone task for humans, and we would much prefer that people be able to explain what they want the computer to do and have the computer determine how to do it. This is a difficult problem and researchers have had to settle for gradually increasing the level at which the user specifies the problem (from machine language to assembly language to high-level programming languages to very high-level programming languages) while

simultaneously decreasing the user's responsibility for deciding exactly how the problem will be solved.

There are currently many researchers working on automatic program synthesis, some of whose projects have been active for more than 20 years. Most of the approaches being pursued can be characterized as either procedural, deductive, transformational, or inspective. In the next four sections we give a brief description of each of these categories and point out some of the important research projects in each area. In later sections of this chapter we discuss which of these approaches to automatic programming we have taken for our work, and why we chose it.

4.2 Procedural Methods

Procedural methods were among the earliest and most successful automatic program synthesis techniques. The procedural approach involves writing a special-purpose program that takes a specification as input and generates the proper implementation as output. The most common example of the procedural approach is the compiler[1]. Compilers generally accept a specification in some "high-level" language and produce an implementation in some lower-level language. Often, the compiler will perform some "programming" (e.g. allocation of registers for variables) and some optimization (e.g. procedure inlining, loop unrolling, common subexpression elimination).

The main advantage of the procedural approach is that the early stages are usually completed quickly and without too much difficulty. Producing a working system with a few basic features for a small subset of the problem domain may be accomplished in a matter of days or weeks. After that, the code can always be modified to try to add more features to the system and to expand the problem domain. Unfortunately, the main drawback of proce-

1. In fact, compilers were considered to be "automatic programming" systems when first introduced, although they are no longer thought of in those terms.

dural systems is that it becomes increasingly difficult to modify the code and add new features. While there are many examples of useful, domain-specific procedural automatic programming systems, the difficulties with modifying and extending such systems makes it unlikely that this approach could yield a general-purpose, full-featured automatic program synthesis method.

4.3 Deductive Methods

"The problem of synthesizing a program satisfying a given specification is formally equivalent to finding a constructive proof of the specification's satisfiability." This observation by Rich and Waters in [RW88] sums up the deductive approach to automatic programming. Deductive techniques are appealing because logic provides a general and powerful reasoning framework which lends itself well to automation. For this reason, deduction has become a technique that has been applied time and again in the field of artificial intelligence.

4.3.1 Overview

Finding a deductive proof of a theorem basically consists of starting with an initial set of facts (or *axioms*) and applying the given *inference rules* to derive new facts until the *goal fact* has been deduced. At each step in this process there may be many different inference rules that could be applied to derive new facts so the search for an inference path from the initial state to the goal state will usually be exponential in nature. Deductive program synthesis systems normally deal with this explosion of the search space by either working only on "small" problems or by asking a human user to direct the search.

Another limitation of the deductive approach is that specifications must be expressed using the notation of the underlying logic, and coming up with a logical specification that is complete, concise, and correct is difficult for most problems of interest to the

automatic programming community. Indeed, it is not clear that producing such a specification would in general be any easier or less error-prone than producing a procedural specification. Furthermore, we note that the deductive approach contains no bias towards finding the proof corresponding to the most efficient program, or even a reasonably efficient program, that solves a given problem. While this requirement is not normally stated explicitly in the specification for an automatic programming system, we might expect any useful automatic programming system to produce programs that are at least somewhat efficient.

4.3.2 Example of the Deductive Approach

The following example from [BIER76] illustrates how the deductive approach can be utilized to synthesize a program to compute a function, $f(x)$.

Assume that $P(x)$ is true if x is a valid input for the program and false otherwise. Likewise, $R(x,z)$ is true if $z = f(x)$ and false otherwise.

A program that computes $f(x)$ can be generated by proving the theorem:

$$\forall x \; P(x) \;\supset\; \exists z \; R(x,z)$$

This theorem states that for all x, the truth of $P(x)$ implies that there exists a z such that $R(x,z)$ is true. Proving this theorem requires that a method be discovered for finding the required z for each such x, and this method is indeed the desired program. Frequently there is no restriction on the input, in which case the theorem to be proven is $\forall x \; \exists z \; R(x,z)$.

The complete process can be illustrated by synthesizing a program to compute $f(x)=x^2+1$. Assume that f_1 and f_2 are primitive computing operations that, respectively, square and increment their arguments:

$$f_1(x) = x^2$$
$$f_2(x) = x + 1$$

Let R_1 and R_2 be predicates for the square and increment operations:

$$R_1(x, z) = \begin{cases} true & if \quad z = x^2 \\ false & \text{otherwise} \end{cases}$$
$$R_2(x, z) = \begin{cases} true & if \quad z = x + 1 \\ false & \text{otherwise} \end{cases}$$

Then $\forall x \; R_1(x,f_1(x))$ and $\forall x \; R_2(x,F_2(x))$ are true, and the program specification is $\exists y \; (R_1(x,y) \;\wedge\; R_2(y,z))$ and the following theorem must be proven:

$$\forall x \; \exists y \; \exists z \; (R_1(x,y) \;\wedge\; R_2(y,z))$$

The theorem prover might attempt many different transformations on this theorem, but one reasonable possibility would be to propose $y = f_1(x)$. Then the following theorem must be proven:

$$\forall x \; \exists z \; (R_1(x,f_1(x)) \;\wedge\; R_2(f_1(x),z))$$

Here the theorem prover might substitute $z = f_2(f_1(x))$, leaving the following theorem to be proven:

$$\forall x \; [R_1(x,f_1(x)) \;\wedge\; R_2(f_1(x),(f_2(f_1(x))))]$$

But this follows from the above assertions, thus completing the proof of the original theorem. Notice that the instantiation of z required to prove the theorem is exactly the desired program:

$$z = f_2(f_1(x))$$

4.3.3 Summary of the Deductive Approach

Rich and Waters conclusion is that “deductive methods are certain to play an important role in the automatic programming systems of the future, ... [but] the challenge is to combine automated deduction with other methods so that its inherent limitations can be avoided.” Some of the important research projects in deductive automatic programming are described in [MW71], [LCW74], and [MW92].

4.4 Transformational Methods

The transformational approach to automatic program synthesis is to take a specification written in a very high level language and to convert it into a low-level implementation through a sequence of *transformations*.

4.4.1 Overview

A transformation has three parts: a pattern, a set of logical applicability conditions, and an action procedure. When an instance of the pattern is found, the logical applicability conditions are checked to see whether the transformation should be applied. If the transformation is applicable, the action procedure is applied to produce a new code segment which replaces the code matched by the pattern. Typically, transformations are correctness preserving, meaning that the matched code and its replacement represent logically equivalent computations.

Transformations can be roughly grouped into those that change the level of abstraction in the specification (e.g. replacing quantification over a set with iteration over a list) and those that do not (e.g. moving an unchanged computation out of a loop). Most transformational automatic programming systems go through many intermediate levels of abstraction in transforming the very high level specification into a low level implementation.

4.4.2 Example of the Transformational Approach

Consider the following set of transformations:

Transformation #1:
Pattern: $\forall x \; x \in S \; F(\mathrm{x})$
Applicability Conditions: *S* is a set and *F* is a function
Action:
Create a list, *L*, of the elements in the set *S*;
Iterate *F(x)* over *L*

Transformation #2:
Pattern: Create a list, *L*, of the elements in the set, *S*
Applicability Conditions: *S* is a set containing elements of type *T*
Action:

```
List of T : L;
New(L);
while (Empty(S) == FALSE)
{
        T : element;
        element=Select_Element(S);
        Add(element,L);
        Remove(element,S);
}
```

Transformation #3:
Pattern: Iterate *F(x)* over *L*
Applicability Conditions: *L* is a list of *T*
Action:

```
T : item;
if (Empty(L) == FALSE)
{
        integer : i;
        for (i=0; i<Length(L); i++)
                F(L[i]);
}
```

Now consider the following high-level program specification:

$$\forall x \; x \in S \; Print(x)$$

This specification matches the pattern in Transformation 1 and satisfies its applicability conditions so it can be replaced by the actions in Transformation 1 to produce:

Create a list, *L*, of the elements in the set, *S*;
Iterate *Print(x)* over *L*

Now the first line of this new specification matches the pattern in Transformation 2 and satisfies its applicability conditions so the transformation can be applied to yield:

```
List of T : L;
New(L);
while (Empty(S) == FALSE)
{
      T : element;
      element=Select_Element(S);
      Add(element,L);
      Remove(element,S);
}
Iterate Print(x) over L
```

Next, Transformation 3 can be applied to the last line of the specification to give:

```
List of T : L;
New(L);
while (Empty(S) == FALSE)
{
      T : element;
      element=Select_Element(S);
      Add(element,L);
      Remove(element,S);
}
T : item;
if (Empty(L) == FALSE)
{
      integer : i;
      for (i=0; i<Length(L); i++)
            Print(L[i]);
}
```

In a more detailed example, additional transformations would probably be applied to further refine this specification until all high-level specification constructs had been removed and a low-level implementation was produced.

4.4.3 Summary of the Transformational Approach

According to Rich and Waters, “a major strength of transformational methods is that they provide a very clear representation for certain kinds of programming knowledge, [and] for this reason, transformational methods in some form are certain to be part of all

future automatic programming systems." Another contribution of the transformational approach is the transformations themselves, which encode knowledge about how to implement algorithms, create and use data structures, optimize code, and move between levels of abstraction.

Unfortunately, deciding which transformation to apply at each step is similar to deciding which inference rule to apply in each step of a deductive system. For this reason, the transformational approach suffers from some of the same rapid expansion of the search space that the preceding section on deductive methods discussed. As with deductive automatic programmers, transformational systems must either rely on the user to direct the search or place strong restrictions on the kinds of transformations that can be used. Some notable transformational automatic programming systems are the TI project [BAL85] led by Robert Balzer, and the work of Cordell Greene [GB75].

4.5 Inspection Methods

The inspective approach to automatic programming is based on the codification and use of *cliches*. A *cliche* has three parts: a skeleton that encompasses every occurrence of the cliche, roles whose contents vary from one occurrence to the next, and constraints on what can fill the roles. Most inspective systems contain algorithmic, data structure, and optimization cliches. For example, if a specification required that a set of names be alphabetized, an inspective automatic programmer might start by choosing among the cliques for various sorting algorithms. One of the roles in the chosen clique would probably be filled by the type of elements to be sorted, and another cliche would probably be used to pick an appropriate data structure to represent a collection of those elements.

The differences between inspective and transformational methods are subtle, but the main distinction is that inspective methods attempt to reduce the search-control problems

that arise with other methods by using some "global understanding" of the problem. The cliques themselves are not as important in an inspective system as are the relationships among various cliches. By contrast, transformational systems do not typically have a "global understanding" of the problem since the use of a particular transformation at one point in the process will not be used to subsequently choose other transformations. This "global understanding" of the problem is the major advantage of the inspective approach and it helps such systems to make high-level decisions before considering low-level details. The inspective approach is also closer to how humans normally program by recognizing what well-known concepts are applicable to a given problem and then applying them with some minor customizations.

As with deductive and transformational systems, it has not yet been shown that general-purpose inspective methods can be automated without advice from the user. This is due to the fact that the inspective approach is based on experience and only applies to the routine parts of programming problems. Inspective systems usually require the user to identify which cliques are applicable after which the system will typically be able to fill in many of the details without the user's assistance. MIT's Programmer's Apprentice [RICH81], [WAT85] is an example of an inspective system that works in this fashion.

4.6 Automatic Programming in the CPAL Evaluation System

As noted at the beginning of this chapter, we intended this explanation of automatic programming to prepare the reader for the presentation of our protocol analysis methodology (in Chapter 5), which contains an important automatic programming component. Recall that after having the user specify a protocol, *P*, in CPAL, and a statement of failure, *F*, for the protocol, our strategy is to take the statement of failure as a postcondition for the protocol and then employ Yasinsac's technique for computing the weakest preconditions for $P = S_1;S_2;S_3; \ldots ; S_N$.

$wp_1(S_1,...)$

$S_1;$

.

.

.

$wp_{N-2}(S_{N-2},wp(S_{N-1},wp(S_N,F)))$

S_{N-2}

$wp_{N-1}(S_{N-1},wp(S_N,F))$

$S_{N-1};$

$wp_N(S_N,F)$

$S_N;$

F

Note that satisfying any one of these weakest preconditions will cause the protocol to fail since *F*, the postcondition, is a statement of failure. Therefore, we can view each weakest precondition as the specification of a program that will cause the protocol to fail and use an automatic programming technique to attempt to generate the program from its specification. If this step succeeds, the generated program can be incorporated into the protocol and the attack scenario can be reported to the user.

In developing our automatic programming system we chose the deductive approach because we believe that we can benefit from its considerable strengths while avoiding many of its limitations. As previously mentioned, the strengths of deductive systems are that they are general, powerful, and lend themselves to automation. In fact, there are many freely available automatic theorem proving systems that can serve as the basis for a deductive automatic programming system. At the same time, the constraints our domain offers minimize the limitations of the deductive approach that were mentioned earlier.

For example, the fact that CPAL contains no looping or procedure call constructs means that we only need to generate “straight-line” programs. Furthermore, we know that most attacks on cryptographic protocols are very short, usually consisting of just a few lines

of code. For these reasons, we have found the search space to be manageable in all of the protocols we have analyzed. In addition, the CPAL system automatically generates our program specifications in logic form so we do not burden the user with a difficult specification problem. Lastly, we are interested only in whether or not there is an attack on a protocol that meets a given specification. We are not concerned about whether the generated attack is the most efficient or even reasonably efficient.

4.7 Summary

In this chapter we have given an overview of the field of automatic program generation, with an emphasis on the deductive approach. We chose the deductive approach for use in our CPAL evaluation system because of its considerable strengths, ease of implementation, and because our version of the automatic programming problem mitigate its weaknesses. In the next chapter we give a detailed description of our deductive automatic programming system and its role in our protocol analysis methodology.

Chapter 5

The CPAL Evaluation System

In this chapter we present our methodology for automatically examining cryptographic protocols for flaws. This methodology combines the mechanical nature of the state-based tools with the formal semantics of the logical approach. As discussed in Chapter 2, the logical and state-based techniques are currently the most popular approaches to cryptographic protocol examination. Unifying these two approaches has allowed us to create a system that can examine a large number of possible attacks quickly and exploit the semantics of the protocol to guide the search. We have used our methodology to produce a working cryptographic protocol evaluation system; in Chapter 6 we present the results of its search for flaws in a number of well-known protocols from the literature.

In the next section we present a broad overview of the three main steps our methodology takes to analyze cryptographic protocols. Then we give a detailed description of each step and use a common protocol as a running example to illustrate each step. In the final section we discuss our methodology and note some of its strengths and weaknesses.

5.1 Overview

We first specify the protocol(s) to be tested in the Cryptographic Protocol Analysis Language (CPAL) presented in Chapter 3. Since most protocols are expressed in Standard Notation, the user will have to perform a translation (or idealization) into CPAL. This step is generally trivial because of the similarity of the notations. After a protocol has been expressed in CPAL, the user will be asked to specify what conditions would imply *failure* of the protocol (though the user will not be required to specify *how* those conditions might be achieved). These first two steps are informal and rely heavily on the user. In the sections on these two steps we explain why these two steps are necessary and why we do not expect these steps to be difficult or time-consuming for most users.

After the user has completed these first two steps, the third step is the automatic search for an attack that satisfies the failure conditions of the protocol formulated in step two. Unlike the preceding two steps, the third step is completely formal and does not require assistance from the user. The semantics of CPAL are used to generate theorems corresponding to the failure of the protocol(s) and then a deductive automatic programming system is used to attempt to prove one of these theorems. If the proof succeeds, the actions that it generated are added in the correct places in the protocol and the undermined protocol is displayed so that the user can understand the attack that has been discovered. As we will see in Section 5.5, even when the proof does not succeed and no attack has been found we can draw some useful conclusions about the protocol(s).

5.2 Step 1 - Specifying the Protocol(s) in CPAL

We now describe the protocol that we will use as a running example for the rest of this chapter. This protocol was designed to have an obvious flaw so that it could be used to demonstrate each of the three steps in our methodology.

Consider the following scenario: agent *A* wants to send a secret, *x*, to agent *B*. Upon receipt of *A*'s message, we would like *B* to believe that *x* was sent by *A* and that *x* is not known by anyone besides *A* and himself. One possible protocol is the following. Assume that agent *A* has a public/private key pair. Recall that *A*'s public key is known to everyone but the private key is known only to her, and that messages encrypted with one of the two keys can only be decrypted with the other. Then *A* could encrypt *x* with her private key and send the result to *B*. When *B* receives this message, he could decrypt it with *A*'s public key and learn *x*. This protocol would be expressed in the Standard Notation as follows:

$$A \rightarrow B: \qquad \{x\}\mathrm{KA}^{-1}$$

The flaw in this protocol is obvious. Since *A*'s public key is well known, anyone who can intercept the message from *A* to *B* can perform the decryption and learn *x*. In the next few sections we demonstrate how this weakness can be uncovered automatically by our methodology.

The first step we take is to translate the protocol into a CPAL representation:

(1) *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

(2) *X*: **assume**(*B.pub_A == A.pub_A);*

(3) *X*: **assume**(*I.pub_A == A.pub_A);*

(4) *A*: *x* := *new*;

(5) *A*: $\rightarrow B$ *(ep[x]priv_A)*;

(6) *B*: $\leftarrow (msg)$;

(7) *B*: *x* := *dp[msg]pub_A*;

Line 1 of the CPAL specification defines agent *A*'s public/private key pair, and lines 2 and 3 state that agents *B* and *I* know *A*'s public key. In step 4, *A* creates a new value and sends it to *B* in line 5. Agent *B* receives *A*'s message and decrypts it to learn *x* in lines 6 and 7, respectively.

5.3 Step 2 - Specifying the Failure Condition

The next step is to specify a failure condition for the protocol. This step is not unlike the "specification acquisition" step in automatic programming which many researchers consider a "major problem" [BAL85] due to the difficulty of producing a specification that is correct, complete, and unambiguous. While we cannot claim that specifying the failure condition for a cryptographic protocol is trivial there are some characteristics of the limited domain that make this problem easier than the general version which the automatic programming community must deal with. For one thing these failure conditions are typically much shorter than program specifications and are therefore easier to express and understand.

Furthermore, there are generally a small number of standard failure conditions for each type of protocol, and the failure condition for a protocol is often simply the negation of the protocol's stated goals. For example, an authentication protocol would be said to fail if agent C could convince agent A that he (C) was B. A natural failure condition for a key distribution protocol would be one that states that the "secret" key that is agreed upon by A and B is also known by an intruder. Another possible failure for a key distribution protocol would be if the two principals believe that they share a key but do not realize that they haven't agreed on the same key.

Note that the failure conditions given above describe what it means for the protocol to fail but do not specify how that failure might be brought about. In the key distribution example an intruder could accomplish the first failure condition (knowing the "secret" key agreed on by the two principals) by getting them to accept as the "secret" key some value she already knows, by tricking one of the two agents into telling her the key they have agreed on, or through some other sequence of actions. Allowing the user to express the failure condition in general terms not only makes these specifications simpler, but it also leaves

open the possibility that the system will discover a way to attack a protocol that the user might not have foreseen.

In our current implementation, we restrict the user to expressing failure conditions in propositional calculus. We made this choice because a substantial number of failure conditions can be expressed in this notation and because limiting our theorems to statements in propositional calculus greatly decreased the amount of time it took to prove the resulting theorems. In the Related Work section of Chapter 7, we discuss how we would go about modifying our system so that failure conditions could be expressed in first-order logic or any of the specialized modal logics that have been developed for cryptographic protocol analysis.

Returning to the example protocol that we introduced in Section 5.3, we recall that it was intended to transfer a secret piece of data from *A* to *B*. One obvious failure condition for that protocol would be if the intruder was somehow able to learn the data that the protocol is supposed to protect. We can express this statement in predicate calculus as *Same(I.x, A.x)*. This predicate is true if a variable in the intruder's address space (*x* in this case) is the same as the "secret" data, *x*, from *A*'s address space. Here then is the CPAL specification for the protocol and its failure condition expressed in predicate calculus which together form the input for the third step of our methodology:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\rightarrow B$ *(ep[x]priv_A)*;

B: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_A*;

Same(I.x, A.x)

5.4 Step 3 - Automatically Searching for an Attack

The final step in our methodology is to attempt to modify the protocol so that the failure condition is achieved. If we return to the definition of weakest precondition given in Chapter 3, we recall that the *weakest precondition*, $wp(S,R)$, represents the weakest solution to the equation $Q : Q \{S\} R$, where $Q \{S\} R$ is a *Hoare triple* with Q and R being boolean expressions called the precondition and postcondition, respectively. $Q \{S\} R$ is said to hold if program segment S, beginning with Q satisfied, is guaranteed to establish R, if S terminates. By using the formal semantics of CPAL we can compute the weakest precondition at each point in the protocol to determine what assumptions, Q_i, must hold prior to each statement in the protocol, S_i, for the remainder of the protocol to execute and establish the postcondition, R. So for the protocol, P, composed of a sequence of statements:

$$P = S_1;S_2;S_3; \ldots ; S_N$$

and the failure condition, F, we can compute $wp(P,F)$:

$$
\begin{array}{l}
\quad wp_1(S_1,\ldots) \\
S_1; \\
. \\
. \\
. \\
\quad wp_{N-2}(S_{N-2},wp(S_{N-1},wp(S_N,F))) \\
S_{N-2} \\
\quad wp_{N-1}(S_{N-1},wp(S_N,F)) \\
S_{N-1}; \\
\quad wp_N(S_N,F) \\
S_N; \\
\quad F
\end{array}
$$

Each wp_i is then the assumptions that must hold prior to S_i for $S_i;S_{i+1}; \ldots ; S_{N-1}; S_N$ to execute and establish *F*. Therefore, we can think of each wp_i as a formal specification of a program that, if added to the protocol before statement S_i, guarantees that the protocol will have achieved the failure condition, *F*, upon completion. Recall from Chapter 4 that we can apply a deductive automatic programming approach to try to find a constructive proof of wp_i and, if that succeeds, we can add the generated program, *GP*, to the protocol before statement S_i.

$$
\begin{array}{l}
\quad wp_1(S_1,\ldots) \\
S_1; \\
. \\
. \\
S_{i-1}; \\
GP; \\
S_i; \\
. \\
. \\
\quad wp_{N-2}(S_{N-2},wp(S_{N-1},wp(S_N,F))) \\
S_{N-2} \\
\quad wp_{N-1}(S_{N-1},wp(S_N,F)) \\
S_{N-1}; \\
\quad wp_N(S_N,F) \\
S_N; \\
\quad F
\end{array}
$$

Since wp_i are the assumptions that must hold prior to S_i for $S_i;S_{i+1}; \ldots ; S_{N-1}; S_N$ to execute and establish *F*, and since *G* establishes wp_i, $P' = S_1;S_2; \ldots S_{i-1};GP;S_i; \ldots ; S_{N-1};S_N$ is an example of a successful attack on *P*. So, basically, step three of our methodology computes the weakest preconditions for the protocol and then tries to find a constructive proof for one of them. Once a proof is found, the modifications it prescribes are made to the protocol and the resulting attack scenario is reported to the user.

Actually, our methodology does not require that an entire weakest precondition be proved all at once. In our current implementation, for example, we transform all the weakest preconditions into conjunctive normal form and then try to prove any clause in a weakest precondition. If a proof succeeds, we make the necessary modifications to the protocol, recompute the weakest preconditions for the modified protocol, and then try to prove another clause. We continue with this process until one of the weakest preconditions has had all of its clauses proved. This could give rise to extra actions being added to the protocol that are not part of a successful attack, but this does not concern us since we are primarily interested in whether or not an attack exists. In the next subsection, we give the weakest preconditions for our example protocol. In the subsections after that, we show exactly how we prove theorems and use those proofs to modify protocols.

5.4.1 Computing the Weakest Preconditions

Computing the weakest preconditions for a protocol expressed in CPAL is simply an exercise in mechanically applying the predicate transforms that were given in Chapter 3. Yasinsac developed such a tool as part of his dissertation and we currently use a version of that program in our system. The results of computing the weakest preconditions for the example protocol and the failure condition are given below. As noted in Chapter 3, before computing the weakest precondition for a protocol, all insecure sends are replaced with secure sends to the intruder, who then forwards the message unmodified to its intended recipient.

Same(I.x, unique1)

(1) *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

Same(I.x, unique1)

(2) *X*: **assume**(*B.pub_A == A.pub_A);*

Same(I.x, unique1)

(3) *X*: **assume**(*I.pub_A == A.pub_A);*

Same(I.x, unique1)

(4) *A*: *x* := *new*;

Same(I.x, A.x)

(5) *A*: $\Rightarrow I$ *(ep[x]priv_A)*;

Same(I.x, A.x)

(6) *I*: $\leftarrow (tmp1)$;

Same(I.x, A.x)

(7) *I*: $\Rightarrow B$ *(tmp1)*;

Same(I.x, A.x)

(8) *B*: $\leftarrow (msg)$;

Same(I.x, A.x)

(9) *B*: *x* := *dp[msg]pub_A*;

Same(I.x, A.x)

Figure 5.1 : The Protocol (with Weakest Preconditions) in CPAL

5.4.2 Searching for a Constructive Proof

The next step after computing the weakest preconditions for a protocol is to use an automatic theorem proving system to try to find a constructive proof of one of the weakest preconditions. Many different axioms and inference rules make up the input that will be given to the automatic theorem prover, but they can all be generated automatically or derived from the protocol or its weakest preconditions. In the next several subsections we present all of the axioms and inference rules that we currently include, and we discuss why we chose each axiom/rule.

The axioms and inference rules we currently use can be divided into *standard* rules and axioms that do not differ from protocol to protocol, and the *protocol-specific* axioms that depend on the protocol. We discuss each of these sets in the following two subsections.

5.4.2.1 Standard Axioms and Inference Rules

The standard axioms which we define mostly describe some useful predicates that allow us to reason about some basic properties of a protocol and the underlying cryptographic algorithms. One of these predicates which we have already seen is the *Same()* predicate which we have already used to specify the failure condition. We use the *Same()* predicate to express the equivalence of two values either in the same address space or in different address spaces. Three of the standard axioms that deal with the *Same()* predicate are the following:

Same(X, X).
-Same(X,Y) | Same(Y, X).
-Same(X,Y) | -Same(Y,Z) | Same(X,Z)

The first axiom states that the *Same()* predicate is reflexive - meaning that any value, *X*, is "the same" as itself. The second axiom uses negation (-) and disjunction (|) to state that the *Same()* predicate is also symmetric. Translating the second axiom literally, it reads: "Either *X* and *Y* are not the same, or *Y* and *X* are the same." Readers familiar with logical notation should recognize that this axiom could be rewritten as:

not(*Same*(*X*,*Y*)) **or** *Same*(*Y*,*X*)

which is equivalent to the statement:

Same(*X*,*Y*) **implies** *Same*(*Y*,*X*)

under the usual interpretations. The last axiom states that the *Same()* predicate is also transitive.

Another predicate that we will use (especially when dealing with protocols that employ public-key cryptosystems) is the *Inverse()* predicate. This predicate is used to describe pairs of values which, when used with some function, can each be used to invert the results of the function applied to the other. Two of our standard axioms about the *Inverse()* predicate are:

-Inverse(X,Y) | Inverse(Y,X).
-Same(X,Y) | -Inverse(Y,Z) | Inverse(X,Z).

The first axiom tells us that the *Inverse()* predicate is symmetric just like the *Same()* predicate. Note that there are no transitivity or reflexivity axioms for the *Inverse()* predicate. However, the second axiom above states that if *X* and *Y* are the same and *Y* and *Z* are inverses, then *X* and *Z* are also inverses.

Two other predicates that we use often are the *Iknows()* and *Icontrols()* predicates. We use these predicates to reason about what values the intruder knows and controls, respectively. We say that the intruder *knows* a value if he can recognize and repeat it. We say that the intruder *controls* a value if he *knows* it and can also overwrite it with another value. Two of our standard axioms that deal with these two predicates are the following:

-Iknows(X) | -Same(X,Y) | Iknows(Y) | $ANS(Sub(Y,X)).
-Icontrols(X) | -Iknows(Y) | Same(X,Y) | $ANS(Assign(X,Y)).

The first axiom states that if the intruder knows *X* and *X* is the same as *Y*, then the intruder knows *Y*. The last clause in this axiom, *$ANS()*, is called an *answer literal* and is used to record instantiations of variables in input clauses during the search for a proof. Given the logical formulae below, with lower case names representing *literals* and upper case names representing *variables*:

Iknows(a).
Same(a,b).
-Iknows(X) | -Same(X,Y) | Iknows(Y) | $ANS(Sub(Y,X)).

a proof of the theorem *Iknows(b)* would yield the answer literal:

$ANS(Sub(b,a)).

which tells us that during the proof, *b* instantiated the variable *Y* and *a* instantiated the variable *X*. When we construct protocol actions this will allow us to substitute the correct values from the protocol into those actions.

In the axiom given above for the *Icontrols()* predicate we see that if the intruder controls some value, *X*, and knows some value, *Y*, then she can cause *X* and *Y* to have the same value by overwriting *X* with *Y*. This axiom also contains an answer literal that will tell us which value was overwritten with which value during the proof.

The remainder of the standard axioms deal with some of the properties of symmetric and asymmetric cryptosystems. They are only added to the input file if our system determines that a protocol makes use of the corresponding cryptographic functions. For symmetric cryptosystems the axioms are:

-Iknows(K) | -Iknows(e(X,K)) | Iknows(X) | $ANS(Sub(X,d(e(X,K),K))). : If the intruder knows a value, *K*, and a message encrypted using that value, *e(X,K)*, then he also knows the contents of the message, *X*

-Iknows(K) | -Iknows(d(X,K)) | Iknows(X) | $ANS(Sub(X,e(d(X,K),K))). : If the intruder knows a value, *K*, and a message decrypted using that value, *d(X,K)*, then he also knows the contents of the message, *X*

-Same(K1,K2) | Same(d(e(X,K1),K2),X). : Encrypting a message, *X*, and then decrypting the result using the same key yields *X*.

-Same(K1,K2) | Same(e(d(X,K1),K2),X). : Decrypting a message, *X*, and then encrypting the result using the same key yields *X*.

-Same(X,Y) | -Iknows(e(X,K)) | Iknows(e(Y,K)) | $ANS(Sub(X,Y)). : If *X* and *Y* are the same and the intruder knows the encryption of *X* under some key, *K*, then he also knows the encryption of *Y* under *K*.

-Same(X,Y) | -Iknows(d(X,K)) | Iknows(d(Y,K)) | $ANS(Sub(X,Y)). : If *X* and *Y* are the same and the intruder knows the decryption of *X* under some key, *K*, then he also knows the decryption of *Y* under *K*.

-Iknows(K) | -Iknows(X) | Iknows(e(X,K)) | $ANS(Create(e(X,K))). : If the intruder knows some value, *K*, and some other value, *X*, then he can produce the encryption of *X* under *K*.

-Iknows(K) | -Iknows(X) | Iknows(d(X,K)) | $ANS(Create(d(X,K))). : If the intruder knows some value, *K*, and some other value, *X*, then he can produce the decryption of *X* under *K*.

-Icontrols(X) | -Iknows(Y) | -Iknows(d(X,K)) | Iknows(d(Y,K)) | $ANS(Sub(X,Y)). : When the intruder controls a value, *X*, knows a value, *Y*, and knows the decryption of *X* with *K*, then he can learn the decryption of *Y* with *K* by overwriting *X* with *Y* and then performing whatever actions allowed him to discover *d(X,K)*.

-Icontrols(X) | -Iknows(Y) | -Iknows(e(X,K)) | Iknows(e(Y,K)) | $ANS(Sub(X,Y)). : When the intruder controls a value, *X*, knows a value, *Y*, and knows the encryption of *X* with *K*, then he can learn the encryption of *Y* with *K* by overwriting *X* with *Y* and then performing whatever actions allowed him to discover *e(X,K)*.

For asymmetric cryptosystems the axioms are:

-Iknows(K1) | -Inverse(K1,K2) | -Iknows(ep(X,K2)) | Iknows(X) |

$ANS(Sub(X,dp(ep(X,K2),K1))). : If the intruder knows a value, *K1*, and *K1* is the inverse of *K2*, and she also knows the public-key encryption of *X* under *K2*, then she can decrypt the message using *K1* and learn *X*.

-iknows(K1) | -Inverse(K1,K2) | -Iknows(dp(X,K2)) | Iknows(X) |

$ANS(Sub(X,ep(dp(X,K2),K1))). : If the intruder knows a value, *K1*, and *K1* is the inverse of *K2*, and she also knows the public-key decryption of *X* under *K2*, then she can encrypt the message using *K1* and learn *X*.

-Inverse(K1,K2) | Same(dp(ep(X,K1),K2),X). : Encrypting a message, *X*, and then decrypting the result using inverse keys yields *X*.

-Inverse(K1,K2) | Same(ep(dp(X,K1),K2),X). : Decrypting a message, *X*, and then encrypting the result using inverse keys yields *X*.

-Same(X,Y) | -Iknows(ep(X,K)) | Iknows(ep(Y,K)) | $ANS(Sub(X,Y)). : If two values, *X* and *Y*, are the same and an intruder knows the public-key encryption of one then she also knows the public-key encryption of the other.

-Same(X,Y) | -Iknows(dp(X,K)) | Iknows(dp(Y,K)) | $ANS(Sub(X,Y)). : If two values, *X* and *Y*, are the same and an intruder knows the public-key decryption of one then she also knows the public-key decryption of the other.

-Iknows(K) | -Iknows(X) | Iknows(ep(X,K)) | $ANS(Create(ep(X,K))). : If the intruder knows some value, *K*, and some other value, *X*, then he can produce the public-key encryption of *X* under *K*.

-Iknows(K) | -Iknows(X) | Iknows(dp(X,K)) | $ANS(Create(dp(X,K))). : If the intruder knows some value, *K*, and some other value, *X*, then he can produce the public-key decryption of *X* under *K*.

-Icontrols(X) | -Iknows(Y) | -Iknows(dp(X,K)) | Iknows(dp(Y,K)) | $ANS(Sub(X,Y)).: When the intruder controls a value, *X*, knows a value, *Y*, and knows the public-key decryption of *X* with *K*, then she can learn the public-key decryption of *Y* with *K* by overwriting *X* with *Y* and then performing whatever actions allowed her to discover *dp(X,K)*.

-Icontrols(X) | -Iknows(Y) | -Iknows(ep(X,K)) | Iknows(ep(Y,K)) | $ANS(Sub(X,Y)).: When the intruder controls a value, *X*, knows a value, *Y*, and knows the public-key encryption of *X* with *K*, then she can learn the public-key encryption of *Y* with *K* by overwriting *X* with *Y* and then performing whatever actions allowed her to discover *ep(X,K)*.

These axioms are not intended to allow us to prove everything that could be proven about these predicates but were chosen because they allow us to prove many useful things about these predicates quickly. We discuss this issue more formally and in more depth later in Section 5.5.

5.4.2.2 Protocol-Specific Axioms

The rest of the axioms that make up the input file for the automatic theorem prover are derived from the protocol and from the particular weakest precondition that we are attempting to prove. Since we could attempt to prove any of the weakest preconditions in the protocol, our approach is to work on the weakest precondition for the first statement of

the protocol first, and, if it has not been proven in some user-specified amount of time, move on to the next weakest precondition. If at any time a proof succeeds and the protocol is modified we recompute the weakest preconditions for the new protocol and start again with the weakest precondition for the first statement. We continue in this manner until one or more of the weakest preconditions contains no clauses (meaning we have satisfied a weakest precondition and discovered a successful attack) or until no part of any weakest precondition can be proved at which point the system reports it could not find any successful attacks in the allotted time.

For the purpose of this discussion let us assume that we have failed to prove any part of the weakest preconditions for statements 1-6 in Figure 5.1 and are now considering the weakest precondition for statement 7. It is:

Same(i_x, a_x).

This is the theorem to be proved and, as is usual in resolution theorem proving, we add the negation of this theorem to our set of axioms:

-Same(i_x, a_x).

Next, we add axioms corresponding to any assumptions that are given in the protocol under consideration. In this case, that would be those given in lines 1-3 of Figure 5.1:

Inverse(a_pub_A, a_priv_A).
Same(b_pub_A, a_pub_A).
Same(i_pub_A, a_pub_A).

Then we generate axioms that describe for which variables in the protocol the *Iknows()*, *Icontrols()*, and *Same()* predicates hold.

For the first two, it is always the case that the intruder both knows and controls all the values in her own address space so from Figure 5.1 we get the axioms:

Iknows(i_tmp1).
Icontrols(i_tmp1).
Iknows(i_x).
Icontrols(i_x).
Iknows(i_pub_A).
Icontrols(i_pub_A).

To derive a *Same()* predicate for each variable in the protocol we use the variable as a "postcondition" and compute the weakest precondition from that point in the protocol to the beginning. In this example, we are attempting to prove the weakest precondition for statement 7 so the *Same()* predicate for the variable *I.tmp1* would be computed as follows:

ep[unique1]A.priv_A

(1) *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

ep[unique1]A.priv_A

(2) *X*: **assume**(*B.pub_A == A.pub_A);*

ep[unique1]A.priv_A

(3) *X*: **assume**(*I.pub_A == A.pub_A);*

ep[unique1]A.priv_A

(4) *A*: *x* := *new*;

ep[A.x]A.priv_A

(5) *A*: $\Rightarrow I$ *(ep[x]priv_A)*;

*I.*queue**

(6) *I*: $\leftarrow (tmp1)$;

I.tmp1

This gives us the axiom:

Same(i_tmp1, ep(unique1, a_priv_A)).

By repeating this process for the other variables in the protocol we get:

Same(a_x, unique1).
Same(a_pub_A, a_pub_A).

Same(a_priv_A, a_priv_A).
Same(b_msg, b_msg).
Same(b_pub_A, b_pub_A).
Same(i_x, i_x).
Same(i_pub_A, i_pub_A).

5.4.2.3 Parallel Session Axioms

The final set of axioms we derive from the protocol are the *parallel-session axioms* which describe the effect of running fragments of the protocol. These axioms will allow us to reason about *parallel-session attacks* in which the intruder may participate in more than one session of a protocol simultaneously and use messages from one session in another. These types of attacks are very common on cryptographic protocols and examples of successful parallel session attacks can be found in [AN94], [BIR93], [BAN89], [CAR94], and [KMM94], to name a few.

A parallel session fragment is just some initial prefix of a protocol with agents assigned to each "role" in the protocol. For example, the protocol in Figure 5.2 begins with some agent playing the role associated with agent A in the specification. It is important to

note that any agent can play the role of “A” in this protocol. So if agent B wants to transmit a piece of secret data to agent A the protocol will look like:

(1) *X*: **assume**(*Inverse(B.pub_B, B.priv_B));*

(2) *X*: **assume**(*A.pub_B == B.pub_B);*

(3) *X*: **assume**(*I.pub_B == B.pub_B);*

(4) *B*: *x* := *new*;

(5) *B*: $\Rightarrow I$ *(ep[x]priv_B)*;

(6) *I*: $\leftarrow (tmp1)$;

(7) *I*: $\Rightarrow A$ *(tmp1)*;

(8) *A*: $\leftarrow (msg)$;

(9) *A*: *x* := *dp[msg]pub_B*;

Figure 5.2 : A Sample Protocol

In this instance of the protocol we would say that agent B is playing the role of “A” in the protocol and that agent A is playing the role of “B” (Note that the intruder is playing the role of “I” and is the only agent who can, since the role of “I” must be played by a dishonest principal). Likewise, if agent A uses the protocol to send a secret to agent C the protocol looks like:

(1) *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

(2) *X*: **assume**(*C.pub_A == A.pub_A);*

(3) *X*: **assume**(*I.pub_A == A.pub_A);*

(4) *A*: *x* := *new*;

(5) *A*: $\Rightarrow I$ *(ep[x]priv_A)*;

(6) *I*: $\leftarrow (tmp1)$;

(7) *I*: $\Rightarrow C$ *(tmp1)*;

(8) *C*: $\leftarrow (msg)$;

(9) *C*: *x* := *dp[msg]pub_A*;

Now, if we break the protocol down into turns:

Turn 1: *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

Turn 2: *I*: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Turn 3: *B*: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_A*;

and enumerate all valid prefixes of the protocol (a valid prefix of a protocol must start with the first statement of the protocol and end on a turn boundary) we get the following:

Prefix 1:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

Prefix 2:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Prefix 3:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

B: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_A*;

Prefix 1 gives us 3 distinct parallel session fragments (since agent *A*, *B*, or *I* can play the role of "A" in Prefix 1). They are:

Parallel Session 1 (*A* plays "A"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

Parallel Session 2 (*B* plays "A"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow I$ *(ep[x]priv_B)*;

Parallel Session 3 (*I* plays "A"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow$ *I (ep[x]priv_I)*;

Prefix 2 gives us 3 more parallel session fragments (since agent *A*, *B*, or *I* can play the role of "A" and only the intruder can play the role of "I"):

Parallel Session 4 (*A* plays "A" and *I* plays "I"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow$ *I (ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow$ *B (tmp1)*;

Parallel Session 5 (*B* plays "A" and *I* plays "I"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow$ *I (ep[x]priv_B)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow$ *B (tmp1)*;

Parallel Session 6 (*I* plays "A" and *I* plays "I"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow$ *I (ep[x]priv_I)*;

I: $\leftarrow(tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Finally, Prefix 3 gives us 9 more parallel session fragments (since agent *A*, *B*, or *I* can play the role of "A", only *I* can play the role of "I", and agent *A*, *B*, or *I* can play the role of "B") of which we list only the first below. A complete list of all 15 parallel session fragments for the protocol in Figure 5.2 is given in Appendix B.

Parallel Session 7 (*A* plays "A", *I* plays "I", and *A* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow(tmp1)$;

I: $\Rightarrow A$ *(tmp1)*;

A: $\leftarrow(msg)$;

A: *x* := *dp[msg]pub_A*;

Each parallel session fragment can be used as an axiom when attempting to prove a weakest precondition though we may want to eliminate duplicate or "useless" parallel session fragments from the list in order to minimize the number of axioms the automatic theorem prover needs to consider.

The last thing we must do to the list of parallel session fragments is to rename some of the variables. We do this because we expect that certain of the variables in these fragments will refer to a different memory location in the principal's address space on each run of the protocol. For instance, in the statement:

B: $\leftarrow(msg)$;

we would not expect the variable *msg* to refer to the same location in *B*'s address space in each run of the protocol. On the other hand, in the statement:

A: $\Rightarrow I$ *(ep[x]priv_A)*;

we would expect the variable *priv_A* to refer to the same location in *A*'s address space in each run of the protocol since it corresponds to a permanent and global value. By performing these substitutions and using the same procedure (described in Section 5.4.2.2) used to generate protocol-specific axioms we get the following list of parallel-session axioms which are added to the input file for the automatic theorem prover:

Same(a_tmp1, unique2) | $ANS(session(1)).
Same(b_tmp1, unique3) | $ANS(session(2)).
(Same(a_tmp2, unique4) & Iknows(i_pub_A) & Icontrols(i_pub_A) &
    Same(i_pub_A, a_pub_A) & Iknows(i_tmp2) & Icontrols(i_tmp2) &
    Same(i_tmp2, ep(unique4, a_priv_A))) | $ANS(session(3)).
(Same(b_tmp2, unique5) & Iknows(i_pub_B) & Icontrols(i_pub_B) &
    Same(i_pub_B, b_pub_B) & Iknows(i_tmp3) & Icontrols(i_tmp3) &
    Same(i_tmp3, ep(unique5, b_priv_B))) | $ANS(session(4)).
(Same(a_tmp3, unique6) & Iknows(i_pub_A) & Icontrols(i_pub_A) &
    Same(i_pub_A, a_pub_A) & Iknows(i_tmp4) & Icontrols(i_tmp4) &
    Same(i_tmp4,ep(unique6,a_priv_A)) & Same(a_msg ,ep(unique6,a_priv_A)) &
    Same(a_tmp4,dp(ep(unique6, a_priv_A), a_pub_A) | $ANS(session(5)).
(Same(a_tmp5, unique7) & Iknows(i_pub_A) & Icontrols(i_pub_A) &
    Same(i_pub_A, a_pub_A) & Iknows(i_tmp5) & Icontrols(i_tmp5) &
    Same(i_tmp5,ep(unique7,a_priv_A)) &Same(b_tmp3,ep(unique7,a_priv_A)) &
    Same(b_tmp4,dp(ep(unique7, a_priv_A), b_pub_A) | $ANS(session(6)).
(Iknows(i_pub_A) & Icontrols(i_pub_A) & Same(i_pub_A, a_pub_A) &
    Iknows(i_tmp6) & Icontrols(i_tmp6) & Same(a_tmp6, i_tmp6) &
    Same(a_tmp7,dp(i_tmp6, a_pub_A)) | $ANS(session(7)).
(Iknows(i_pub_B) & Icontrols(i_pub_B) & Same(i_pub_B, b_pub_B) &
    Iknows(i_tmp7) & Icontrols(i_tmp7) & Same(b_tmp4, i_tmp7) &
    Same(b_tmp5,dp(i_tmp7, a_pub_A)) | $ANS(session(8)).
(Same(b_tmp6, unique8) & Iknows(i_pub_B) & Icontrols(i_pub_B) &

Same(i_pub_B, b_pub_B) & Iknows(i_tmp8) & Icontrols(i_tmp8) &
Same(i_tmp8,ep(unique8,b_priv_B)) &Same(a_tmp7,ep(unique8,b_priv_B)) &
Same(a_tmp8,dp(ep(unique8, b_priv_B), a_pub_B) | $ANS(session(9)).
(Same(b_tmp7, unique9) & Iknows(i_pub_B) & Icontrols(i_pub_B) &
Same(i_pub_B, a_pub_B) & Iknows(i_tmp9) & Icontrols(i_tmp9) &
Same(i_tmp9,ep(unique9,b_priv_B)) &Same(b_tmp8,ep(unique9,b_priv_B)) &
Same(b_tmp9,dp(ep(unique9, b_priv_B), b_pub_B) | $ANS(session(10)).

5.4.3 Protocol Interactions

Note that there is no reason to limit the parallel-session axioms to only those derived from a single protocol. We could generate parallel-session axioms for a number of different protocols and add them all to the input file that we use to examine any one of them. If any of these axioms are used during the proof, the corresponding actions would be added to the protocol yielding an attack that exploited one of the other protocols in the set to undermine the protocol being examined. We consider this to be an important contribution of our approach. An example of this type of *protocol-interaction attack* is shown in Chapter 6.

5.4.4 The Automatic Theorem Prover's Input File

Now that we have explained each of its components, we can present the complete input file that will be used by the automatic theorem prover.

```
set(auto).
set(prolog_style_variables).
assign(max_seconds,5).
list(usable).
% =================== Standard Axioms ===================
Same(X, X).
-Same(X,Y) | Same(Y, X).
-Same(X,Y) | -Same(Y,Z) | Same(X,Z).
-Inverse(X,Y) | Inverse(Y,X).
-Same(X,Y) | -Inverse(Y,Z) | Inverse(X,Z).
-Iknows(X) | -Same(X,Y) | Iknows(Y) | $ANS(Sub(Y,X)).
-Icontrols(X) | -Iknows(Y) | Same(X,Y) | $ANS(Assign(X,Y)).
% =================== Public-Key Rules ===================
```

```
-Iknows(K1) | -Inverse(K1,K2) | -Iknows(ep(X,K2)) | Iknows(X) | $ANS(Sub(X,dp(ep(X,K2),K1))).
-iknows(K1) | -Inverse(K1,K2) | -Iknows(dp(X,K2)) | Iknows(X) | $ANS(Sub(X,ep(dp(X,K2),K1))).
-Inverse(K1,K2) | Same(dp(ep(X,K1),K2),X).
-Inverse(K1,K2) | Same(ep(dp(X,K1),K2),X).
-Same(X,Y) | -Iknows(ep(X,K)) | Iknows(ep(Y,K)) | $ANS(Sub(X,Y)).
-Same(X,Y) | -Iknows(dp(X,K)) | Iknows(dp(Y,K)) | $ANS(Sub(X,Y)).
-Iknows(K) | -Iknows(X) | Iknows(ep(X,K)) | $ANS(Create(ep(X,K))).
-Iknows(K) | -Iknows(X) | Iknows(dp(X,K)) | $ANS(Create(dp(X,K))).
-Icontrols(X) | -Iknows(Y) | -Iknows(dp(X,K)) | Iknows(dp(Y,K)) | $ANS(Sub(X,Y)).
-Icontrols(X) | -Iknows(Y) | -Iknows(ep(X,K)) | Iknows(ep(Y,K)) | $ANS(Sub(X,Y)).
% ============== Protocol-Specific Axioms ==============
Iknows(i_tmp1).
Icontrols(i_tmp1).
Iknows(i_x).
Icontrols(i_x).
Iknows(i_pub_A).
Icontrols(i_pub_A).
Same(i_tmp1, ep(unique1, a_priv_A)).
Same(a_x, unique1).
Same(a_pub_A, a_pub_A).
Same(a_priv_A, a_priv_A).
Same(b_msg, b_msg).
Same(b_pub_A, b_pub_A).
Same(i_x, i_x).
Same(i_pub_A, i_pub_A).
% ============= Protocol Assumptions =================
Inverse(a_pub_A, a_priv_A).
Same(b_pub_A, a_pub_A).
Same(i_pub_A, a_pub_A).
end_of_list.
% =========== Negation of the Weakest Precondition =========
formula_list(usable).
-Same(i_x, unique1).
% ============= Parallel Session Axioms ================
Same(a_tmp1, unique2) | $ANS(session(1)).
Same(b_tmp1, unique3) | $ANS(session(2)).
(Same(a_tmp2, unique4) & Iknows(i_pub_A) & Icontrols(i_pub_A) & Same(i_pub_A, a_pub_A) &
    Iknows(i_tmp2)&Icontrols(i_tmp2)&Same(i_tmp2,ep(unique4,a_priv_A))) |$ANS(session(3)).
(Same(b_tmp2, unique5) & Iknows(i_pub_B) & Icontrols(i_pub_B) & Same(i_pub_B, b_pub_B) &
    Iknows(i_tmp3)&Icontrols(i_tmp3)&Same(i_tmp3,ep(unique5,b_priv_B)))|$ANS(session(4)).
(Same(a_tmp3, unique6) & Iknows(i_pub_A) & Icontrols(i_pub_A) &Same(i_pub_A, a_pub_A) &
    Iknows(i_tmp4) & Icontrols(i_tmp4) &Same(i_tmp4,ep(unique6,a_priv_A)) &
    Same(a_msg ep(unique6,a_priv_A))&Same(a_tmp4,dp(ep(unique6,a_priv_A),a_pub_A)|
```

$ANS(session(5)).
(Same(a_tmp5, unique7) & Iknows(i_pub_A) & Icontrols(i_pub_A) & Same(i_pub_A, a_pub_A) &
Iknows(i_tmp5) & Icontrols(i_tmp5) & Same(i_tmp5,ep(unique7,a_priv_A)) &
Same(b_tmp3, ep(unique7, a_priv_A)) & Same(b_tmp4, dp(ep(unique7, a_priv_A), b_pub_A) |
$ANS(session(6)).
(Iknows(i_pub_A) & Icontrols(i_pub_A) & Same(i_pub_A, a_pub_A) & Iknows(i_tmp6) &
Icontrols(i_tmp6) & Same(a_tmp6, i_tmp6) & Same(a_tmp7,dp(i_tmp6, a_pub_A)) |
$ANS(session(7)).
(Iknows(i_pub_B) & Icontrols(i_pub_B) & Same(i_pub_B, b_pub_B) &Iknows(i_tmp7) &
Icontrols(i_tmp7) & Same(b_tmp4, i_tmp7) &Same(b_tmp5,dp(i_tmp7, a_pub_A)) |
$ANS(session(8)).
(Same(b_tmp6, unique8) & Iknows(i_pub_B) & Icontrols(i_pub_B) &
Same(i_pub_B, b_pub_B) & Iknows(i_tmp8) & Icontrols(i_tmp8) &
Same(i_tmp8,ep(unique8,b_priv_B)) &Same(a_tmp7,ep(unique8,b_priv_B)) &
Same(a_tmp8,dp(ep(unique8, b_priv_B), a_pub_B) | $ANS(session(9)).
(Same(b_tmp7, unique9) & Iknows(i_pub_B) & Icontrols(i_pub_B) &
Same(i_pub_B, a_pub_B) & Iknows(i_tmp9) & Icontrols(i_tmp9) &
Same(i_tmp9,ep(unique9,b_priv_B)) &Same(b_tmp8,ep(unique9,b_priv_B)) &
Same(b_tmp9,dp(ep(unique9, b_priv_B), b_pub_B) | $ANS(session(10)).
end_of_list.

Figure 5.3 : The Input File for OTTER

For this input file, the automatic theorem prover produces the following output:

----> UNIT CONFLICT at 0.08 sec ----> 277 [binary,276.1,12.1] $ANS(sub(ep(uniq
ue1,a_priv_a),i_tmp1))|$ANS(sub(unique1,dp(ep(unique1,a_priv_a),b_pub_a)))|$ANS(
sub(b_pub_a,a_pub_a))|$ANS(sub(a_pub_a,i_pub_a))|$ANS(sub(a_x,unique1))|$ANS(ass
ign(i_x,a_x)).

Length of proof is 9. Level of proof is 5.

---------------- PROOF ----------------
1 [] -same(A,B)|same(B,A).
3 [] -same(A,B)| -inverse(B,C)|inverse(A,C).
4 [] -icontrols(A)| -iknows(B)|same(A,B)|$ANS(assign(A,B)).
5 [] -iknows(A)| -same(A,B)|iknows(B)|$ANS(sub(B,A)).
6 [] -iknows(A)| -inverse(A,B)| -iknows(ep(C,B))|iknows(C)|$ANS(sub(C,dp(ep(C,B)
,A))).
12 [] -same(i_x,a_x).

```
14 [] same(i_tmp1,ep(unique1,a_priv_a)).
15 [] iknows(i_tmp1).
18 [] icontrols(i_x).
19 [] iknows(i_pub_a).
21 [] same(a_x,unique1).
22 [] inverse(a_pub_a,a_priv_a).
23 [] same(b_pub_a,a_pub_a).
24 [] same(i_pub_a,a_pub_a).
67 [hyper,14,5,15] iknows(ep(unique1,a_priv_a))|$ANS(sub(ep(unique1,a_priv_a),i_
tmp1)).
79 [hyper,21,1] same(unique1,a_x).
127 [hyper,23,3,22] inverse(b_pub_a,a_priv_a).
128 [hyper,23,1] same(a_pub_a,b_pub_a).
163 [hyper,24,5,19] iknows(a_pub_a)|$ANS(sub(a_pub_a,i_pub_a)).
231 [hyper,128,5,163] iknows(b_pub_a)|$ANS(sub(b_pub_a,a_pub_a))|$ANS(sub(a_pub_
a,i_pub_a)).244 [hyper,67,6,231,127] $ANS(sub(ep(unique1,a_priv_a),i_tmp1))|iknows(unique1)|
$ANS(sub(unique1,dp(ep(unique1,a_priv_a),b_pub_a)))|$ANS(sub(b_pub_a,a_pub_a))|$
ANS(sub(a_pub_a,i_pub_a)).
255 [hyper,244,5,79] $ANS(sub(ep(unique1,a_priv_a),i_tmp1))|$ANS(sub(unique1,dp(
ep(unique1,a_priv_a),b_pub_a)))|$ANS(sub(b_pub_a,a_pub_a))|$ANS(sub(a_pub_a,i_pu
b_a))|iknows(a_x)|$ANS(sub(a_x,unique1)).
276 [hyper,255,4,18] $ANS(sub(ep(unique1,a_priv_a),i_tmp1))|$ANS(sub(unique1,dp(
ep(unique1,a_priv_a),b_pub_a)))|$ANS(sub(b_pub_a,a_pub_a))|$ANS(sub(a_pub_a,i_pu
b_a))|$ANS(sub(a_x,unique1))|same(i_x,a_x)|$ANS(assign(i_x,a_x)).
277 [binary,276.1,12.1] $ANS(sub(ep(unique1,a_priv_a),i_tmp1))|$ANS(sub(unique1,
dp(ep(unique1,a_priv_a),b_pub_a)))|$ANS(sub(b_pub_a,a_pub_a))|$ANS(sub(a_pub_a,i
_pub_a))|$ANS(sub(a_x,unique1))|$ANS(assign(i_x,a_x)).
------------ end of proof -------------
```

Figure 5.4 : Output of the Automatic Theorem Prover

The first line of the output:

```
----> UNIT CONFLICT at   0.08 sec ----> 277 [binary,276.1,12.1] $ANS(sub(ep(uniq
ue1,a_priv_a),i_tmp1))|$ANS(sub(unique1,dp(ep(unique1,a_priv_a),b_pub_a)))|$ANS(
sub(b_pub_a,a_pub_a))|$ANS(sub(a_pub_a,i_pub_a))|$ANS(sub(a_x,unique1))|$ANS(ass
ign(i_x,a_x)).
```

is a summary of the proof and contains all the information we will need to make modifications to the protocol. Note that the summary contains 6 components:

```
$ANS(sub(ep(unique1,a_priv_a),i_tmp1))
$ANS(sub(unique1,dp(ep(unique1,a_priv_a),b_pub_a)))
$ANS(sub(b_pub_a,a_pub_a))
$ANS(sub(a_pub_a,i_pub_a))
$ANS(sub(a_x,unique1))
$ANS(assign(i_x,a_x))
```

each of which is an answer literal that was produced by use of a particular inference rule during the proof. The first, *$ANS(sub(ep(unique1,a_priv_a),i_tmp1))*, tells us that it was necessary to substitute *i_tmp1* for *ep(unique1,a_priv_a)* at some point in the proof. Likewise, the last, *$ANS(assign(i_x,a_x))*, tells us that the theorem prover discovered that the value *a_x* could be assigned to the variable *i_x*. Although it is not represented in the proof summary, the answer literal:

$ANS(assign(i_x,a_x))

was generated from the inference rule:

-Icontrols(X) | -Iknows(Y) | Same(X,Y) | $ANS(Assign(X,Y)).

once the automatic theorem prover had deduced:

Icontrols(i_x)

and

Iknows(a_x)

In the next section we show how we use this proof summary to make modifications to the protocol.

5.4.5 Using the Proof to Modify the Protocol

Modifying the protocol as dictated by the proof summary is straightforward since the answer literal for each inference rule states explicitly what changes are required. For

example, a *sub(x,y)* predicate indicates that the value *y* must be substituted for the value *x* in any statement that is to be added to the protocol as a result of the proof. A *session(x)* predicate signifies that parallel session fragment *x* should be added to the protocol, and an *assign(x,y)* predicate prescribes that the assignment statement "*x := y;*" should be added. Using these rules we can demonstrate what modifications to the protocol are implied by the proof in Figure 5.4. As mentioned above, the proof summary contains six components:

```
$ANS(sub(ep(unique1,a_priv_a),i_tmp1))
$ANS(sub(unique1,dp(ep(unique1,a_priv_a),b_pub_a)))
$ANS(sub(b_pub_a,a_pub_a))
$ANS(sub(a_pub_a,i_pub_a))
$ANS(sub(a_x,unique1))
$ANS(assign(i_x,a_x))
```

By removing the first clause from the list and performing the substitution it prescribes (*i_tmp1* for *ep(unique1,a_priv_a)*) we get:

```
$ANS(sub(unique1,dp(i_tmp1,b_pub_a)))
$ANS(sub(b_pub_a,a_pub_a))
$ANS(sub(a_pub_a,i_pub_a))
$ANS(sub(a_x,unique1))
$ANS(assign(i_x,a_x))
```

Processing the new head of the list in a similar manner we get:

```
$ANS(sub(b_pub_a,a_pub_a))
$ANS(sub(a_pub_a,i_pub_a))
$ANS(sub(a_x,dp(i_tmp1,b_pub_a)))
$ANS(assign(i_x,a_x))
```

then:

```
$ANS(sub(a_pub_a,i_pub_a))
$ANS(sub(a_x,dp(i_tmp1,a_pub_a)))
$ANS(assign(i_x,a_x))
```

and:

```
$ANS(sub(a_x,dp(i_tmp1,i_pub_a)))
$ANS(assign(i_x,a_x))
```

and finally:

```
$ANS(assign(i_x,dp(i_tmp1,i_pub_a)))
```

The modification prescribed by this line is to add the statement:

I: x := dp[tmp1]pub_A;

to the protocol in place of the weakest precondition that produced the proof. After doing this our list of modifications is empty and the modified protocol is:

(1) *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

(2) *X*: **assume**(*B.pub_A == A.pub_A);*

(3) *X*: **assume**(*I.pub_A == A.pub_A);*

(4) *A*: *x* := *new*;

(5) *A*: $\Rightarrow I$ *(ep[x]priv_A)*;

(6) *I*: $\leftarrow (tmp1)$;

(7) I: x := dp[tmp1]pub_A;

(8) *I*: $\Rightarrow B$ *(tmp1)*;

(9) *B*: $\leftarrow (msg)$;

(10) *B*: *x* := *dp[msg]pub_A*;

Line (7) is the program that was extracted from the proof in Figure 5.4 and has been added to the protocol. We then recompute the weakest preconditions for the modified protocol:

Same(dp[ep[unique1]A.priv_A]A.pub_A,dp[ep[unique1]A.priv_A]A.pub_A)

(1) *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

Same(dp[ep[unique1]A.priv_A]A.pub_A,dp[ep[unique1]A.priv_A]A.pub_A)

(2) *X*: **assume**(*B.pub_A == A.pub_A);*

Same(dp[ep[unique1]A.priv_A]A.pub_A,dp[ep[unique1]A.priv_A]B.pub_A)

(3) *X*: **assume**(*I.pub_A == A.pub_A);*

Same(dp[ep[unique1]A.priv_A]I.pub_A,dp[ep[unique1]A.priv_A]B.pub_A)

(4) *A*: *x* := *new*;

Same(dp[ep[A.x]A.priv_A]I.pub_A,dp[ep[A.x]A.priv_A]B.pub_A)

(5) *A*: $\Rightarrow$ *I* *(ep[x]priv_A)*;

*Same(dp[I.*queue*]I.pub_A,dp[I.*queue*]B.pub_A)*

(6) *I*: $\leftarrow (tmp1)$;

Same(dp[I.tmp1]I.pub_A,dp[I.tmp1]B.pub_A)

(7) *I: x := dp[tmp1]pub_A;*

Same(I.x,dp[I.tmp1]B.pub_A)

(8) *I*: $\Rightarrow$ *B* *(tmp1)*;

*Same(I.x,dp[B.*queue*]B.pub_A)*

(9) *B*: $\leftarrow (msg)$;

Same(I.x,dp[B.msg]B.pub_A)

(10) *B*: *x* := *dp[msg]pub_A*;

Same(I.x,A.x)

Then we start over by trying to prove the weakest precondition for the first statement. In this particular case, we are able to prove the verification condition for the protocol and therefore know that no further modifications are necessary. We can now report the attack scenario to the user.

Line 7 is the attack - a modification to the original protocol that causes the failure condition to be satisfied by the end of the protocol. In this line the intruder decrypts the message she received (line 6) which was encrypted by *A* using her private key. The intruder was able to perform this decryption because we assumed (line 3) that she knew *A*'s public key. In the next section we offer a discussion of the methodology we have presented in this chapter and note some of its strengths and weaknesses.

5.5 Discussion of Our Methodology

As noted in [SIM94], the research community has found it difficult to "either [design] sound information-based protocols or [to prove] that a candidate protocol is sound." One reason for this is that it has never been shown that the cryptographic algorithms upon which many of these protocols are built are truly one-way trapdoor functions. If they are not then clearly protocols that utilize these algorithms are unsound. However, even when we adopt the popular approach of assuming that the underlying cryptographic algorithms are good, Simmons still cautions that "it is extremely difficult to determine whether an information-based protocol is sound, even for very simple protocols." This is due to the same difficulties found in the field of program verification where researchers have not been able to formally define what correctness means for most programs or to show that a formal proof about an abstraction implies the correct functioning of the program's concrete implementation [DLP79].

Researchers have long known that they cannot prove, in general, that a protocol has no flaws but realize that it is still important to try to rid candidate protocols of all the flaws that they can find through analysis. Traditionally, analysis has consisted of the designer (and perhaps others) attempting to find flaws in a protocol using all of the experience and analytical tools at their disposal. However, Simmons points to the large number of protocols that have "survived several rounds of this process, only to have subsequently been shown to have a protocol failure" as evidence of the need for "more systematic and formal [methods] of evaluating protocols."

Our methodology addresses both of Simmons' criteria well. It not only performs an extensive search for attacks but it also allows us to make useful statements about protocols even when no flaws are found. For example, if we assume that the protocol has been translated correctly into CPAL and that a useful failure condition has been chosen then we have

a set of theorems for which we will try to find a constructive proof. If we assume that the theorem prover has been given a complete set of axioms with which to prove the theorems and that the theorem prover performs a breadth-first search for a proof, then we should be able to stop the theorem prover at any point in the search and say that no constructive proof of the given theorem exists that is shorter than the length of the longest proof the theorem prover has considered to that point. Since the proof is constructive, with each inference relating to an action that must be added to the protocol, we can make definite statements about the minimum length of any attack that might satisfy the protocol's failure condition even if we have not yet generated such an attack. More importantly, if the theorem prover finds a constructive proof of one of the theorems we can make an even stronger statement, namely, that there is an attack (which we can exhibit) that achieves the failure condition of the protocol.

The implementation of our methodology that we have described in this chapter does not attempt to satisfy many of the assumptions stated in the preceding paragraph and therefore cannot make statements as strong as those given above when the system is stopped before it has found a constructive proof of a theorem. This was done purposely to make the system find the "standard" types of attacks faster. For instance, the automatic theorem prover that we use does not perform a breadth-first search when attempting to prove a theorem. Instead it uses a number of heuristics to direct the search which means that many longer inference paths could be examined before a shorter (and potentially successful) path is found. This makes it much harder to characterize the set of proofs that have not been examined at any point in the search, but, in general, the heuristics the automatic theorem prover uses enables it to find proofs much faster than a breadth-first theorem prover would.

Another choice we have made in our implementation to decrease the amount of time it takes to find proofs is to use only a selected set of axioms. This decision means that there may be constructive proofs for a theorem that the theorem prover will not be able to find

because it lacks some of the necessary axioms. For example, we do not currently include any algebraic or number theoretic axioms so it is not possible for the system to discover attacks which depend on algebraic properties of the underlying cryptosystems. We chose to exclude such axioms for now because there are very few examples of such attacks and the inclusion of these axioms would have caused the system to take considerably longer to find the more common attacks. In the Future Work section we discuss how additional axioms could be added to our system thereby enabling it to discover such attacks. In the next chapter we demonstrate how our system can automatically find many of the well-known attacks on cryptographic protocols from the literature.

5.6 Summary

In this chapter we have presented a three-step methodology for examining cryptographic protocols for flaws. First, the user expresses the protocol(s) in CPAL. Next, the user specifies the failure conditions for the protocol(s). Third, a deductive automatic programming system is employed to try to modify the protocol(s) so that a failure condition is achieved. If this last step succeeds we have identified an attack that subverts the goals of a protocol and we can demonstrate the attack to the user.

The main strength of this approach is that the formal semantics of protocols are exploited in a number of ways to make the analysis comprehensive, automatic, and tractable. The semantics of CPAL allow us to formally specify a failure condition for a protocol and then employ a very powerful automatic programming system to search for an attack that achieves the failure condition. The search for an attack is not user-directed but instead is guided by the formal semantic definition of failure derived from the protocol and the user-defined failure condition.

Another contribution of this approach is that it can discover attacks that arise due to

the interaction of two or more cryptographic protocols running in the same environment. This represents a new class of attack that we call *protocol-interaction attacks.* Considering all possible interactions among all of the protocols that may be used together is clearly beyond the capability of any human inspector, and since these types of attacks are also not currently handled by any other cryptographic protocol analysis tool we see an opportunity for our method to complement many existing cryptographic protocol evaluation techniques.

In the next chapter we demonstrate our method by presenting the results of our system's analysis of a number of cryptographic protocols from the literature. We show that our system can find well-known "standard" attacks as well as the new protocol-interaction attacks.

Chapter 6

Protocols Analyzed Using CPAL-ES

In this chapter we present the results of our analysis of a number of well-known cryptographic protocols. We demonstrate our methodology's ability to discover many of the known attacks on these protocols and then turn our attention to some of the new types of attacks that we have uncovered. We close this chapter by characterizing the class of attacks that our system can currently discover and discuss how this class could be expanded to include additional ones.

6.1 Finding an Attack on a Simple Cryptographic Protocol

We begin this chapter by working a detailed example that shows all the steps taken by our system to discover a flaw in a simple cryptographic protocol. The protocol (with line

numbers added for reference) is given in CPAL below.

(1) A: $k := new$;

(2) A: $\rightarrow B$ (k);

(3) B: $\leftarrow (k)$;

(4) A: $x := new$;

(5) A: $\rightarrow B$ $(e[x]k)$;

(6) B: $\leftarrow (msg)$;

(7) B: $x := d[msg]k$;

Figure 6.1 : A Simple Cryptographic Protocol

The protocol is intended to transfer a value, *x*, from user *A* to user *B* while keeping it secret from other users on the network. To accomplish this, agent *A* creates a new key, *k*, and sends it to *B* (lines 1 and 2). Principal *B* receives this key from *A* and stores it in the variable *k* in his address space (line 3). Agent *A* then generates the piece of secret data, *x*, and sends it to *B* encrypted under the key, *k*, using a symmetric cryptosystem (lines 4 and 5). After principal *B* receives this message from *A*, he can decrypt it and learn the value, *x*, that *A* has sent him. Of course, since both the key and the encrypted message are sent insecurely, an eavesdropper could observe these two messages and use the first to decrypt the second and learn *x*.

After specifying the protocol in CPAL (Figure 6.1), the next step in our methodology is to establish a statement of failure for the protocol. For this protocol we will use the predicate:

Same(I.x, A.x)[1]

---

1. We could also use the predicate *Same(I.x,B.x)* as our failure condition and although some of the details that follow would be slightly different the final result of our system's analysis would be identical.

This predicate is true when the value stored in the variable *x* in the intruder's address space is equivalent to the value stored in the variable *x* in agent *A*'s address space. Since the protocol is intended to transmit the value *x* from *A* to *B* privately, we are saying that the protocol has failed if that value also somehow appears in the intruder's address space (since that implies that *x* is no longer known to only *A* and *B*). The input to the CPAL-ES system is the protocol and the failure condition as shown in Figure 6.2 below.

(1) A: $k := new$;

(2) A: $\rightarrow B\ (k)$;

(3) B: $\leftarrow (k)$;

(4) A: $x := new$;

(5) A: $\rightarrow B\ (e[x]k)$;

(6) B: $\leftarrow (msg)$;

(7) B: $x := d[msg]k$;

Same(I.x,A.x)

Figure 6.2 : Input to the CPAL-ES System

From this point on, the system operates without any further interaction with the user. First, each insecure send in the protocol is replaced with a secure send to the intruder and the weakest precondition of the protocol and the failure condition is computed. The result is shown in Figure 6.3.

Same(I.x,unique2)

(1) A: $k := new$;

Same(I.x,unique2)

(2) A: $\Rightarrow I\ (k)$;

Same(I.x,unique2)

(3) I: $\leftarrow (tmp1)$;

Same(I.x,unique2)

(4) I: $\Rightarrow B\ (tmp1)$;

Same(I.x,unique2)

(5) B: $\leftarrow (k)$;

Same(I.x,unique2)

(6) A: $x := new$;

Same(I.x,A.x)

(7) A: $\Rightarrow I\ (e[x]k)$;

Same(I.x,A.x)

(8) I: $\leftarrow (tmp2)$;

Same(I.x,A.x)

(9) I: $\Rightarrow B\ (tmp2)$;

Same(I.x,A.x)

(10) B: $\leftarrow (msg)$;

Same(I.x,A.x)

(11) B: $x := d[msg]k$;

Same(I.x,A.x)

Figure 6.3 : Weakest Preconditions for the Simple Protocol

Next, the system attempts to find a constructive proof for one of the weakest preconditions. It starts with the weakest precondition for the first statement of the protocol and tries to prove it using an automatic theorem prover. The theorem prover will either succeed in proving the theorem or give up after some user-specified time limit has been

reached (one minute in this example). If no proof for the weakest precondition for the first statement is found in the allotted time, the system moves on to the next weakest precondition and attempts to prove it. For the protocol in Figure 6.3, the system fails to prove any of the weakest preconditions for statements one through eight, however, it does succeed for statement nine. The automatic theorem prover's input file for statement nine (which was automatically generated from the protocol as described in Chapter 5) is given in Figure 6.4. For simplicity, the parallel session axioms have been omitted from Figure 6.4 since there are 28 of them and none of them are used in the proof of the theorem.

```
set(auto).
set(prolog_style_variables).
assign(max_seconds,60).

list(usable).

% =================== Standard Axioms ===================

same(X,X).
-same(X,Y) | same(Y,X).
-same(X,Y) | -same(Y,Z) | same(X,Z).
-inverse(K1,K2) | inverse(K2,K1).
-same(K1,K2) | -inverse(K2,K3) | inverse(K1,K3).
-icontrols(X) | -iknows(Y) | same(X,Y) | $ANS(assign(X,Y)).
-iknows(X) | -same(X,Y) | iknows(Y) | $ANS(sub(Y,X)).
-same(A,dot(B,C)) | -same(B,D) | same(A,dot(D,C)) | $ANS(sub(B,D)).
-iknows(X) | iknows(dot(X,Y)).
-same(A,B) | same(dot(A,C),dot(B,C)).
dot($nil,X) = $nil.
dot([X|Y],1) = X.
dot([X|Y],2) = dot(Y,1).
dot([X|Y],3) = dot(Y,2).
dot([X|Y],4) = dot(Y,3).
-iknows(X) | -same(dot(X,Y),Z) | iknows(Z).

% =================== Private-Key Axioms ===================

-iknows(K) | -iknows(e(X,K)) | iknows(X) | $ANS(sub(X,d(e(X,K),K))).
-iknows(K) | -iknows(d(X,K)) | iknows(X) | $ANS(sub(X,e(d(X,K),K))).
-same(K1,K2) | same(d(e(X,K1),K2),X).
-same(K1,K2) | same(e(d(X,K1),K2),X).
-same(X,Y) | -iknows(e(X,K)) | iknows(e(Y,K)) | $ANS(sub(X,Y)).
-same(X,Y) | -iknows(d(X,K)) | iknows(d(Y,K)) | $ANS(sub(X,Y)).
-icontrols(X) | -iknows(Y) | -iknows(d(X,K)) | iknows(d(Y,K)) | $ANS(sub(X,Y)).
-same(A,B) | same(d(A,C),d(B,C)).
-same(A,B) | same(e(A,C),e(B,C)).
```

```
-icontrols(X) | -iknows(K) | -iknows(Y) | -same(Z,d(X,K)) | same(Y,d(X,K)) |
$ANS(assign(X,e(Y,K))).
-icontrols(X) | -iknows(K) | -iknows(Y) | -same(Z,e(X,K)) | same(Y,e(X,K)) |
$ANS(assign(X,d(Y,K))).

% ==================== Protocol-Specific Axioms ====================

iknows(i_tmp2).
icontrols(i_tmp2).
same(i_tmp2,e(unique2, unique1)).
iknows(i_tmp1).
icontrols(i_tmp1).
same(i_tmp1,unique1).
iknows(i_x).
icontrols(i_x).
same(i_x,i_x).
same(b_k,i_tmp1).
same(a_k,unique1).
same(a_x,unique2).

end_of_list.

% ================ Negation of the Weakest Precondition ================

formula_list(usable).

-same(i_x, a_x).

end_of_list.
```

Figure 6.4 : Input File For Automatic Theorem Prover

The proof that the automatic theorem prover generates is shown in Figure 6.5.

```
----> UNIT CONFLICT at   2.41 sec ----> 1677 [binary,1676.1,22.1] $ANS(sub(e(unique2,
unique1),i_tmp2)) | $ANS(sub(unique2,d(e(unique2,unique1),unique1))) | $ANS(sub(uniq
ue1,i_tmp1)) | $ANS(sub(a_x,unique2)) | $ANS(assign(i_x,a_x)).

Length of proof is 6.  Level of proof is 4.

---------------- PROOF ----------------

1 [] -same(A,B) | same(B,A).
5 [] -icontrols(A) |  -iknows(B) | same(A,B) | $ANS(assign(A,B)).
6 [] -iknows(A) |  -same(A,B) | iknows(B) | $ANS(sub(B,A)).
11 [] -iknows(A) |  -iknows(e(B,A)) | iknows(B) | $ANS(sub(B,d(e(B,A),A))).
22 [] -same(i_x,a_x).
34 [] iknows(i_tmp2).
36 [] same(i_tmp2,e(unique2,unique1)).
37 [] iknows(i_tmp1).
39 [] same(i_tmp1,unique1).
41 [] icontrols(i_x).
```

```
44 [] same(a_x,unique2).
120 [hyper,39,6,37] iknows(unique1) | $ANS(sub(unique1,i_tmp1)).
192 [hyper,44,1] same(unique2,a_x).
335 [hyper,36,6,34] iknows(e(unique2,unique1)) | $ANS(sub(e(unique2,unique1),i_tmp2)).
1345 [hyper,335,11,120] $ANS(sub(e(unique2,unique1),i_tmp2)) | iknows(unique2)
| $ANS(sub(unique2,d(e(unique2,unique1),unique1))) | $ANS(sub(unique1,i_tmp1)).
1498 [hyper,1345,6,192] $ANS(sub(e(unique2,unique1),i_tmp2)) | $ANS(sub(unique2,
d(e(unique2,unique1),unique1))) | $ANS(sub(unique1,i_tmp1)) | iknows(a_x) | $ANS(sub(a
_x,unique2)).
1676 [hyper,1498,5,41] $ANS(sub(e(unique2,unique1),i_tmp2)) | $ANS(sub(unique2,
d(e(unique2,unique1),unique1))) | $ANS(sub(unique1,i_tmp1)) | $ANS(sub(a_x,unique2))
| same(i_x,a_x) | $ANS(assign(i_x,a_x)).
1677 [binary,1676.1,22.1] $ANS(sub(e(unique2,unique1),i_tmp2)) | $ANS(sub(unique2,
d(e(unique2,unique1),unique1))) | $ANS(sub(unique1,i_tmp1)) | $ANS(sub(a_x,unique2))
| $ANS(assign(i_x,a_x)).

------------ end of proof -------------
```

Figure 6.5 : The Proof Generated by the Theorem Prover

The system then uses this proof to make modifications to the protocol. By parsing the summary line:

```
----> UNIT CONFLICT at   2.41 sec ----> 1677 [binary,1676.1,22.1] $ANS(sub(e(unique2,
unique1),i_tmp2)) | $ANS(sub(unique2,d(e(unique2,unique1),unique1))) | $ANS(sub(uniq
ue1,i_tmp1)) | $ANS(sub(a_x,unique2)) | $ANS(assign(i_x,a_x)).
```

the system discovers that there are five modifications that need to be made to the protocol. These modifications correspond to the answer literals that were generated during the proof:

```
sub(e(unique2, unique1),i_tmp2)
sub(unique2,d(e(unique2,unique1),unique1))
sub(unique1,i_tmp1)
sub(a_x,unique2)
assign(i_x,a_x)
```

The system removes the first clause from this list and performs the substitution it prescribes (*i_tmp2* for *e(unique2,unique1)*) on the remaining clauses. This yields:

```
sub(unique2,d(i_tmp2,unique1))
sub(unique1,i_tmp1)
sub(a_x,unique2)
assign(i_x,a_x)
```

Processing the new head of the list in a similar manner we get:

```
sub(unique1,i_tmp1)
sub(a_x,d(i_tmp2,unique1))
assign(i_x,a_x)
```

then:

```
sub(a_x,d(i_tmp2,i_tmp1))
assign(i_x,a_x)
```

and finally:

```
assign(i_x,d(i_tmp2,i_tmp1))
```

This clause can be translated into the following CPAL statement:

I: x := d[tmp2]tmp1;

which is then added to the protocol in place of the weakest precondition that was proven. The result is the following[1]:

(1) *A*: $k := new$;

(2) *A*: $\Rightarrow I$ *(k);*

(3) *I:* $\leftarrow (tmp1)$;

(4) *I*: $\Rightarrow B$ *(tmp1);*

(5) *B:* $\leftarrow (k)$;

(6) *A*: $x := new$;

(7) *A*: $\Rightarrow I$ *(e[x]k);*

(8) *I:* $\leftarrow (tmp2)$;

*** (9) *I: x := d[tmp2]tmp1;***

(10) *I*: $\Rightarrow B$ *(tmp2);*

(11) *B:* $\leftarrow (msg)$;

(12) *B: x := d[msg]k;*

Figure 6.6 : The Modified Protocol

The system then recomputes the weakest preconditions for the protocol:

1. Asterisks are used to denote lines of a protocol that were inserted or modified by our system.

Same(d[e[unique2]unique1]unique1,unique2)

(1) *A*: *k* := *new*;

Same(d[e[unique2]A.k]A.k,unique2)

(2) *A*: ⇒ *I* (*k*);

*Same(d[e[unique2]A.k]I.*queue*,unique2)*

(3) *I*: ← (*tmp*1) ;

Same(d[e[unique2]A.k]I.tmp1,unique2)

(4) *I*: ⇒ *B* (*tmp1*);

Same(d[e[unique2]A.k]I.tmp1,unique2)

(5) *B*: ← (*k*) ;

Same(d[e[unique2]A.k]I.tmp1,unique2)

(6) *A*: *x* := *new*;

Same(d[e[A.x]A.k]I.tmp1,A.x)

(7) *A*: ⇒ *I* (*e[x]k*);

*Same(d[I.*queue*]I.tmp1,A.x)*

(8) *I*: ← (*tmp*2) ;

Same(d[I.tmp2]I.tmp1,A.x)

(9) *I: x := d[tmp2]tmp1;*

Same(I.x,A.x)

(10) *I*: ⇒ *B* (*tmp2*);

Same(I.x,A.x)

(11) *B*: ← (*msg*) ;

Same(I.x,A.x)

(12) *B: x := d[msg]k;*

Same(I.x,A.x)

Figure 6.7 : The Modified Protocol with Weakest Preconditions

The system then starts over by trying to prove the weakest precondition for the first statement. In this case, the proof of the first weakest precondition succeeds since:

Same(d[e[unique2]unique1]unique1,unique2)

can be simplified to:

Same(unique2,unique2)

and then to:

TRUE

No modifications to the protocol need to be made as a result of this proof so the system reports the compromised protocol (Figure 6.6) to the user.

6.2 Published Protocols Analyzed Using CPAL-ES

While cryptographic protocols have been proposed for such things as electronic voting, oblivious funds transfers, and threshold secret release, by far the majority of published cryptographic protocols deal with authentication and key-exchange. These two classes have also served as the focus of research in cryptographic protocol analysis with specialized logics being designed just to examine authentication protocols [BAN89] and elaborate tools being designed just to examine key-exchange protocols [MEA91]. We have chosen to concentrate on these two types of protocols in this section, both because of their preponderance and because they represent the majority of the types of cryptographic protocols on which attacks have been published.

To give some structure to our discussion of the numerous protocols that will be presented in this section, we will further subdivide protocols based on the types of known attacks to which they are susceptible. Table 6.1 gives a rough classification of protocols based on their type (authentication or key-exchange) and their known weaknesses (single-session, multi-session, or key-compromise).

Attack Type

Protocol Type

	Single-Session	Multi-Session	Key-Compromise
Authentication	CCITT X.509	ISO	Needham and Schroeder
Key-Exchange	Neuman and Stubblebine TMN	Denning and Sacco	Beller et al.

Table 6.1: A Rough Classification of Protocols

A *single-session* attack uses only a single run of the protocol whereas a *multi-session* attack requires that two or more instances of the protocol be run. The attack that we demonstrated for the protocol in Section 6.1 is a single-session attack.

A good example of a multi-session attack is the following. Imagine that there is a room with a door that is being protected by a guard. The guard will only admit people who know the correct password. When an individual approaches the door, the guard challenges her to say the password, and if she answers correctly the guard will allow her to enter. This describes a protocol for controlling access to the room. One attack on this protocol is for an intruder to wait (within earshot) for an authorized user to approach the door and then eavesdrop as he tells the guard the correct password. The intruder can then walk up to the guard, repeat the password, and enter the room. This is an example of a multi-session attack since the protocol must be run at least twice (once for the intruder to learn the password and once for the intruder to use the password) for the attack to succeed. Some of the multi-session attacks we will demonstrate later in this section will be much more complex with different sessions of the protocol being run between different principals either simultaneously or in an interleaved manner.

Key-compromise attacks are somewhat unique to cryptographic protocols and are based on the assumption that certain cryptographic keys are “weaker” than others and more susceptible to compromise. In fact, this is the justification behind many key-distribution protocols since they assume that principals already share some secret and want to use it to establish a new cryptographic key to protect a single “conversation”. One obvious question is why don’t agents just use the existing shared secret to communicate privately rather than going through some protocol to establish a conversation key? The answer is that if all communications between two principals is encrypted with the same key, then if that key is broken, an attacker can read every message that the two have ever exchanged. But if each conversation is encrypted with a different key then breaking a single key only reveals the contents of a single conversation. In addition, cryptanalysis usually becomes easier as the amount of ciphertext available for a given cryptographic key increases. So session keys also make the cryptanalyst’s job harder by limiting the amount of ciphertext available for any one key (especially the long-held shared secrets).

So while agreeing on and using session keys is an important goal or subgoal of many cryptographic protocols, our analysis of these protocols must take into account the fact that these conversation keys may be much more likely to be broken (especially over time) than many of the other cryptographic keys used in these protocols. A *key-compromise* attack is one in which the breaking of a conversation key (usually long after the conversation in which it was used has ended) allows an intruder not only to read messages from that old conversation, but to use some of the old messages and knowledge of the compromised key to attack current and future conversations.

As noted above there are types of protocols and types of known attacks besides those given in Table 6.1, but we feel that our classification covers the majority of both protocols and known attack types. We now turn our attention to the protocols given in Table 6.1. For each we will describe the protocol and its goals, the results of our system’s analysis,

and a description of the weaknesses our system uncovers. All of the protocols presented in this chapter were thought to be correct at the time they were published and were subsequently shown to suffer from at least one weakness from Table 6.1. With the exception of one protocol (as noted) all of the flaws we will demonstrate in this section had been discovered before we ran our system on the protocols, but our system was able to discover each flaw independently.

We begin our discussion with a protocol proposed by Tatebayashi, Matsuzaki, and Newman (TMN) that was used as a common test protocol in a paper [KMM94] that compared three independently developed systems for cryptographic protocol analysis. In this paper, each tool (Millen's Interrogator, Meadows' NRL Protocol Analyzer, and Kemmerer's Inatest system) was used to examine the TMN protocol to illustrate the differences among these approaches. All three systems were able to find the simple single-session attack that is illustrated in the next section, but only Kemmerer's method was able to discover the more complex attack on this protocol given in Section 2.1.2.

6.2.1 The TMN Protocol

As shown in Table 6.1, TMN is a key-exchange protocol that is vulnerable to a single-session attack by an intruder. The protocol assumes that there are a number of mobile "ground stations" that all know the public key of a central trusted server. When two ground stations want to communicate securely with each other, they establish a session key for the conversation with the help of the server. The following description of the TMN protocol is taken from [TMN91]:

- When user A wishes to communicate with user B, it encrypts a random number with the server's public key, and sends the encrypted random number, along with A's and B's names.
- When the server receives the request, it decrypts the random number and stores it

as a key-encryption key for that conversation; it also notifies *B* that *A* wishes to speak to it.

- User *B*, on receiving the notification from the server, generates a random number to be used as a session key, encrypts it with the server's public key, and sends it to the server.
- The server decrypts the response, encrypts the key with *A*'s random number using a private-key algorithm, and sends the result to *A*.
- User *A* decrypts the message from the server using the original random number it had generated and assumes that the result is the session key.

It is assumed that all ground stations can generate random numbers and know the public key of the server. We now present the TMN protocol in CPAL. We let *A* represent user *A*, *B* represent user *B*, *S* represent the server, and *I* represent the intruder (a dishonest ground station):

```
(1)  X: assume(Inverse(S.priv_S,S.pub_S));
(2)  X: assume(A.pub_S == S.pub_S);
(3)  X: assume(B.pub_S == S.pub_S);
(4)  X: assume(I.pub_S == S.pub_S);
(5)  A: r1 := new;
(6)  A:  → S  (<A,B,ep[r1]pub_S>);
(7)  S:  ← (msg1) ;
(8)  S: ka := dp[msg1.3]priv_S;
(9)  S:  → B  (msg1.1);
(10) B:  ← (msg2) ;
(11) B: r2 := new;
(12) B:  → S  (ep[r2]pub_S);
(13) S:  ← (msg3) ;
(14) S: kb := dp[msg3]priv_S;
(15) S:  → A  (e[kb]ka);
(16) A:  ← (msg4) ;
(17) A: ck := d[msg4]r1;
```

Figure 6.8 : The TMN Protocol

Since this is a key-exchange protocol we take the following as its failure condition:

Same(I.k, A.ck)

As in the example from Section 6.1, this predicate states that the value that agent *A* believes is the key for a private conversation with agent *B* appears in the intruder's address space. If

we give the TMN protocol (as specified in Figure 6.8) and the above failure condition to the CPAL-ES system it produces the following output:

(1) *X: assume(Inverse(S.priv_S,S.pub_S));*

(2) *X: assume(A.pub_S == S.pub_S);*

(3) *X: assume(B.pub_S == S.pub_S);*

(4) *X: assume(I.pub_S == S.pub_S);*

(5) *A: r1 := new;*

(6) *A:* $\Rightarrow I$ (<A,B,ep[r1]pub_S>);

(7) *I:* $\leftarrow (tmp1)$;

(8) *I:* $\Rightarrow S$ (*tmp1*);

(9) *S:* $\leftarrow (msg1)$;

(10) *S: ka := dp[msg1.3]priv_S;*

(11) *S:* $\Rightarrow I$ (*msg1.1*);

(12) *I:* $\leftarrow (tmp2)$;

(13) *I:* $\Rightarrow B$ (*tmp2*);

(14) *B:* $\leftarrow (msg2)$;

(15) *B: r2 := new;*

(16) *B:* $\Rightarrow I$ *(ep[r2]pub_S);*

(17) *I:* $\leftarrow (tmp3)$;

*** (18) *I:*** $\Rightarrow S$ **(*ep[k]pub_S*);**

(19) *S:* $\leftarrow (msg3)$;

(20) *S: kb := dp[msg3]priv_S;*

(21) *S:* $\Rightarrow I$ *(e[kb]ka);*

(22) *I:* $\leftarrow (tmp4)$;

(23) *I:* $\Rightarrow A$ (*tmp4*);

(24) *A:* $\leftarrow (msg4)$;

(25) *A: ck := d[msg4]r1;*

Figure 6.9 : An Attack on the TMN Protocol

The attack occurs in line (18) of Figure 6.9. Instead of allowing *B*'s message (which contains a proposed session key encrypted under *S*'s public key) to reach *S*, the intruder substitutes a message he creates that contains the encryption of a value that he knows under the server's public key. The intruder can create this message because we assume (in line 4) that he knows *S*'s public key. Once *B*'s message to *S* has been replaced in this manner, the

rest of the protocol executes as usual - the server will receive the intruder's message (thinking it came from *B*), decrypt it with his private key, encrypt the proposed conversation key with *A*'s random number, and send the result to *A*. Upon receiving this message from the server, *A* will decrypt it using her random number and accept the result as the conversation key to talk privately with *B*. However, all messages sent out by *A* encrypted under the agreed-upon conversation key will be readable by the intruder but not by *B*. Note that this is a single-session attack since it does not rely on multiple runs of the protocol.

This attack was first illustrated by Simmons and was duplicated by the three systems compared in [KMM94]. Kemmerer's approach also discovered a different, more complex multi-session attack (also previously reported by Simmons) that depends on several mathematical properties of the particular symmetric-key cryptosystem that is used (see Section 2.1.2). As mentioned in Chapter 5, our system cannot currently discover attacks based on properties of the underlying cryptosystems, but it could be extended to do so in the future (see Chapter 7). As stated in [KMM94], "Success on this example does not prove the power of the ... systems, [and] this example is simply a vehicle for illuminating such issues as specification style, user interaction, and differences with respect to the kind of informal analysis done by an expert such as Simmons."

6.2.2 The Neuman and Stubblebine Protocol

Another key-exchange protocol that is susceptible to a single-session attack is that of Neuman and Stubblebine [NS93]. The protocol seeks to help users *A* and *B* agree on a session key with the help of an authentication server, *S*, with whom they share the secret keys, *Kas* and *Kbs*, respectively:

- Principal *A* begins the protocol by sending a message to *B* containing her name and a nonce
- Upon receipt of this message *B* creates his own nonce and sends a message to the

authentication server containing *B*'s name, *B*'s nonce, and the encryption under *Kbs* of *A*'s name, *A*'s nonce, and a timestamp from *B*.

- The authentication server receives *B*'s message, decrypts the part encrypted under *Kbs*, and chooses *Kab*, a new conversation key for *A* and *B*. The authentication server then sends a message to *A* containing a "ticket" for *A*, a "ticket" for *B*, and *B*'s nonce. *A*'s ticket contains *A*'s name, *A*'s nonce, the newly created conversation key, and *B*'s nonce, all encrypted under *Kas*. *B*'s ticket contains *A*'s name, the conversation key, and *B*'s nonce, all encrypted under *Kbs*.
- Agent A can decrypt his ticket and check the nonce to ensure the ticket was generated by the authentication server during the current run of the protocol. If the ticket is fresh then *A* also accepts the conversation key in it and forwards *B*'s ticket to *B* along with *B*'s nonce encrypted using the conversation key.
- Upon receipt of this message from *A*, *B* can decrypt his ticket, check its freshness in the same manner that *A* checked the freshness of her ticket, and extract the session key from the ticket. *B* can then decrypt the second part of *A*'s message and check to see that it matches his own nonce.

In Figure 6.10 we present the Neuman and Stubblebine protocol in CPAL. *S* is the authentication server, *A* is user *A*, *B* is user *B*, *I* is the intruder, *na* is *A*'s nonce, *nb* is *B*'s

nonce, *tb* is *B*'s timestamp, *kas* is the shared key between *A* and *S*, *kbs* is the shared key between *B* and *S*, and *kab* is *S*'s proposed conversation key for *A* and *B*.

(1) *X: assume(A.kas == S.kas);*

(2) *X: assume(B.kbs == S.kbs);*

(3) *A: na := new;*

(4) *A:* $\rightarrow B$ *(<A, na>);*

(5) *B:* $\leftarrow (msg1)$;

(6) *B: nb := new;*

(7) *B:* $\rightarrow S$ *(<B, e[<msg1.1, msg1.2, tb>]kbs, nb>);*

(8) *S:* $\leftarrow (msg2)$;

(9) *S: msg3 := d[msg2.2]kbs;*

(10) *S: kab := new;*

(11) *S:* $\rightarrow A$ *(<e[<msg2.1, msg3.2, kab, msg3.3>]kas, e[<msg3.1, kab, msg3.3>]kbs,msg2.3>);*

(12) *A:* $\leftarrow (msg4)$ *;*

(13) *A: msg5 := d[msg4.1]kas;*

(14) *A: assert(A.msg5.2 == A.na);*

(15) *A: kab := msg5.3;*

(16) *A:* $\rightarrow B$ *(<msg4.2, e[msg5.2]kab>);*

(17) *B:* $\leftarrow (msg6)$

(18) *B: msg7 := d[msg6.1]kbs;*

(19) *B: assert(B.msg7.1 == B.msg1.1);*

(20) *B: kab := msg7.2;*

(21) *B: nb1 := d[msg6.2]kab;*

(22) *B: assert(B.nb1 == B.nb);*

Figure 6.10 : The Neuman and Stubblebine Protocol

The statement of failure is very similar to that of the TMN protocol:

Same(I.k, B.kab)

That is, the intruder knows what *B* believes to be the conversation key. The results of our system's analysis of this protocol is given below.

```
    (1)  X: assume(A.kas == S.kas);
    (2)  X: assume(B.kbs == S.kbs);
    (3)  A: na := new;
    (4)  A: ⇒ I (<A, na>);
    (5)  I: ← (tmp1);
    (6)  I: ⇒ B (tmp1);
    (7)  B: ← (msg1);
    (8)  B: nb := new;
    (9)  B: ⇒ I (<B, e[<msg1.1, msg1.2, tb>]kbs, nb>);
   (10)  I: ← (tmp2);
   (11)  I: ⇒ S (tmp2);
   (12)  S: ← (msg2);
   (13)  S: msg3 := d[msg2.2]kbs;
   (14)  S: kab := new;
   (15)  S: ⇒ I (<e[<msg2.1, msg3.2, kab, msg3.3>]kas, e[<msg3.1, kab, msg3.3>]kbs,msg2.3>);
   (16)  I: ← (tmp3);
   (17)  I: ⇒ A (tmp3);
   (18)  A: ← (msg4);
   (19)  A: msg5 := d[msg4.1]kas;
   (20)  A: assert(A.msg5.2 == A.na);
   (21)  A: kab := msg5.3;
   (22)  A: ⇒ I (<msg4.2, e[msg5.2]kab>);
   (23)  I: ← (tmp4);
 * (24)  I: k := tmp1.2;
 * (25)  I: ⇒ B (<tmp2.2, e[tmp2.3]tmp1.2>);
   (26)  B: ← (msg6);
   (27)  B: msg7 := d[msg6.1]kbs;
   (28)  B: assert(B.msg7.1 == B.msg1.1);
   (29)  B: kab := msg7.2;
   (30)  B: nb1 := d[msg6.2]kab;
   (31)  B: assert(B.nb1 == B.nb);
```

This attack is based on the similarity between the first message *B* sends out (line 9) and *B*'s ticket (generated by *S* in line 15 and sent to *B* in line 25). These messages can be represented as:

Message 1: e[A, na, tb]kbs

Message 2: e[A, kab, tb] kbs

In particular, note that both messages are composed of three values encrypted under *kbs*. In fact, the only difference between these two messages is in the second field, which is *A*'s nonce in one and the session key in the other. Since the protocol does not state that keys and nonces can be distinguished from one another, the intruder can substitute the first of these messages when *B* is expecting the second and get *B* to believe that *A*'s nonce is the session key. This is all that is needed to subvert the protocol since *B* does not check the conversation key in any way and since *A*'s nonce is passed in the clear and therefore known to the intruder. This is an example of a single-session attack and was first reported by Syverson in [SYV93].

6.2.3 The CCITT X.509 Protocol

The X.509 family of protocols proposed by the Consultative Committee for International Telegraph and Telephony (CCITT) are intended for signed, secure communication between two principals, assuming that each knows the public key of the other. While these protocols are designed to protect both the integrity and privacy of messages, the attack that we will demonstrate in this section exploits a flaw in the authenticity of a message and we have included the CCITT X.509 protocol in the row of authentication protocols in Table 6.1 for that reason.

For our analysis we consider the "one-message" protocol in which agent *A* sends a message to agent *B* containing *A*'s name and list of values signed with *A*'s private key. In pseudocode notation *A*'s message to *B* would be represented as:

$$A, \{Ta, Na, B, Xa, \{Ya\}Kb\}Ka^{-1}$$

Here *Ta* is a timestamp generated by *A*, *Na* is a nonce generated by *A*, and *Xa* and *Ya* are user data. The protocol is intended to ensure the integrity of *Xa* and *Ya*, assure the recipient of their origin, and guarantee the privacy of *Ya*. The CPAL representation of the one-message CCITT X.509 protocol is given in Figure 6.11 below.

```
(1)  X: assume(Inverse(A.pub_A, A.priv_A));
(2)  X: assume(Inverse(B.pub_B, B.priv_B));
(3)  X: assume(Inverse(I.pub_I, I.priv_I));
(4)  X: assume(B.pub_A == A.pub_A);
(5)  X: assume(I.pub_A == A.pub_A);
(6)  X: assume(A.pub_B == B.pub_B);
(7)  X: assume(I.pub_B == B.pub_B);
(8)  X: assume(A.pub_I == I.pub_I);
(9)  X: assume(B.pub_I == I.pub_I);
(10) A:  → B (<A, e[<Ta, Na, B, Xa, e[Ya]pub_B>]priv_A>);
(11) B:  ← (msg1) ;
(12) B: msg2 := msg1.1;
(13) B: if (Same(msg2, A)) then {msg3 := d[msg1.2]pub_A};
(14) B: if (Same(msg2, I)) then {msg3 := d[msg1.2]pub_I};
(15) B: msg4 := d[msg3.5]priv_B;
(16) B: assert(Same(B.msg4, A.Ya));
```

Figure 6.11 : The CCITT X.509 "One-Message" Protocol

For the failure condition we take:

Same(B.msg2, I)

which implies that *B* has verified *I*'s signature on the message and therefore believes that both *Xa* and *{Ya}Kb* came from the intruder. The result of the our system's analysis of this protocol and failure condition are given in Figure 6.12.

```
(1)    X: assume(Inverse(A.pub_A, A.priv_A));
(2)    X: assume(Inverse(B.pub_B, B.priv_B));
(3)    X: assume(Inverse(I.pub_I, I.priv_I));
(4)    X: assume(B.pub_A == A.pub_A);
(5)    X: assume(I.pub_A == A.pub_A);
(6)    X: assume(A.pub_B == B.pub_B);
(7)    X: assume(I.pub_B == B.pub_B);
(8)    X: assume(A.pub_I == I.pub_I);
(9)    X: assume(B.pub_I == I.pub_I);
(10)   A:  ⇒ I (<A, e[<Ta, Na, B, Xa, e[Ya]pub_B>]priv_A>);
(11)   I:  ← (tmp1) ;
* (12) I: tmp2 := e[d[tmp1.2]pub_A]priv_I;
* (13) I:  ⇒ B (<I, tmp2>);
(14)   B:  ← (msg1) ;
(15)   B: msg2 := msg1.1;
(16)   B: if (Same(msg2, A)) then {msg3 := d[msg1.2]pub_A};
(17)   B: if (Same(msg2, I)) then {msg3 := d[msg1.2]pub_I};
(18)   B: msg4 := d[msg3.5]priv_B;
(19)   B: assert(Same(B.msg4, A.Ya));
```

Figure 6.12 : An Attack on the CCITT X.509 "Single-Message" Protocol

The attack is as follows. In line 11 above, the intruder receives the message that *A* has signed and sent to *B*. Since the intruder knows A's public key he can remove her signature from the message and sign the results with his own private key (line 12). Note that the intruder cannot decrypt *{Ya}Kb* to learn *Ya*, but that this value will occur inside his signed message. The intruder then forwards this message to *B* claiming to be its originator (line

13), and after *B* checks the intruder's signature he will believe that both *Xa* and *Ya* came from the intruder and that *Ya* is a secret shared by only himself and the intruder.

This single-session attack on the CCITT X.509 protocol was first publicized by Abadi and Needham in [AN94] where they note that:

> ... although *Ya* is transferred in a signed message, there is no evidence to suggest that the sender is actually aware of the data sent in the private part of the message. This corresponds to a scenario where some third party intercepts a message and removes the existing signature while adding his own, blindly copying the encrypted section within the signed message.

This concludes our discussion of single-session attacks on cryptographic protocols. We next take up the class of attacks that depend on executing two or more sessions of the same protocol either sequentially or in parallel.

6.2.4 The Proposed ISO Protocol

We begin our discussion of multi-session attacks with a protocol proposed for the International Organization for Standardization (ISO) standard for "entity authentication using symmetric techniques" [ISO90]. The protocol assumes that users *A* and *B* share some secret key, *k*, and that for one agent to authenticate the other they should engage in the following protocol:

- Agent *A* generates a nonce and sends it to *B* encrypted under *k*
- Agent *B* decrypts *A*'s message and sends back the nonce to *A*
- Agent *A* checks to see that *B*'s response matches her nonce

Here *B* is proving his identity to *A* by demonstrating his knowledge of the key *k* which *A* believes only she and *B* know. Figure 6.13 gives the CPAL representation of this protocol.

(1) *X: assume(A.k == B.k);*

(2) *A: na := new;*

(3) *A:* $\rightarrow B$ *(e[na]k);*

(4) *B:* $\leftarrow (msg1)$ *;*

(5) *B:* $\rightarrow A$ *(d[msg1]k);*

(6) *A:* $\leftarrow (msg2)$ *;*

(7) *A: assert(A.na == A.msg2);*

Figure 6.13 : The Proposed ISO Protocol

As the failure condition for this protocol we use the predicate:

PlayRole(B,I)

This is a failure condition that we will see in many of the authentication protocols in this chapter. The predicate *PlayRole(B,I)* basically represents the fact that in order for the

protocol to fail, the intruder must be able to successfully pass himself off as *B* to *A*. The attack that CPAL-ES finds on the protocol is given below.

(1) *X: assume(A.k == B.k);*

(2) *A: na := new;*

(3) *A:* $\Rightarrow$ *I (e[na]k);*

(4) *I:* $\leftarrow (tmp1)$;

*** (1')** ***I:*** $\Rightarrow$ ***A (tmp1);***

*** (2')** ***A:*** $\leftarrow$ **$(tmp10)$;**

*** (3')** ***A:*** $\Rightarrow$ ***I (d[tmp10]k);***

*** (4')** ***I:*** $\leftarrow$ **$(tmp15)$;**

*** (5)** ***I:*** $\Rightarrow$ ***A (tmp15);***

(6) *A:* $\leftarrow (msg2)$ *;*

(7) *A: assert(A.na == A.msg2);*

Figure 6.14 : An Attack on the Proposed ISO Protocol

As stated earlier, the attack is a multi-session attack. When agent *A* sends the encryption of her nonce under *k* to start the protocol with *B*, the intruder intercepts that message. The intruder then starts a run of the protocol with *A* saying, in effect, “I'm agent *B* and I'd like you to authenticate yourself to me.” by sending *A*'s message back to her. Agent *A* answers this challenge by receiving the message, decrypting it using the key she shares with *B*, and returning the answer to *B*. Of course, the intruder intercepts *A*'s reply and can now send it back to *A* as it is the proper reply to *A*'s challenge in the original session. Upon receipt of this reply from the intruder, *A* believes that she has just successfully authenticated agent *B*! The attack is illustrated graphically below. Messages from the first run of the protocol are denoted by solid arrows and messages from the second run of the protocol are denoted by dashed arrows.

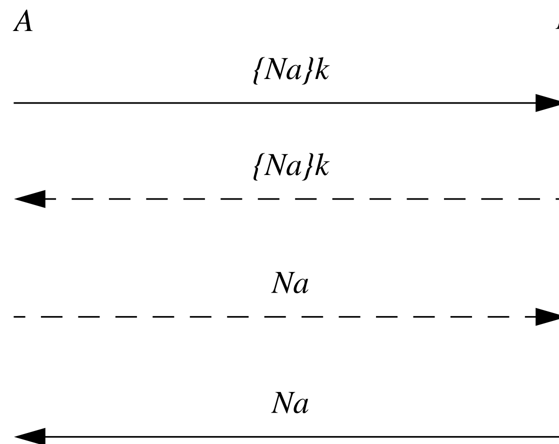


Figure 6.15 : A Graphical Representation of the Attack on the Proposed ISO Protocol

This type of attack is often called an *oracle attack* since the intruder is using agent *A* as an oracle to answer a question that the intruder cannot answer himself. This attack on the proposed ISO protocol was publicized in [BIR93].

6.2.5 The Denning and Sacco Protocol

In [DS81], Denning and Sacco proposed a key-exchange protocol based on timestamps and public-key cryptography. In the actual protocol, agents A and B interact with an authentication server to learn each other's public keys, but if we assume that they already know each other's public keys then the protocol operates as follows:

- Agent *A* opens the conversation by sending her name in the clear to *B* to notify *B* that she wishes to talk to him
- Agent *A* then generates a new session key, signs it and a timestamp with her private key, encrypts the signed message with *B*'s public key, and sends the result to *B*
- Agent B decrypts A's message, and, if A's signature is authentic, accepts the conversation key it contains

The CPAL representation of the Denning and Sacco protocol is given in Figure 6.16. For the failure condition for this protocol we use:

((PlayRole(A,I)) and Iknows(B.kab))

since the intruder must not only learn the conversation key but also must convince agent *B* that it is a good key for a conversation with agent *A*.

The result of our system's analysis of the Denning and Sacco protocol is the attack scenario illustrated in Figure 6.17. The output of the CPAL-ES system is given in Figure 6.18.

```
(1)  X: assume(Inverse(A.pub_A, A.priv_A));
(2)  X: assume(Inverse(B.pub_B, B.priv_B));
(3)  X: assume(Inverse(I.pub_I, I.priv_I));
(4)  X: assume(B.pub_A == A.pub_A);
(5)  X: assume(I.pub_A == A.pub_A);
(6)  X: assume(A.pub_B == B.pub_B);
(7)  X: assume(I.pub_B == B.pub_B);
(8)  X: assume(A.pub_I == I.pub_I);
(9)  X: assume(B.pub_I == I.pub_I);
(10) A:  → B (A);
(11) B:  ← (msg1);
(12) A: kab := new;
(13) A:  → B (e[e[<kab, ta>]priv_A]pub_B);
(14) B:  ← (msg2);
(15) B: msg3 := d[msg2]priv_B;
(16) B: if (Same(msg1, A)) then {kab := d[msg3]pub_A.1};
(17) B: if (Same(msg1, I)) then {kab := d[msg3]pub_I.1};
```

Figure 6.16 : The Denning and Sacco Protocol

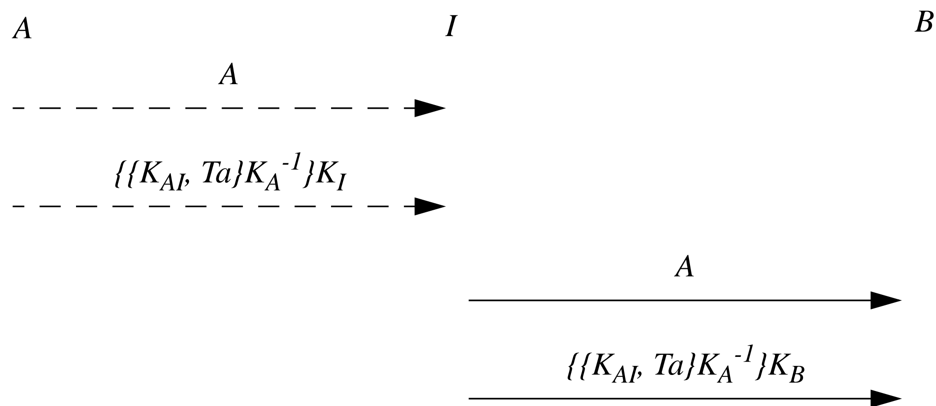


Figure 6.17 : A Graphical Representation of the Attack on the DS Protocol

(1) *X: assume(Inverse(A.pub_A, A.priv_A));*

(2) *X: assume(Inverse(B.pub_B, B.priv_B));*

(3) *X: assume(Inverse(I.pub_I, I.priv_I));*

(4) *X: assume(B.pub_A == A.pub_A);*

(5) *X: assume(I.pub_A == A.pub_A);*

(6) *X: assume(A.pub_B == B.pub_B);*

(7) *X: assume(I.pub_B == B.pub_B);*

(8) *X: assume(A.pub_I == I.pub_I);*

(9) *X: assume(B.pub_I == I.pub_I);*

*** (1')** ***A: ⇒ I (A);***

*** (2')** ***I: ← (tmp15);***

*** (3')** ***A: tmp7 := new;***

*** (4')** ***A: ⇒ I (e[e[<tmp7, ta>]priv_A]pub_I);***

*** (5')** ***I: ← (tmp16);***

(10) *I: ⇒ B (A);*

(11) *B: ← (msg1) ;*

*** (12)** ***I: tmp3 := e[d[tmp16]priv_I]pub_B;***

*** (13)** ***I: ⇒ B (tmp3);***

(14) *B: ← (msg2) ;*

(15) *B: msg3 := d[msg2]priv_B;*

(16) *B: if (Same(msg1, A)) then {kab := d[msg3]pub_A.1};*

(17) *B: if (Same(msg1, I)) then {kab := d[msg3]pub_I.1};*

Figure 6.18 : An Attack on the Denning and Sacco Protocol

This attack depends on agent *A* having run the protocol with the intruder at some time in the past. Once principal A does this the intruder receives the message:

$\{\{K_{AI}, Ta\}K_A^{-1}\}K_I$

which contains a portion signed by *A* proposing K_{AI} as a session key. The intruder can decrypt this message to learn K_{AI} and then encrypt the signed portion with *B*'s public key:

$$\{\{K_{AI}, Ta\}K_A^{-1}\}K_B$$

The intruder can then run the protocol with agent *B* and claim to be agent *A*. In his second message to *B* the intruder will send the above message and when *B* decrypts it and checks the signature he will believe that agent *A* has proposed K_{AI} as a conversation key for the two. At this point the intruder, masquerading as principal *A*, has established a "private" session with agent *B*. This attack on the Denning and Sacco protocol was demonstrated in Abadi and Needham's 1994 paper "Prudent Engineering Practice for Cryptographic Protocols" [AN94].

We now move on to another class of attacks that, strictly speaking, are multi-session attacks but are different enough from the types of multi-session attacks we have discussed so far to merit their own class. These attacks are called *key-compromise attacks* and were introduced at the beginning of Section 6.2.

Recall that the justification for session keys is that there are benefits to using a different, weaker key to protect each new conversation with a principal as opposed to always using the same "strong" key. By utilizing session keys, users are accepting the fact that an intruder may be able, with great effort, to break a session key and gain access to an old conversation that it protected. Users hope that the time and effort necessary to compromise an old session key are not worth it to an intruder if he will only get to read one very old conversation. However, if the compromise of one old session key allows an intruder to easily compromise **every** conversation from that point on, then it may well be worth the time and effort necessary to compromise that one key.

Protocols in which current runs become vulnerable due to the compromise of a single old session key are said to suffer from weaknesses that can be exploited by a key-

compromise attack. In the next few subsections we will present examples of several protocols that are vulnerable to key-compromise attacks and show that these attacks can be discovered by our system.

6.2.6 The Needham and Schroeder Private-Key Protocol

Needham and Schroeder's 1978 paper on "Using Encryption for Authentication in Large Networks of Computers" [NS78] is probably responsible for stimulating most of the work done on cryptographic protocols to date. Not only were they the first to suggest using cryptographic protocols to accomplish various sensitive tasks over an insecure network, but one of the protocols presented in their seminal paper turned out to be flawed giving rise to the field of cryptographic protocol analysis. The protocol that Denning and Sacco attacked in [DS81] was an authentication protocol that used symmetric-key cryptography and a trusted authentication server to allow two users to prove their identities to each other and establish a shared key for a private conversation.

Needham and Schroeder assume that each principal shares a secret key with the authentication server (e.g. K_{AS} is known only to agent A and the authentication server and K_{BS} is known only to agent B and the authentication server). For agent A to establish a private conversation with agent B with the help of S, the trusted authentication server, the protocol is as follows:

- User A sends a message in the clear to S. The message contains A's name, the name of the party A wishes to talk to (B), and a nonce generated by A
- Upon receipt of A's message the authentication server generates a new conversation key for A and B to use and sends a reply to A. The reply, which is encrypted under the secret key that A and S share, contains A's nonce, B's name, the conversation key, and a ticket for B. The ticket is encrypted using S and B's shared key so A will not be able to read it. It contains A's name and the conversation key.

- When agent *A* gets this response back from the authentication server she decrypts it and checks that it contains *B*'s name and her own nonce, and, if so, stores the conversation it contains and sends the ticket on to *B*
- After receiving the ticket and decrypting it, agent B knows that agent A wants to talk to him and also knows the conversation key the authentication server has chosen

At this point we interrupt our description of the protocol to review what each party knows according to Needham and Schroeder:

> *A* now knows that any communication [she] receives encrypted with [the conversation key] must have originated with *B*, and also that any communication [she] emits [encrypted with the conversation key] will be understood only by *B*. Both are known because the only messages containing [the conversation key] that have ever been sent are tied to *A*'s and *B*'s secret keys. *B* is in a similar state, mutatis mutandis. It is important, however, to be sure that no part of the protocol exchange or ensuing conversation is being replayed by an intruder from a recording of a previous conversation between *A* and *B*. In relationship to this question the positions of *A* and *B* differ. *A* is aware that he has not used [the conversation key] before and therefore has no reason to fear that material encrypted with it is other than the legitimate responses from *B*. *B* ... is unclear that [the ticket] and the subsequent messages supposedly from *A* are not being replayed.

- To guard against the possibility of replay discussed above, agent *B* generates a nonce and sends it to agent *A* encrypted under the conversation key
- Upon receipt of *B*'s "challenge", *A* decrypts the message to learn *B*'s nonce, decrements it by one, encrypts the result with the conversation key, and sends that value to *B*
- Once agent *B* receives this reply and checks it, "the mutual confidence is sufficient to enable substantive communication, encrypted with [the conversation key], to begin" according to Needham and Schroeder

The CPAL representation of the Needham and Schroeder Private-Key protocol in given in Figure 6.19.

```
(1)  X: assume(A.ka == S.ka);
(2)  X: assume(B.kb == S.kb);
(3)  X: assume(not(Iknows(X)) or Iknows(Decrement(X)));
(4)  X: assume(WeakKey(S.ck));
(5)  A: na := new;
(6)  A: r := B;
(7)  A: → S (<A,B,na>);
(8)  S: ← (msg1) ;
(9)  S: ck := new;
(10) S: → A (e[<msg1.3,msg1.2,ck,e[<ck,msg1.1>]kb>]ka);
(11) A: ← (msg2) ;
(12) A: msg3 := d[msg2]ka;
(13) A: na1 := msg3.1;
(14) A: r1 := msg3.2;
(15) A: ck := msg3.4;
(16) A: ticket := msg3.4;
(17) A: assert (A.na == A.na1);
(18) A: assert (A.r == A.r1);
(19) A: → B (ticket);
(20) B: ← (msg4) ;
(21) B: msg5 := d[msg4]kb;
(22) B: ck := msg5.1;
(23) B: r2 := msg5.2;
(24) B: nb := new;
(25) B: → A (e[nb]ck);
(26) A: ← (msg6) ;
(27) A: msg7 := d[msg6]ck;
(28) A: msg8 := Decrement(msg7);
(29) A: → B (e[msg8]ck);
(30) B: ← (msg9) ;
(31) B: msg10 := d[msg9]ck;
(32) B: assert(B.msg10 == Decrement(B.nb));
```

Figure 6.19 : The Needham and Schroeder Private-Key Protocol

Before moving on to the statement of failure for this protocol we need to explain lines three and four. Line three is an example of how protocol-specific axioms can be added to our system. Specifically, we are defining an axiom about the *Decrement()* predicate which is just an uninterpreted predicate to our system. In this case, all we're telling the system about the *Decrement()* predicate is that if the intruder knows a value, *x*, then he also knows *Decrement(x)*.

In line four we are tagging all conversation keys generated by the authentication server as "weak" keys. In our system all keys are assumed to be "strong" unless they are explicitly declared to be "weak". The system assumes that strong keys will never be cryptanalytically compromised by an intruder while weak keys can be broken by an intruder using some (possibly large) amount of time and effort. All key-compromise attacks depend on certain keys being "weak" and we added this designation into our system to allow it to discover such attacks.

Since this is an authentication protocol we take the following as its failure condition:

PlayRole(A,I)

This means that the intruder, posing as *A*, was able to establish a private conversation with *B*. The output from our system is given below.

*** (1')** ***A: tmp206 := new;***

*** (2')** ***A: tmp204 := B;***

*** (3')** ***A: ⇒ I (<A, B, tmp206>);***

*** (4')** ***I: ← (tmp1);***

*** (5')** ***I: ⇒ S (tmp1);***

*** (6')** ***S: ← (tmp154);***

*** (7')** ***S: tmp155 := new;***

*** (8')** ***S: ⇒ I (e[<tmp154.3, tmp154.2, tmp155, e[<tmp155, tmp154.1>]kb>]ka);***

*** (9')** ***I: ← (tmp125)***

*** (10')** ***I: ⇒ A (tmp125);***

*** (11')** ***A: ← (tmp202);***

*** (12')** ***A: tmp201 := d[tmp202]ka;***

*** (13')** ***A: tmp205 := tmp201.1;***

*** (14')** ***A: tmp203 := tmp201.2;***

*** (15')** ***A: tmp200 := tmp201.4;***

*** (16')** ***A: tmp199 := tmp201.4;***

*** (17')** ***A: assert(A.tmp206 == A.tmp205);***

*** (18')** ***A: assert(A.tmp204 == A.tmp203);***

*** (19')** ***A: ⇒ I (tmp199);***

*** (20')** ***I: ← (tmp124);***

*** (21')** ***S: ⇒ I (tmp155);***

*** (22')** ***I: ← (tmp123);***

(1) *X: assume(A.ka == S.ka);*

(2) *X: assume(B.kb == S.kb);*

(3) *X: assume(not(Iknows(X)) or Iknows(Decrement(X)));*

(4) *X: assume(WeakKey(S.ck));*

(5) *I: ⇒ S (tmp1);*

(6) *S: ← (msg1);*

(7) *S: ck := new;*

(8) S: ⇒ *I (e[<msg1.3, msg1.2, ck, e[<ck, msg1.1>]kb>]ka);*

(9) *I:* ← *(tmp2) ;*

*** (10) *I:* ⇒ *B (tmp124);***

(11) *B:* ← *(msg4) ;*

(12) *B: msg5 := d[msg4]kb;*

(13) *B: ck := msg5.1;*

(14) *B: r2 := msg5.2;*

(15) *B: nb := new;*

(16) *B:* ⇒ *I (e[nb]ck);*

(17) *I:* ← *(tmp4) ;*

*** (18) *I: tmp638 := e[Decrement(d[tmp4]tmp123)]tmp123;***

*** (19) *I:* ⇒ *B (tmp638);***

(20) *B:* ← *(msg9) ;*

(21) *B: msg10 := d[msg9]ck;*

(22) *B: assert(B.msg10 == Decrement(B.nb));*

Figure 6.20 : An Attack on the Needham and Schroeder Private-Key Protocol

The attack is both a multi-session and key-compromise attack. First the intruder observes all the messages in an old run of the protocol between *A* and *B* (lines 1' through 22'). By doing this, the intruder learns the ticket that *S* generated for *B*, but the intruder cannot read it because the message is encrypted under *B*'s secret key:

$\{A, ck\}K_{BS}$

The intruder then records all the messages (encrypted under *ck*) that *A* and *B* exchange in the subsequent conversation. Again, the intruder cannot read these messages because he doesn't know *ck*. However, the intruder can then cryptanalyze the conversation between *A* and *B* and, with some time and effort, determine the value of *ck*. At this point the intruder can read the conversation that *A* and *B* had, but it is assumed that the conversation will be old enough by the time the intruder breaks *ck* to be of little value to him. This was exactly

the job of the conversation key - to protect *A* and *B*'s conversation for some period of time but not indefinitely. If the intruder needs to expend much time and effort to compromise a conversation key, and as a result he is only able to read one conversation that is quite old, then compromising a conversation key is probably not worth the time and effort. However, if we look at Figure 6.20 we see that by compromising *ck* the intruder can do much more than just read an old conversation between *A* and *B*. It may take a substantial amount of time, but once the intruder has compromised a conversation key, he can begin the protocol at the first message A sends to B asking to start a new private conversation. The intruder sends the old ticket to *B* claiming to be *A* and wanting to start a new private conversation (line 10). After receiving this ticket and decrypting it, *B* will send the challenge (a nonce encrypted under the proposed conversation key) back to *A*. The intruder will intercept this message to *A*. Now the intruder's effort to break the old conversation key pays off because he is able to decrypt *B*'s message, decrement the nonce, encrypt the result using the conversation key, and send the correct reply back to *B* (lines 18 and 19). After verifying the intruder's reply, *B* believes that he has established a new private conversation with *A* though that conversation is really with the intruder. This is the same attack that was pointed out by Denning and Sacco in [DS81].

6.2.7 The Beller Protocol

Another protocol that is vulnerable to a key-compromise attack is one proposed by Beller et al. in [BEL93]. In this paper a protocol is proposed to protect communications between PCS (Portable Communications Systems) or "portables" and the telephone network. It is assumed that portables communicate via microwave radio waves with the nearest RCE (Radio Control Equipment) which are located on utility poles or in buildings. Each RCE would be connected to the telephone network and contain hardware to perform certain cryptographic functions. Also connected to the telephone network is a database

which also contains cryptographic hardware. It is assumed that each portable and RCE has a unique key known only to itself and the database.

Beller et al. propose the following protocol to allow a portable to establish a conversation key with an RCE to protect communications between the two:

- Upon service request by the portable, the RCE requests a conversation key from the database.
- The database generates a new key and sends a two-part message back to the RCE. One part is the conversation key encrypted with the RCE's key and the other part is the conversation key encrypted with the portable's key.
- The RCE decrypts the first part of the message to learn the conversation key and sends the second part of the message on to the portable.
- The portable receives the message, decrypts it, and learns the conversation key as well.

According to [BEL93], "[The portable and the RCE] now share a common session key which, after verifying it with standard messages, may be used for encryption and decryption." The CPAL specification of this protocol is given in Figure 6.21. We denote the portable by P, the RCE by R, and the database by D, and intruder by I.

(1) *X: assume(D.kp == A.kp);*

(2) *X: assume(D.kr == R.kr);*

(3) *X: assume(WeakKey(D.ck));*

(4) *P:* $\rightarrow R$ *(P);*

(5) *R:* $\leftarrow (msg1)$ *;*

(6) *R:* $\rightarrow D$ *(<msg1, R>);*

(7) *D:* $\leftarrow (msg2)$;

(8) *D: ck := new;*

(9) *D:* $\rightarrow R$ *(<e[ck]kr,e[ck]kp>);*

(10) *R:* $\leftarrow (msg3)$ *;*

(11) *R: ck := d[msg3.1]kr;*

(12) *R:* $\rightarrow P$ *(msg3.2);*

(13) *P:* $\leftarrow (msg4)$ *;*

(14) *A: ck := d[msg4]kp;*

Figure 6.21 : The Beller Private-Key Protocol

Since this is a key-exchange protocol we use the following as a statement of failure:

Same(I.ck, A.ck)

Note that as with the Needham and Schroeder protocol in Section 6.2.6 we have specified that conversation keys are “weak” (line 3). The attack on this protocol is given in Figure 6.22 below.

*** (1’)** ***P:*** $\Rightarrow$ ***I (P);***

*** (2’)** ***I:*** $\leftarrow$ ***(tmp271);***

*** (3’)** ***I:*** $\Rightarrow$ ***R (tmp271);***

*** (4’)** ***R:*** $\leftarrow$ ***(tmp82);***

*** (5’)** ***R:*** $\Rightarrow$ ***I (<tmp82, R>);***

*** (6’)** ***I:*** $\leftarrow$ ***(tmp270);***

*** (7')** ***I: ⇒ D (tmp270);***

*** (8')** ***D: ← (tmp138) ;***

*** (9')** ***D: tmp139 := new;***

*** (10')** ***D: ⇒ I (<e[tmp139]kr, e[tmp139]kp>);***

*** (11')** ***I: ← (tmp269) ;***

*** (12')** ***D: ⇒ I (tmp139);***

*** (13')** ***I: ← (tmp268) ;***

(1) *X: assume(D.kp == A.kp);*

(2) *X: assume(D.kr == R.kr);*

(3) *P: ⇒ I (P);*

(4) *I: ← (tmp1) ;*

(5) *I: ⇒ R (tmp1);*

(6) *R: ← (msg1) ;*

(7) *R: ⇒ I (<msg1, R>);*

(8) *I: ← (tmp2) ;*

(9) *I: ⇒ D (tmp2);*

(10) *D: ← (msg2) ;*

(11) *D: ck := new;*

(12) *D: ⇒ I (<e[ck]kr,e[ck]kp>);*

(13) *I: ← (tmp3) ;*

*** (14)** ***I: ck := tmp269;***

*** (15)** ***I: ⇒ R (tmp269);***

(16) *R: ← (msg3) ;*

(17) *R: ck := d[msg3.1]kr;*

(18) *R: ⇒ I (msg3.2);*

(19) *I: ← (tmp4) ;*

(20) *I: ⇒ P (tmp4);*

(21) *P: ← (msg4) ;*

(22) *A: ck := d[msg4]kp;*

Figure 6.22: An Attack on the Beller Private-Key Protocol

The attack is very similar to the one we saw in Section 6.2.6 on the Needham and Schroeder protocol. After observing a run of the protocol and compromising the session key, the intruder can replay the database's message suggesting that session key and force the portable and RCE to use an old, compromised session key (that is known to the intruder) to protect their communication. No attack on the Beller Private-Key protocol has been previously published.

This concludes our discussion of the types of known-attacks that our system can discover. In the next section we will present a new type of attack on cryptographic protocols and demonstrate our system's ability to handle this type of attack.

6.3 A New Type of Attack Defined and Demonstrated

Up to this point we have analyzed each protocol in isolation and all the attacks given, whether single-session, multi-session, or key-compromise, have involved only that one protocol. Real-world computer systems, however, typically allow users to run many different cryptographic protocols. A user might run one protocol for authentication, another for key-exchange, another for electronic funds transfer, and still others for various other tasks. While each protocol might be secure in isolation there is a possibility that the interaction between two or more protocols might render one of them insecure.

Unanticipated interactions among protocols are even more likely, given that agents typically use the same shared secrets and public/private key pairs in a number of different protocols. For example, if agents A and B have a shared secret which they use to both authenticate each other and to agree on new session keys for private conversations there is a possibility that an intruder will be able to use a message from the authentication protocol to attack the key-exchange protocol or vice versa. We call this type of attack a *protocol-interaction attack* and currently there are no cryptographic protocol analysis tools that look

for these types of attacks. In the following sections we will demonstrate a protocol-interaction attack and show how our system can find such attacks.

6.3.1 The Protocols

Let us assume that agents A and B share a secret key, K_{AB}. In order for A to authenticate B she runs the following protocol:

- Agent A generates a nonce and sends it to B
- Agent B encrypts A's nonce using K_{AB} and returns it to A
- Agent A checks B's reply, and if it is correct, believes that it must have come from B since he is the only other principal that knows K_{AB}

This is simply the one-way version of the ISO authentication protocol. Recall that this protocol is flawed since an intruder can start a parallel session pretending to be B and send A's challenge back to her. Agent A will reply, as dictated by the protocol, by encrypting the value with K_{AB} and sending it back to the intruder. The intruder can then resume the original session with A and send back the answer he has just learned. The fact that this authentication protocol is flawed will play no role in the protocol-interaction attack that we intend to demonstrate.

Let us also assume that in order for A and B to agree on a session key for a private conversation, they engage in the following key-exchange protocol:

- Agent A generates a new conversation key, encrypts it under K_{AB}, and sends the result to agent B
- Agent B receives A's message, decrypts it, and stores the conversation key
- Agents A and B can then send each other sensitive data encrypted using the conversation key

The CPAL specifications for these two protocols (with failure conditions added) are given below.

(1) *X: assume(A.kab == B.kab);*

(2) *A: x := new;*

(3) *A:* $\rightarrow B$ *(x);*

(4) *B:* $\leftarrow (msg1)$ *;*

(5) *B:* $\rightarrow A$ *(e[msg1]kab);*

(6) *A:* $\leftarrow (msg2)$;

(7) *A: x1 := d[msg2]kab;*

(8) *A: assert(A.x == A.x1);*

PlayRole(B,I)

Figure 6.23 : The Authentication Protocol

(1) *X: assume(A.kab == B.kab);*

(2) *A: ck := new;*

(3) *A:* $\rightarrow B$ *(e[ck]kab);*

(4) *B:* $\leftarrow (msg1)$ *;*

(5) *B: ck:= d[msg1]kab;*

Same(I.k, B.ck)

Figure 6.24 : The Key-Exchange Protocol

Notice that there is no assumption in the key-exchange protocol that conversation keys are "weak". This being the case, we do not know of any attacks (other than brute-force) that this protocol is vulnerable to if run in isolation.

6.3.2 A Protocol-Interaction Attack

The key-exchange protocol in Figure 6.24 is susceptible to an attack (shown in Figure 6.25) if it is run in the same environment as the authentication protocol given above. In Figure 6.25, solid arrows represent messages from the key-exchange protocol and dashed arrows represent messages from the authentication protocol.

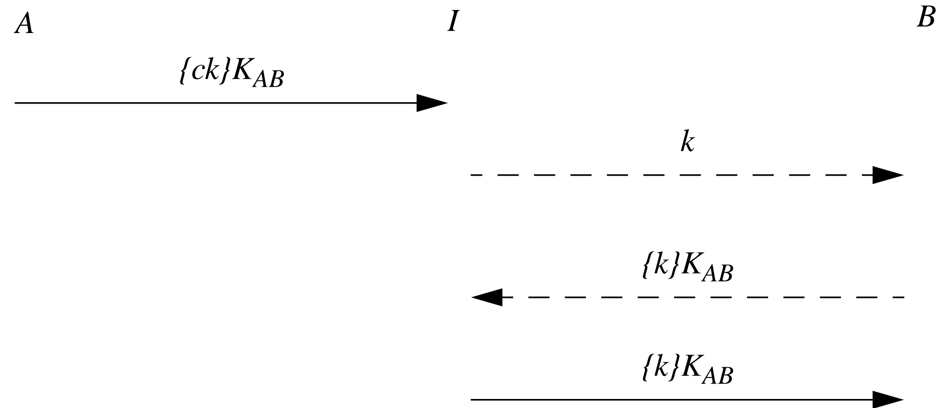


Figure 6.25 : A Graphical Representation of a Protocol-Interaction Attack

The attack from Figure 6.25 works as follows. Agent *A* starts the key-exchange protocol with agent *B* by sending a new session key encrypted under K_{AB}. The intruder intercepts *A*'s message to *B* and starts the authentication protocol with *B* pretending to be *A*. Instead of the nonce normally sent in the first message of authentication protocol, the intruder sends *k*, some value that he would eventually like to fool *B* into accepting as the session key for *A*'s conversation request. Of course *B* can't tell the difference between *k* and some nonce from *A* so he follows the authentication protocol and sends *A* the encryption of *k* under K_{AB} to authenticate himself. The intruder intercepts this message which is exactly what he needs to send to *B* to complete the key-exchange protocol and get *B* to accept *k* as the conversation key.

Extending our methodology to include protocol-interaction attacks is actually quite easy. Normally, we generate all the axioms that will go into the automatic theorem prover from a single protocol, but there is no reason that we must limit ourselves in this way. We modified our system so that it can read in any number of protocols (each with its own specific statement of failure) and store them on a stack. Then we pop them off the stack one at a time and examine them individually as before - except that the parallel-session axioms

are generated, not just from the protocol under consideration, but from all the protocols that were given. After making these modifications and inputting the protocols from Figures 6.23 and 6.24 our system was able to find the protocol-interaction attack shown in Figure 6.25 as shown below.

(1) *X: assume(A.kab == B.kab);*

(2) *A: ck := new;*

(3) *A:* $\Rightarrow$ *I (e[ck]kab);*

(4) *I:* $\leftarrow (tmp1)$;

*** (1')** ***I:*** $\Rightarrow$ ***B (k);***

*** (2')** ***B:*** $\leftarrow (msg37)$ **;**

*** (3')** ***B:*** $\Rightarrow$ ***I (e[msg37]kab);***

*** (4')** ***I:*** $\leftarrow (tmp65)$ ***;***

*** (5)** ***I:*** $\Rightarrow$ ***B (tmp65);***

(6) *B:* $\leftarrow (msg1)$ *;*

(7) *B: ck:= d[msg1]kab;*

Figure 6.26 : A Protocol-Interaction Attack on the Key-Exchange Protocol

6.4 Summary

In this chapter we have presented the results of our analysis of a number of well-known cryptographic protocols. We demonstrated our methodology's ability to discover many of the known attacks on these protocols and then presented an important new type of attack that our approach also handles quite easily. In the next chapter we present our conclusions and suggest some future work.

Chapter 7

Conclusions

Computer networks have become immensely popular and extremely valuable in the personal computer age, and we can expect their popularity to increase dramatically as advances continue to be made that expand their content and speed while decreasing their cost. One problem with computer networks is that information transmitted on them tends to be particularly vulnerable to eavesdropping and modification by intruders, and their connectivity and openness tends to exacerbate many other computer security concerns. Part of the solution to the network security problem has been the development of cryptographic protocols that are aimed at protecting the privacy and integrity of messages that travel over insecure networks.

When designed and used correctly, cryptographic protocols can provide security for tasks such as authentication, key-distribution, voting, electronic commerce, and others. However, the discovery of flaws in a number of proposed and implemented protocols has lead the security community to conclude that protocol design is difficult and that techniques for examining proposed protocols must be developed.

7.1 Review of the Research

In this dissertation we have presented a methodology for examining cryptographic protocols that both compliments existing methods and builds upon them. Our approach is to have the user specify the protocol(s) to be tested in CPAL and a statement of failure for each protocol in predicate calculus. The system then uses the formal semantics of CPAL to generate theorems that correspond to the preconditions at each point for failure of the protocol. An automatic theorem prover is then used to try to find a constructive proof of one of these theorems, and, if it succeeds, the proof it generates is used to generate intruder actions that create the conditions for a failure. The resulting attack on the protocol can then be reported to the user.

As stated in Chapter 1, reasoning about the correctness of cryptographic protocols depends on showing that for all possible sequences of actions by all possible intruders, the protocol achieves its goals. Our method does not attempt to reason about protocol correctness but instead transforms the above statement into a statement about the satisfiability of the preconditions for failure at each point in the protocol and thereby defines a theoretical basis for attack generation.

Our approach is based on the general-purpose techniques of weakest precondition reasoning and deductive automatic programming and is extensible since new weakest preconditions can be defined and additional axioms can be added to the logical system. Furthermore, the search for an attack is performed completely formally (and automatically) once the protocol and failure condition have been specified. Lastly, the methodology is powerful since given an adequate set of axioms and enough time, it will find the constructive proof corresponding to any attack that exists for a given protocol, failure condition, and axiom set.

If we assume that the theorem prover performs a breadth-first search for a proof then we can halt its search at any point and be sure that no constructive proof for the given theorem and axiom set exists that is shorter than the longest proof that the theorem prover considered. This allows our approach to make a useful statement about a protocol even when it cannot prove that a flaw exists.

Another major contribution of this work is the discovery and handling of the important new class of *protocol-interaction attacks*. Since real computer systems typically include more than one type of protocol and since the number of possible interactions grows quickly as each protocol is added, it is important to check groups of protocols for these types of flaws. The power and automation of our approach makes it well-suited for this purpose.

7.2 Future Work

There are many potential extensions to this research, and in this section we discuss three specific areas that we consider to be of interest. They are: extension of the set of axioms used to prove theorems, adoption of another type of logic with which to prove theorems, and improvement of the automatic theorem prover. We discuss each of these in more detail below.

7.2.1 Additional Axioms

Some of the axioms that we specifically chose not to include in our preliminary implementation were those expressing the mathematical properties of the cryptosystems used by protocols. This was done primarily to reduce the amount of time it took our system to find attacks not based on the properties of cryptosystems (which constitute the majority of known attacks on published protocols). Without these axioms our system cannot find such attacks as Simmons' attack on the TMN protocol in [SIM85] and Bird's attack on his own XOR protocol in [BIR93]. For example, the Simmons attack on the TMN protocol

alluded to above is based on the fact that the encryption algorithms used in the protocol have the following properties. For the asymmetric cryptosystem:

- $(\forall x)(\forall y)$ *ep(X,K) * ep(Y,K) = ep(X*Y,K)*

and for the symmetric cryptosystem:

- $(\forall x)(\forall y)$ *e(X,Y) = e(Y,X)*

Our axioms do not currently model cryptographic algorithms at this level. Furthermore, the above properties are not part of the specification for the TMN protocol, but are characteristics of the cryptographic algorithms that were suggested for implementation. So it is inappropriate to view them as protocol-specific axioms. The addition of algebraic axioms would allow our system to discover this and other types of attacks, but would require improvements (see Section 7.2.3) to be made to the automatic theorem prover to find proofs in a reasonable amount of time.

7.2.2 Utilizing Other Logics

The use of another logic besides predicate calculus would allow us to take advantage of the increased power and expressiveness of other logical systems. The use of either first-order logic, temporal logic, nonmonotonic logic, or one of the specialized modal logics developed for reasoning about cryptographic protocols ([BAN89], [AT91], [BIE90], etc.) might help our system to find new types of attacks based upon the properties that such logics can represent. In order to use another logic in our system we would merely need to add their inference rules to our system and update the module of our system that makes modifications to a protocol once a proof is found.

7.2.3 Improvements to the Automatic Theorem Prover

Our system would also benefit from improvements made to the automatic theorem proving module that would allow it to find proofs more quickly. Such improvements could

include new heuristics that would be useful for proving theorems about cryptographic protocols or a parallelization of the theorem prover's search for a proof.

7.3 Summary

In this dissertation we have presented a powerful, general, extensible, and formal methodology that automatically examines cryptographic protocols. Given an adequate set of axioms and enough time, our method will find any attack that exists for a given protocol and failure condition. Even if our methodology does not discover a flaw in the amount of time it is given to run, we can make a concrete statement about the minimum length of a constructive proof for any attack that might exist on the protocol (for the given failure condition and axiom set) as a result of its analysis. A preliminary implementation of our methodology has had great success in finding both known and previously unknown flaws in a significant number of published protocols.

Appendix A

A Sample CPAL Analysis

In this appendix we will illustrate how a cryptographic protocol can be specified using CPAL and we will demonstrate Yasinsac's analysis technique from [YAS96]. We take as our example protocol the Needham and Schroeder Private-Key Protocol which we have already discussed in Chapter 1. To quickly review, this protocol is an authentication and key-distribution scheme for two agents, *A* and *B*, who both trust a single authentication server, *AS*. It is assumed that both *A* and *B* share secret keys (*KA* and *KB* respectively) with the authentication server, and that they wish to establish a new conversation key, *CK*, to encrypt communications between themselves.

The protocol operates as follows. In the first step of the protocol *A* sends a message in the clear to the authentication server. The message contains *A*'s name, *B*'s name, and a nonce generated by *A*. In the second step, the authentication server replies to *A* with a message encrypted using the key *KA*. In this message the authentication server includes the nonce that *A* sent, *B*'s name, a session key that the *AS* created, and the ticket for *B* which contains the session key and *A*'s name encrypted under *KB.* Upon receipt of this message *A*

can remove the outer layer of encryption using *KA* and then check to see that the nonce is the same one that she sent in step one. If so, she knows that the message from the authentication server is *fresh* because it must have been generated after *A* generated her nonce. Since she could be attempting to establish communication sessions with a number of agents simultaneously, *A* also checks for *B*'s name to make sure that this is a reply to her request to talk to *B*. From this same message *A* also learns *CK*, the conversation key created by the authentication server, and the ticket. Since *A* does not know *KB* she cannot read the contents of the ticket, but she can forward it to *B* which she does in step three.

When *B* receives the message *A* sent in step three he decrypts it and discovers that *A* wishes to talk to him and that the session key is *CK*. In step four, *B* generates a nonce, encrypts it under the session key, and sends the result back to *A*. In the final step of the protocol, *A* receives *B*'s challenge, decrypts it, subtracts one from the nonce, encrypts that value with the session key, and sends the result to *B*. After decrypting this reply and checking that *A* has indeed returned one less than the nonce sent in step four, *A* and *B* each believe that they have authenticated each other and now share a conversation key known only to them and the authentication server whom they trust. The protocol is given in pseudocode below:

$$\begin{array}{ll} A \rightarrow AS: & A, B, I_{A1} \\ AS \rightarrow A: & \{I_{A1}, B, CK, \{CK, A\}_{KB}\}_{KA} \\ A \rightarrow B: & \{CK, A\}_{KB} \\ B \rightarrow A: & \{I_B\}_{CK} \\ A \rightarrow B: & \{I_B - 1\}_{CK} \end{array}$$

Figure A.1: Pseudocode Representation of the Needham and Schroeder Private-Key Protocol

Translating the Protocol into CPAL

The first step in our analysis is to translate the protocol from pseudocode into CPAL as shown below:

Pseudocode		CPAL
		X: **assume**(*AS.ka* == *A.ka*);
		X: **assume**(*AS.kb* == *B.kb*);
		A: *i* := *new*;
$A \rightarrow AS$:	A,B,I_{A1}	*A*: $\rightarrow AS$(<*A*,*B*,*i*>) ;
		AS: $\leftarrow (msg)$;
		AS: *ck* := *new*;
		AS: *ticket* := e[<*ck*, *msg.1*>]*kb*;
$AS \rightarrow A$:	$\{I_{A1},B,CK,\{CK,A\}KB\}KA$	*AS*: $\rightarrow A$ (e[<*msg.3,B,ck,ticket*>]*ka*);
		A: $\leftarrow (msg)$;
		A: *tmp* := d[*msg*]*ka*;
		A: **assert**(*i* == *tmp.1*);
		A: **assert**(*B* == *tmp.2*);
		A: *ck* := *tmp.3*;
		A: *ticket* := *tmp.4*;
$A \rightarrow B$:	$\{CK,A\}KB$	*A*: $\rightarrow B$ (*ticket*);
		B: $\leftarrow (msg)$;
		B: *ck* := (d[*msg*]*kb*).*1*;
		B: *i* := *new*;
$B \rightarrow A$:	$\{I_B\}CK$	*B*: $\rightarrow A$ (e[*i*]*ck*);
		A: $\leftarrow (msg2)$;
		A: *i2* := d[*msg2*]*ck*;
$A \rightarrow B$:	$\{I_B - 1\}CK$	*A*: $\rightarrow B$ (*i2-1*);
		B: $\leftarrow (msg2)$;
		B: **assert**(*i-1* == d[*msg2*]*ck*)

It is clear from this example that it normally takes more CPAL statements than pseudocode to specify a protocol, but this is mainly due to CPAL's insistence on making all assumptions, actions and goals explicit in the protocol specification.

The first step we must take before we can use CPAL's formal semantics to analyze this protocol is to replace all of the insecure sends with secure sends to the intruder. To begin with, we will assume that the intruder simply preforms the benign action of forwarding a message unmodified to the intended recipient, but we know that the intruder may decide to take other actions later. Replacing the insecure sends with secure sends to the intruder we get:

X: **assume**(*AS.ka* == *A.ka*);

X: **assume**(*AS.kb* == *B.kb*);

A: *i* := *new*;

A: $\Rightarrow I(<A,B,i>)$;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow AS(tmp1)$;

AS: $\leftarrow (msg)$;

AS: *ck* := *new*;

AS: *ticket* := e[<*ck*, *msg.1*>]*kb*;

AS: $\Rightarrow I$ (e[<*msg.3,B,ck,ticket*>]*ka*);

I: $\leftarrow (tmp2)$;

I: $\Rightarrow A$ (tmp2);

A: $\leftarrow (msg)$;

A: *tmp* := d[*msg*]*ka*;

A: **assert**(*i* == *tmp.1*);

A: **assert**(*B* == *tmp.2*);

A: *ck* := *tmp.3*;

A: *ticket* := *tmp.4*;

A: $\Rightarrow I$ (*ticket*);

I: $\leftarrow (tmp3)$;

I: $\Rightarrow B$ (tmp3);

B: $\leftarrow (msg)$;

B: *ck* := (d[*msg*]*kb*).*1*;

B: *i* := *new*;

B: $\Rightarrow I$ (e[*i*]*ck*);

I: $\leftarrow (tmp4)$;

I: $\Rightarrow A$ (tmp4);

A: $\leftarrow (msg2)$;

A: *i2* := d[*msg2*]*ck*;

A: $\Rightarrow I$ (*i2-1*);

I: $\leftarrow (tmp5)$;

I: $\Rightarrow B$ (*tmp5*);

B: $\leftarrow (msg2)$;

B: **assert**(*i-1* == d[*msg2*]*ck*)

We are now ready to use Yasinsac's weakest precondition definitions for CPAL to assign a formal semantics to this protocol. For this example, we will simply use the predicate TRUE as the postcondition we want to hold after execution of the protocol since the protocol's goals are represented by the assert statement in the last line. Given this postcondition, we can then calculate the weakest precondition of the last statement in the protocol and this postcondition: *wp*("*B*: **assert**(i-1 == d[*msg2*]*ck*)", *TRUE*). Using the weakest precondition definitions in Table 3.2 we find that ($TRUE \wedge$ (*i-1* == d[*msg2*]*ck*)) is the weakest precondition for the final statement of the protocol. Using the definition of statement concatenation in Table 3.2, we then use this weakest precondition for the final statement as the postcondition for the second to last statement and compute the weakest precondition for the second to last statement: *wp*("*B*: $\leftarrow (msg2)$ ", ($TRUE \wedge$ (*i-1* == d[*msg2*]*ck*))) which yields ($TRUE \wedge$ (*i-1* == d[*B's input queue*]*ck*)) as the weakest precondition for the second to last statement of the protocol. By continuing in this manner backwards through the

protocol we can compute the weakest precondition for each statement in the protocol as illustrated below:

```
(AS.ka == A.ka) ⊃ ((AS.kb == B.kb) ⊃ (TRUE ∧ (B == (d[e[<(<A,B,unique.3>).3,
B,unique.2, e[<unique.2, (<A,B,unique.3>).1>] AS.kb>]AS.ka]
A.ka).2) ∧ (unique.3 == (d[e[<(<A,B,unique.3>).3, B,unique.2,
e[<unique.2,(<A,B,unique.3>).1>] AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 ==
d[(d[e[unique.1] (d[(d[e[<(<A,B,unique.3>).3,B, unique.2, e[<unique.2,
(<A,B,unique.3>).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(<A,B,unique.3>).3,B,unique.2,e[<unique.2, (<A,B,unique.3>).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(<A,B,unique.3>).3,B, unique.2,
e[<unique.2,(<A,B,unique.3>).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)))
```

X: **assume**(*AS.ka* == *A.ka*);

```
(AS.kb == B.kb) ⊃ (TRUE ∧ (B == (d[e[<(<A,B,unique.3>).3,B,unique.2,
e[<unique.2, (<A,B,unique.3>).1>] AS.kb>]AS.ka] A.ka).2) ∧ (unique.3 ==
(d[e[<(<A,B,unique.3>).3, B,unique.2, e[<unique.2,(<A,B,unique.3>).1>]
AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 == d[(d[e[unique.1]
(d[(d[e[<(<A,B,unique.3>).3,B, unique.2, e[<unique.2,
(<A,B,unique.3>).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(<A,B,unique.3>).3,B,unique.2,e[<unique.2, (<A,B,unique.3>).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(<A,B,unique.3>).3,B, unique.2,
e[<unique.2,(<A,B,unique.3>).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1))
```

X: **assume**(*AS.kb* == *B.kb*);

```
TRUE ∧ (B == (d[e[<(<A,B,unique.3>).3,B,unique.2,e[<unique.2,
(<A,B,unique.3>).1>] AS.kb>]AS.ka] A.ka).2) ∧ (unique.3 ==
(d[e[<(<A,B,unique.3>).3, B,unique.2, e[<unique.2,(<A,B,unique.3>).1>]
AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 == d[(d[e[unique.1]
(d[(d[e[<(<A,B,unique.3>).3,B, unique.2, e[<unique.2,
(<A,B,unique.3>).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(<A,B,unique.3>).3,B,unique.2,e[<unique.2, (<A,B,unique.3>).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(<A,B,unique.3>).3,B, unique.2,
e[<unique.2,(<A,B,unique.3>).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)
```

A: *i* := *new*;

```
TRUE ∧ (B == (d[e[<(<A,B,A.i>).3,B,unique.2,e[<unique.2,(<A,B,A.i>).1>]
AS.kb>]AS.ka] A.ka).2) ∧ (A.i == (d[e[<(<A,B,A.i>).3, B,unique.2,
e[<unique.2,(<A,B,A.i>).1>] AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 ==
d[(d[e[unique.1](d[(d[e[<(<A,B,A.i>).3,B, unique.2,
e[<unique.2,(<A,B,A.i>).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(<A,B,A.i>).3,B,unique.2,e[<unique.2, (<A,B,A.i>).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(<A,B,A.i>).3,B, unique.2,
e[<unique.2,(<A,B,A.i>).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)
```

A: $\Rightarrow I(<A,B,i>)$;

```
TRUE ∧ (B == (d[e[<(I’s input queue).3,B,unique.2,e[<unique.2,(I’s input
queue).1>] AS.kb>]AS.ka] A.ka).2) ∧ (A.i == (d[e[<(I’s input queue).3,
B,unique.2, e[<unique.2,(I’s input queue).1>] AS.kb>]AS.ka]A.ka).1)
 ∧ (B.i-1 == d[(d[e[unique.1](d[(d[e[<(I’s input queue).3,B, unique.2,
e[<unique.2,(I’s input queue).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(I’s input queue).3,B,unique.2,e[<unique.2, (I’s input queue).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(I’s input queue).3,B, unique.2,
e[<unique.2,(I’s input queue).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)
```

I: $\leftarrow (tmp1)$;

```
TRUE ∧ (B == (d[e[<(AS.tmp1).3,B,unique.2,e[<unique.2,(AS.tmp1).1>]
AS.kb>]AS.ka] A.ka).2) ∧ (A.i == (d[e[<(AS.tmp1).3, B,unique.2,
e[<unique.2,(AS.tmp1).1>] AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 ==
d[(d[e[unique.1](d[(d[e[<(AS.tmp1).3,B, unique.2,
e[<unique.2,(AS.tmp1).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(AS.tmp1).3,B,unique.2,e[<unique.2, (AS.tmp1).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(AS.tmp1).3,B, unique.2,
e[<unique.2,(AS.tmp1).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)
```

I: $\Rightarrow AS(tmp1)$;

```
TRUE ∧ (B == (d[e[<(AS’s input queue).3,B,unique.2,e[<unique.2,(AS’s
input queue).1>] AS.kb>]AS.ka] A.ka).2) ∧ (A.i == (d[e[<(AS’s input
queue).3, B,unique.2, e[<unique.2,(AS’s input queue).1>]
AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 == d[(d[e[unique.1](d[(d[e[<(AS’s input
queue).3,B, unique.2, e[<unique.2,(AS’s input
queue).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1] (d[e[<(AS’s input
queue).3,B,unique.2,e[<unique.2, (AS’s input queue).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(AS’s input queue).3,B, unique.2,
e[<unique.2,(AS’s input queue).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)
```

AS: $\leftarrow (msg)$;

```
TRUE ∧ (B == (d[e[<(AS.msg).3,B,unique.2,e[<unique.2,(AS.msg).1>]
AS.kb>]AS.ka] A.ka).2) ∧ (A.i == (d[e[<(AS.msg).3,B,unique.2,
e[<unique.2,(AS.msg).1>] AS.kb>]AS.ka]A.ka).1) ∧ (B.i-1 ==
d[(d[e[unique.1](d[(d[e[<(AS.msg).3,B,unique.2,e[<unique.2,(AS.msg).1>]
AS.kb>]AS.ka]A.ka).4]B.kb).1] (d[e[<(AS.msg).3,B,unique.2,e[<unique.2,
(AS.msg).1>]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(AS.msg).3,B,unique.2,
e[<unique.2,(AS.msg).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)
```

AS: *ck* := *new*;

```
TRUE ∧ (B == (d[e[<(AS.msg).3,B,AS.ck,e[<AS.ck,(AS.msg).1>]AS.kb>]AS.ka]
A.ka).2) ∧ (A.i == (d[e[<(AS.msg).3,B,AS.ck,e[<AS.ck,(AS.msg).1>]
AS.kb>]AS.ka]A.ka).1) ∧ (B.i-1 == d[(d[e[unique.1](d[(d[e[<(AS.msg).3,
B,AS.ck,e[<AS.ck,(AS.msg).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(AS.msg).3,B,AS.ck,e[<AS.ck,(AS.msg).1>]AS.kb>]AS.ka]A.ka).3)-1]
(d[(d[e[<(AS.msg).3,B,AS.ck,e[<AS.ck,(AS.msg).1>]AS.kb>]AS.ka]A.ka).4]B
.kb).1)
```

AS: *ticket* := e[<*ck*, *msg.1*>]*kb*;

TRUE ∧ (B == (d[e[<(AS.msg).3,B,AS.ck,AS.ticket>]AS.ka]A.ka).2) ∧ (A.i == (d[e[<(AS.msg).3,B,AS.ck,AS.ticket>]AS.ka]A.ka).1) ∧ (B.i-1 == d[(d[e[unique.1](d[(d[e[<(AS.msg).3,B,AS.ck,AS.ticket>]AS.ka]A.ka).4]B.kb).1](d[e[<(AS.msg).3,B,AS.ck,AS.ticket>]AS.ka]A.ka).3)-1] (d[(d[e[<(AS.msg).3,B,AS.ck,AS.ticket>]AS.ka]A.ka).4]B.kb).1)

AS: ⇒ *I* (e[<*msg.3,B,ck,ticket*>]*ka*);

TRUE ∧ (B == (d[I's input queue]A.ka).2) ∧ (A.i == (d[I's input queue]A.ka).1) ∧ (B.i-1 == d[(d[e[unique.1](d[(d[I's input queue]A.ka).4]B.kb).1](d[I's input queue]A.ka).3)-1] (d[(d[I's input queue]A.ka).4]B.kb).1)

I: ← (*tmp*2);

TRUE ∧ (B == (d[I.tmp2]A.ka).2) ∧ (A.i == (d[I.tmp2]A.ka).1) ∧ (B.i-1 == d[(d[e[unique.1](d[(d[I.tmp2]A.ka).4]B.kb).1](d[I.tmp2]A.ka).3)-1] (d[(d[I.tmp2]A.ka).4]B.kb).1)

I: ⇒ *A* (tmp2);

TRUE ∧ (B == (d[A's input queue]A.ka).2) ∧ (A.i == (d[A's input queue]A.ka).1) ∧ (B.i-1 == d[(d[e[unique.1](d[(d[A's input queue]A.ka).4]B.kb).1](d[A's input queue]A.ka).3)-1] (d[(d[A's input queue]A.ka).4]B.kb).1)

A: ← (*msg*);

TRUE ∧ (B == (d[A.msg]A.ka).2) ∧ (A.i == (d[A.msg]A.ka).1) ∧ (B.i-1 == d[(d[e[unique.1](d[(d[A.msg]A.ka).4]B.kb).1](d[A.msg]A.ka).3)-1] (d[(d[A.msg]A.ka).4]B.kb).1)

A: *tmp* := d[*msg*]*ka*;

TRUE ∧ (B == (A.tmp).2) ∧ (A.i == (A.tmp).1) ∧ (B.i-1 == d[(d[e[unique.1] (d[(A.tmp).4]B.kb).1](A.tmp).3)-1](d[(A.tmp).4]B.kb).1)

A: **assert**(*i* == *tmp.1*);

TRUE ∧ (B == (A.tmp).2) ∧ (B.i-1 == d[(d[e[unique.1](d[(A.tmp).4]B.kb).1] (A.tmp).3)-1](d[(A.tmp).4]B.kb).1)

A: **assert**(*B* == *tmp.2*);

TRUE ∧ (B.i-1 == d[(d[e[unique.1](d[(A.tmp).4]B.kb).1](A.tmp).3)-1] (d[(A.tmp).4]B.kb).1)

A: *ck* := *tmp.3*;

TRUE ∧ (B.i-1 == d[(d[e[unique.1](d[(A.tmp).4]B.kb).1]A.ck)-1] (d[(A.tmp).4]B.kb).1)

A: *ticket* := *tmp.4*;

TRUE ∧ (B.i-1 == d[(d[e[unique.1](d[A.ticket]B.kb).1]A.ck)-1] (d[A.ticket]B.kb).1)

A: $\Rightarrow I$ (*ticket*);

TRUE ∧ (B.i-1 == d[(d[e[unique.1](d[I's input queue]B.kb).1]A.ck)-1] (d[I's input queue]B.kb).1)

I: $\leftarrow (tmp3)$;

TRUE ∧ (B.i-1==d[(d[e[unique.1](d[I.tmp3]B.kb).1]A.ck)-1] (d[I.tmp3]B.kb).1)

I: $\Rightarrow B$ (tmp3);

TRUE ∧ (B.i-1 == d[(d[e[unique.1](d[B's input queue]B.kb).1]A.ck)-1](d[B's input queue]B.kb).1)

B: $\leftarrow (msg)$;

TRUE ∧ (B.i-1 == d[(d[e[unique.1](d[B.msg]B.kb).1]A.ck)-1] (d[B.msg] B.kb).1)

B: *ck* := (d[*msg*]*kb*).*1*;

TRUE ∧ (B.i-1 == d[(d[e[unique.1]B.ck]A.ck)-1]B.ck)

B: *i* := *new*;

TRUE ∧ (B.i-1 == d[(d[e[B.i]B.ck]A.ck)-1]B.ck)

B: $\Rightarrow I$ (e[*i*]*ck*);

TRUE ∧ (B.i-1 == d[(d[I's input queue]A.ck)-1]B.ck)

I: $\leftarrow (tmp4)$

TRUE ∧ (B.i-1 == d[(d[I.tmp4]A.ck)-1]B.ck)

I: $\Rightarrow A$ (tmp4);

TRUE ∧ (B.i-1 == d[(d[A's input queue]A.ck)-1]B.ck)

A: $\leftarrow (msg2)$;

TRUE ∧ (B.i-1 == d[(d[A.msg2]A.ck)-1]B.ck)

A: *i2* := d[*msg2*]*ck*;

TRUE ∧ (B.i-1 == d[A.i2-1]B.ck)

A: $\Rightarrow I$ (*i2-1*);

TRUE ∧ (B.i-1 == d[I's input queue]B.ck)

I: $\leftarrow (tmp5)$;

TRUE ∧ (B.i-1 == d[I.tmp5]B.ck)

I: $\Rightarrow B$ *(tmp5);*

TRUE ∧ (B.i-1 == d[B's imput queue]B.ck)

B: $\leftarrow (msg2)$;

TRUE ∧ (B.i-1 == d[B.msg2]B.ck)

B: **assert**(*i-1* == d[*msg2*]*ck*)

TRUE

The process of generating these weakest precondition predicates is easy to automate since all of the transformations involve simple textual substitutions.

The weakest precondition for the first statement in the protocol is referred to as the *verification condition* since proof of this theorem proves that the protocol will achieve its stated goals if only the actions given in the specification are performed. The verification condition for the Needham and Schroeder protocol that we derived above is:

```
(AS.ka == A.ka) ⊃ ((AS.kb == B.kb) ⊃ (TRUE ∧ (B == (d[e[<(<A,B,unique.3>).3,
B,unique.2, e[<unique.2, (<A,B,unique.3>).1>] AS.kb>]AS.ka]
A.ka).2) ∧ (unique.3 == (d[e[<(<A,B,unique.3>).3, B,unique.2,
e[<unique.2,(<A,B,unique.3>).1>] AS.kb>]AS.ka]A.ka).1)  ∧ (B.i-1 ==
d[(d[e[unique.1] (d[(d[e[<(<A,B,unique.3>).3,B, unique.2, e[<unique.2,
(<A,B,unique.3>).1>]AS.kb>]AS.ka]A.ka).4]B.kb).1]
(d[e[<(<A,B,unique.3>).3,B,unique.2,e[<unique.2, (<A,B,unique.3>).1>
]AS.kb>]AS.ka]A.ka).3)-1](d[(d[e[<(<A,B,unique.3>).3,B, unique.2,
e[<unique.2,(<A,B,unique.3>).1>]AS.kb>] AS.ka]A.ka).4]B.kb).1)))
```

Fortunately, much automatic simplification of this predicate is possible. For example, we can textually substitute “`A`” for “`(<A, B, unique.3>).1`” and “`unique.3`” for “`(<A, B, unique.3>).3`” in the above theorem to yield:

```
(AS.ka == A.ka) ⊃ ((AS.kb == B.kb) ⊃ (TRUE ∧ (B == (d[e[<A, B, unique.2,
e[<unique.2, A>] AS.kb>]AS.ka] A.ka).2) ∧ (unique.3 == (d[e[<unique.3,
B,unique.2, e[<unique.2,A>] AS.kb>]AS.ka] A.ka).1)  ∧ (B.i-1 ==
d[(d[e[unique.1] (d[(d[e[<unique.3,B, unique.2, e[<unique.2,
A>]AS.kb>]AS.ka]A.ka).4]B.kb).1] (d[e[<unique.3,B,unique.2, e[<unique.2,
A> ]AS.kb>]AS.ka] A.ka).3)-1](d[(d[e[<unique.3,B, unique.2,
e[<unique.2,A>] AS.kb>] AS.ka]A.ka).4]B.kb).1)))
```

This theorem can be further simplified using other substitutions and identity relations to the point that it can be proven manually. By proving the verification condition we have proven that this particular trace of the protocol does satisfy the stated goals. While this is an important result, it does not prove that the protocol is secure. If we were to replace one or more of the intruder’s benign actions with other actions, we would need to

recompute the weakest precondition for that trace of the protocol and attempt to prove the new verification condition. This technique is used to analyze a number of protocols in [YAS96].

Appendix B

Parallel Session Fragments

In this appendix we list all the parallel session fragments for a sample cryptographic protocol. The protocol is given below in Figure B.1.

(1) *X*: **assume**(*Inverse(B.pub_B, B.priv_B));*

(2) *X*: **assume**(*A.pub_B == B.pub_B);*

(3) *X*: **assume**(*I.pub_B == B.pub_B);*

(4) *B*: *x* := *new*;

(5) *B*: $\Rightarrow I$ *(ep[x]priv_B)*;

(6) *I*: $\leftarrow (tmp1)$;

(7) *I*: $\Rightarrow A$ *(tmp1)*;

(8) *A*: $\leftarrow (msg)$;

(9) *A*: *x* := *dp[msg]pub_B*;

Figure B.1 : The Sample Protocol

This protocol contains 3 "turns" as illustrated below.

Turn 1: *X*: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

Turn 2: *I*: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Turn 3: *B*: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_A*;

Figure B.2 : The Sample Protocol Divided into Turns

This yields 3 valid prefixes of the protocol (recall that a valid prefix of a protocol must start with the first statement of the protocol and end on a turn boundary):

Prefix 1:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

Prefix 2:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Prefix 3:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

B: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_A*;

Prefix 1 gives us 3 distinct parallel session fragments (since agent *A*, *B*, or *I* can play the role of "A" in Prefix 1). They are:

Parallel Session 1 (*A* plays "A"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

Parallel Session 2 (*B* plays "A"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow I$ *(ep[x]priv_B)*;

Parallel Session 3 (*I* plays "A"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow I$ *(ep[x]priv_I)*;

Prefix 2 gives us 3 more parallel session fragments (since agent *A*, *B*, or *I* can play the role of "A" and only the intruder can play the role of "I"):

Parallel Session 4 (*A* plays "A" and *I* plays "I"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Parallel Session 5 (*B* plays "A" and *I* plays "I"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow I$ *(ep[x]priv_B)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

Parallel Session 6 (*I* plays "A" and *I* plays "I"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A)*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow$ *I (ep[x]priv_I)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow$ *B (tmp1)*;

Finally, Prefix 3 gives us 9 more parallel session fragments (since agent *A*, *B*, or *I* can play the role of "A", only *I* can play the role of "I", and agent *A*, *B*, or *I* can play the role of "B"):

Parallel Session 7 (*A* plays "A", *I* plays "I", and *A* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow$ *I (ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow$ *A (tmp1)*;

A: $\leftarrow (msg)$;

A: *x* := *dp[msg]pub_A*;

Parallel Session 8 (*A* plays "A", *I* plays "I", and *B* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow$ *I (ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow$ *B (tmp1)*;

B: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_A*;

Parallel Session 9 (*A* plays "A", *I* plays "I", and *I* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *x* := *new*;

A: $\Rightarrow I$ *(ep[x]priv_A)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow I$ *(tmp1)*;

I: $\leftarrow (msg)$;

I: *x* := *dp[msg]pub_A*;

Parallel Session 10 (*B* plays "A", *I* plays "I", and *A* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow I$ *(ep[x]priv_B)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow A$ *(tmp1)*;

A: $\leftarrow (msg)$;

A: *x* := *dp[msg]pub_B*;

Parallel Session 11 (*B* plays "A", *I* plays "I", and *B* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow I$ *(ep[x]priv_B)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

B: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_B*;

Parallel Session 12 (*B* plays "A", *I* plays "I", and *I* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

B: *x* := *new*;

B: $\Rightarrow I$ *(ep[x]priv_B)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow I$ *(tmp1)*;

I: $\leftarrow (msg)$;

I: *x* := *dp[msg]pub_B*;

Parallel Session 13 (*I* plays "A", *I* plays "I", and *A* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow I$ *(ep[x]priv_I)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow A$ *(tmp1)*;

A: $\leftarrow (msg)$;

A: *x* := *dp[msg]pub_I*;

Parallel Session 14 (*I* plays "A", *I* plays "I", and *B* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow I$ *(ep[x]priv_I)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow B$ *(tmp1)*;

B: $\leftarrow (msg)$;

B: *x* := *dp[msg]pub_I*;

Parallel Session 15 (*I* plays "A", *I* plays "I", and *I* plays "B"):

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

I: *x* := *new*;

I: $\Rightarrow I$ *(ep[x]priv_I)*;

I: $\leftarrow (tmp1)$;

I: $\Rightarrow I$ *(tmp1)*;

I: $\leftarrow (msg)$;

I: *x* := *dp[msg]pub_I*;

Each parallel session fragment can be used to generate an axiom for the automatic theorem prover as discussed in Chapter 5. Deleting some "useless" parallel session fragments and renaming the variables leaves us with the following 10 parallel session fragments:

Parallel Session 1:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *tmp1* := *new*;

Parallel Session 2:

X: **assume**(*Inverse(B.pub_B, B.priv_B));*

X: **assume**(*B.pub_B == B.pub_B);*

X: **assume**(*I.pub_B == B.pub_B);*

B: *tmp1* := *new*;

Parallel Session 3:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *tmp2* := *new*;

A: $\Rightarrow I$ *(ep[tmp2]priv_A)*;

I: $\leftarrow (tmp2)$;

Parallel Session 4:

X: **assume**(*Inverse(B.pub_B, B.priv_B));*

X: **assume**(*B.pub_B == B.pub_B);*

X: **assume**(*I.pub_B == B.pub_B);*

B: *tmp2* := *new*;

B: $\Rightarrow I$ *(ep[tmp2]priv_B)*;

I: $\leftarrow (tmp3)$;

Parallel Session 5:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*A.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *tmp3* := *new*;

A: $\Rightarrow I$ *(ep[tmp3]priv_A)*;

I: $\leftarrow (tmp4)$;

I: $\Rightarrow A$ *(tmp4)*;

A: $\leftarrow (msg)$;

A: tmp4 := dp[msg]pub_A;

Parallel Session 6:

X: **assume**(*Inverse(A.pub_A, A.priv_A));*

X: **assume**(*B.pub_A == A.pub_A);*

X: **assume**(*I.pub_A == A.pub_A);*

A: *tmp5* := *new*;

A: $\Rightarrow I$ *(ep[tmp5]priv_A)*;

I: $\leftarrow (tmp5)$;

I: $\Rightarrow B$ *(tmp5)*;

B: $\leftarrow (tmp3)$;

B: tmp4 := dp[tmp3]pub_A;

Parallel Session 7:

X: **assume**(*Inverse(I.pub_I, I.priv_I));*

X: **assume**(*A.pub_I == I.pub_I);*

X: **assume**(*I.pub_I == I.pub_I);*

I: $\Rightarrow A$ *(tmp6)*;

A: $\leftarrow (tmp6)$;

A: tmp7 := dp[tmp6]pub_I;

Parallel Session 8:

X: **assume**(*Inverse(I.pub_I, I.priv_I));*

X: **assume**(*B.pub_I == I.pub_I);*

X: **assume**(*I.pub_I == I.pub_I);*

I: $\Rightarrow B$ *(tmp7)*;

B: $\leftarrow (tmp4)$;

B: tmp5 := dp[tmp4]pub_I;

Parallel Session 9:

X: **assume**(*Inverse(B.pub_B, B.priv_B));*

X: **assume**(*A.pub_B == B.pub_B);*

X: **assume**(*I.pub_B == B.pub_B);*

B: *tmp6* := *new*;

B: $\Rightarrow I$ *(ep[tmp6]priv_B)*;

I: $\leftarrow (tmp8)$;

I: $\Rightarrow A$ *(tmp8)*;

A: $\leftarrow (tmp7)$;

A: tmp8 := dp[tmp7]pub_B;

Parallel Session 10:

X: **assume**(*Inverse(B.pub_B, B.priv_B));*

X: **assume**(*B.pub_B == B.pub_B);*

X: **assume**(*I.pub_B == B.pub_B);*

B: *tmp7* := *new*;

B: $\Rightarrow I$ *(ep[tmp7]priv_B)*;

I: $\leftarrow (tmp9)$;

I: $\Rightarrow B$ *(tmp9)*;

B: $\leftarrow (tmp8)$;

B: tmp9 := dp[tmp8]pub_B;

References

[1] Abadi, Martin, and Needham, Roger, “Prudent Engineering Practice for Cryptographic Protocols”, *Proceedings of the 1994 IEEE Symposium On Research in Security and Privacy*, pp. 122-136.

[2] Abadi, Martin, and Tuttle, M.R., “A Semantics for a Logic of Authentication”, *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, August 1991, pp. 201-216.

[3] Balzer, Robert, “A 15 Year Perspective on Automatic Programming”, *IEEE Transactions on Software Engineering*, Volume SE-11, Number 11, November 1985, pp. 1257-1277.

[4] Bauer, R.K., Berson, T.A., and Friertag, R.J., “A Key Distribution Protocol using Event Markers”, *ACM Transactions of Computer Systems*, Volume 1, Number 3, August 1983, pp. 249-255.

[5] Beller, M.J., Chang, Li-Fung, and Yacobi, Y., “Privacy and Authentication on a Portable Communication System”, *IEEE Journal on Selected Areas in Communications*, Volume 11, Number 6, August 1993, pp. 821-829.

[6] Bellovin, Steven M., and Merritt, Michael, “Limitations of the Kerberos Authentication System”, *Computer Communication Review*, Volume 20, Number 5, October 1990, pp. 119-132.

[7] Bellovin, Steven M., and Merritt, Michael, “Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks”, *Proceedings of the 1992 IEEE Symposium On Research in Security and Privacy*, pp. 72-84.

[8] Benaloh, Josh, and Tuinstra, Dwight, “Receipt-Free Secret-Ballot Elections”, *Proceedings of the 26th Annual ACM Symposium on the Theory of Computation*, 1994, pp. 544-553.

[9] Bieber, Pierre, “A Logic of Communication in Hostile Environment”, *Proceedings of the Computer Security Foundations Workshop III*, 1990, pp. 14-22.

[10] Bieber, Pierre, Boulahia-Cuppens, Nora, Lehmann, Thomas, and van Wickeren, Erich, "Abstract Machines for Communication Security", *Proceedings of the Computer Security Foundations Workshop VI*, 1993, pp. 137-146.

[11] Biermann, A.W., "Approaches to Automatic Programming", *Advances in Computers*, Volume 15, 1976, pp. 1-63.

[12] Biham, Eli, and Shamir, Adi, "Differential Cryptanalysis of DES-like Cryptosystems", *Advaces in Cryptology - CRYPTO '90*, pp. 2-21.

[13] Bird, R., et al, "Systematic Design of two-party authentication protocols", *Advances in Cryptology - CRYPTO '91*, pp. 44-61.

[14] Bird, Ray, Gopal, I., Herzberg, Amir, Janson, Philippe A., Kutten, Shay, Molva, Refik, and Yung, Moti, "Systematic Design of a Family of Attack-Resistant Authentication Protocols", *IEEE Journal on Selected Areas in Communications*, Volume 11, Number 5, June 1993, pp. 679-693.

[15] Boyd, Colin, and Mao, Wenbo, "On a Limitation of BAN Logic", *Advances in Cryptology - EYROCRYPT '93*, pp. 240-247.

[16] Boyd, Colin, "Hidden Assumptions in Crytpographic Protocols", *Proceedings of the IEE*, Part E, Volume 137, Number 6, November 1990, pp. 433-436.

[17] Boyd, Colin, and Mao, Wenbo, "Limitations of Logical Analysis of Cryptographic Protocols", -,

[18] Burns, J., and Mitchell, C. J., "A Security Scheme for Resource Sharing Oer a Network", *Computers and Security*, Volume 19, February, 1990, pp. 67-76.

[19] Burrows, Michael, Abadi, Martin, and Needham, Roger, "A Logic of Authentication", *ACM Operating Systems Review*, Volume 23, Number 5, 1989, pp. 1-13.

[20] Burrows, Michael, Abadi, Martin, and Needham, Roger, "Rejoiner to Nessett", *ACM Operating Systems Review*, Volume 24, Number 2, April, 1990, pp. 39-40.

[21] Calvelli, Claudio, and Varadharajan, Vijay, "An Analysis of Some Delegation Protocols for Distributed Systems", *Proceedings of the Computer Security Foundations Workshop V*, 1992, pp. 92-110.

[22] Campbell, E. A., Safavi-Naini, R., and Pleasants, P. A., "Partial Belief and Probabilistic Reasoning in the Analysis of Secure Protocols", *Proceedings of the Computer Security Foundations Workshop V*, 1992, pp. 84-91.

[23] Carlsen, Ulf, "Using Logics to Detect Implementation-Dependent Flaws", *Proceedings of the 9th Annual Computing Security Applications Conference*, December 1993, pp. 64-73.

[24] Carlsen, Ulf, "Cryptographic Protocol Flaws: Know Your Enemy", *Proceedings of the Computer Security Foundations Workshop VII*, 1994, pp. 192-200.

[25] Carlsen, Ulf, "Generating Formal Cryptographic Protocol Specifications", *Proceedings of the 1994 IEEE Symposium On Research in Security and Privacy*, pp. 137-146.

[26] Carnap, R., “The Logical Syntax of Language”, translated by A. Smeaton, London (Routledge and Kegan Paul), 1937.

[27] Cheng, Pau-Chen, and Gligor, Virgil D., “On the Formal Specification and Verification of a Multiparty Session Protocol”, *Proceedings of the 1990 IEEE Symposium On Research in Security and Privacy*, pp. 216-233.

[28] De Santis, Alfredo, Desmedt, Yvo, Frankel, Yair, and Yung, Moti, “How to Share A Function Securely”, *Proceedings of the 26th Annual ACM Symposium on the Theory of Computation*, 1994, pp. 522-533.

[29] DeMillo, R., Lipton, R., and Perlis, A., “Social Processes and Proofs of Theorems and Programs”, *Communications of the ACM*, Volume 22, Number 5, May, 1979, pp. 271-280.

[30] DeMillo, R., Lynch, N., and Merritt, M., “Cryptographic Protocols”, *Proceedings of the 14th ACM Symposium on Theory of Computing*, May 1982, pp. 383-400.

[31] DeMillo, R., and Merritt, M., “Protocols for Data Security”, *Computer*, Volume 16, February 1983, pp. 39-50.

[32] Denning, Dorothy E., and Sacco, Giovanni Maria, “Timestamps in Key Distribution Protocols”, *Communications of the ACM*, Volume 24, Number 8, August 1981, pp. 533-536.

[33] “Department of Defense trusted computer systems evaluation criteria”, National Computer Security Center, DOD 5200.28-STD, December, 1985.

[34] Diffie, W., and Hellman, M.E., “New Directions in Cryptography”, *IEEE Transactions on Information Theory*, Volume IT-11, November 1976, pp. 644-654.

[35] Diffie, W., von Oorschat, P.C., and Wiener, M.J., “Authentication and Authenticated Key Exchanges”, *Designs, Codes, and Cryptography*, Volume 2, 1992, pp. 107-125.

[36] Dolev, Danny, and Yao Andrew C., “On the Security of Public Key Protocols”, *IEEE Transactions on Information Theory*, Volume IT-29, Number 2, March 1983, pp. 198-208.

[37] ElGamal, T., “A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms”, *IEEE Transactions on Information Theory*, Volume IT-31, July 1985, pp. 469-472.

[38] Feige, U., Fiat, A., and Shamir, A., “Zero Knowledge Proofs of Identity”, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987, pp. 210-217.

[39] Feige, Uri, Kilian, Joe, and Naor, Moni, “A Minimal Model for Secure Computation”, *Proceedings of the 26th Annual ACM Symposium on the Theory of Computation*, 1994, pp. 554-563.

[40] Gaarder, K., and Snekkenes, E., “Applying a Formal Analysis Technique to the CCITT X.509 Strong Two-Way Authentication Protocol”, *Journal of Cryptology*, Volume 3, 1991, pp. 81-98.

[41] Goldreich, Oded, Micali, Silvio, and Wigderson, Avi, “Proofs that Yield Nothing But

their Validity and a Methodology of Cryptographic Protocol Design", *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986, pp. 174-187.

[42] Goldwasser, Shafi, and Micali, Silvio, "Probabilistic Encryption", *Journal of Computer and System Sciences*, Volume 28, 1984, pp. 270-299.

[43] Gong, Li, "Using one-way functions for Authentication", *ACM Computer Communications Review*, Volume 19, Number 5, October 1989, pp. 8-11.

[44] Gong, Li, "A Note on Redundancy in Encrypted Messages", *ACM Computer Communications Review*, Volume 20, October 1990, pp. 18-22.

[45] Gong, Li, "Verifiable-text attacks in Cryptographic Protocols", *Proceedings of the IEEE INFOCOM '90*, pp. 686-693.

[46] Gong, Li, Needham, Roger, and Yahalom, Raphael, "Reasoning about Belief in Cryptographic Protocols", *Proceedings of the 1990 IEEE Symposium On Research in Security and Privacy*, pp. 234-248.

[47] Gong, Li, "A Security Risk of Depending on Synchronized Clocks", *Operating Systems Review*, Volume 26, Number 1, January 1992, pp. 49-53.

[48] Gong, Li, "Variations on the Themes of Message Freshness and Replay - or the Difficulty in Devising Formal Methods to Analyze Cryptographic Protocols", *Proceedings of the Computer Security Foundations Workshop VI*, 1993, pp. 131-136.

[49] Greene, C., and Barstow, D., "Some rules for the Automatic Synthesis of programs", *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, 1975, pp. 232-239.

[50] Harn, Lein, and Kiesler, Thomas, "Authenticated Group Key Distribution Scheme For A Large Distributed Network", *Proceedings of the 1989 IEEE Symposium On Research in Security and Privacy*, pp. 300-309.

[51] Heintze, Nevin, and Tygar, J. D., "A Model for Secure Protocols and Their Compositions", *Proceedings of the 1994 IEEE Symposium On Research in Security and Privacy*, pp. 2-13.

[52] I'Anson, C., and Mitchell, C., "Security Defects in CCITT Recommendation X.509", *ACM Computer Communications Review*, Volume 20, Number 2, April 1990, pp. 30-34.

[53] ISO/IEC Draft International Standard 10181-2.2, *Information Technology - Open Systems Interconnection - Security Frameworks for Open Systems: Authentication Framework*, 1993.

[54] Kailar, Rajashekar, and Gligor, Virgil D., "On Belief Evolution in Authentication Protocols", *Proceedings of the 1991 IEEE Symposium On Research in Security and Privacy*, pp. 103-116.

[55] Kao, I-Lung, and Chow, Randy, "An Efficient and Secure Authentication Protocol Using Uncertified Keys", -, pp. 14-21.

[56] Kehne, A., Schonwalder, J., and Langendorfer, H., "A Nonce-Based Protocol For Mul-

tiple Authentications", *Operating Systems Review*, Volume 26, Number 4, October 1992, pp. 84-89.

[57] Kemmerer, Richard A., "Analyzing Encryption Protocols Using Formal Verification Techniques", *IEEE Journal on Selected Areas in Communication*, Volume 7, Number 4, May 1989, pp. 448-457.

[58] Kemmerer, R., Meadows, C., and Millen, J., "Three Systems for Cryptographic Protocol Analysis", *Journal of Cryptology*, Volume 7, Number 2, 1994, pp.79-130.

[59] Kessler, Volker, and Wedel, Gabriele, "AUTLOG - An advanced logic of authentication", *Proceedings of the Computer Security Foundations Workshop VII*, 1994, pp. 90-99.

[60] Lampson, B., "Protection", *Proceedings of the 5th Symposium on Operating Systems*, January 1974.

[61] Lampson, B., Abadi, M., Burrows, M., and Wobber, E., "Authentication in Distributed Systems: Theory and Practice", *ACM Transactions on Computer Systems*, Volume 10, Number 4, November 1992, pp. 265-310.

[62] Lee, R.C.T., Chang, C.L., and Waldinger, R.J., "An Improved Program-Synthesizing Algorithm and Its Correctness", *Communications of the ACM*, Volume 17, Number 4, April 1974, pp. 211-217.

[63] Liebl, Armin, "Authentication in Distributed Systems: A Bibliography", *Operating Systems Review*, Volume 27, Number 4, October 1993, pp. 31-41.

[64] Lin, C. H., Chang, C. C., and Lee, R. C. T., "A New Public-Key Cipher System Based Upon the Diophantine Equations", *IEEE Transactions on Computers*, Volume 44, Number 1, January 1995, pp. 13-18.

[65] Lomsa, T.M.A., Gong, L., Saltzer, J.H., and Needham, R.M., "Reducing Risks from Poorly Chosen Keys", *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989, pp. 14-18.

[66] Longley, D., and Rigby, S., "An Automatic Search for Security Flaws in Key Management Schemes", *Computers and Security*, Volume 11, 1992, pp. 75-89.

[67] Lowe, Gavin, "Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR".

[68] Lu, W.P., and Sundereshan, M.K., "Secure Communication in Internet Environments: A Hierarchical Key Management Scheme for End-to-End Encryption", *IEEE Transactions on Communications*, Volume 37, Number 10, October 1989, pp. 1014-1023.

[69] Lu, W.P., and Sundereshan, M.K., "Enhanced Protocols for Hierarchical Encryption Key Management", *IEEE Transactions on Communications*, Volume 40, Number 4, April 1992, pp. 658-660.

[70] Manna, Zohar, and Waldinger, Richard J., "Towards Automatic Program Synthesis", *Communications of the ACM*, Volume 14, Number 3, March 1971, pp. 151-165.

[71] Manna, Zohar, and Waldinger, Richard, "Fundamentals of Deductive Program Synthe-

sis", *IEEE Transactions on Software Engineering*, Volume 18, Number 8, August 1992, pp. 674-704.

[72] Mao, Wenbo, and Boyd, Colin, "Towards Formal Analysis of Security Protocols", *Proceedings of the Computer Security Foundations Workshop VI*, 1993, pp. 147-158.

[73] Mao, Wenbo, and Boyd, Colin, "Development of Authentication Protocols: Some Misconceptions and a New Approach", *Proceedings of the Computer Security Foundations Workshop VII*, 1994, pp. 178-186.

[74] Massey, J.L., "An Introduction to Contemporary Cryptology", *Proceedings of the IEEE*, Volume 76, Number 5, May 1988, pp. 533-549.

[75] Meadows, Catherine, "Using Narrowing in the Analysis of Key Management Protocols", *Proceedings of the 1989 IEEE Symposium On Research in Security and Privacy*, pp. 138-147.

[76] Meadows, Catherine, "A System for the Specification and Analysis of Key Management Protocols", *Proceedings of the 1991 IEEE Symposium On Research in Security and Privacy*, pp. 182-195.

[77] Meadows, Catherine, "A Model of Computation for the NRL Protocol Analyzer", *Proceedings of the Computer Security Foundations Workshop VII*, 1994, pp. 84-89.

[78] Meadows, Catherine, "Formal Verification of Cryptographic Protocols: A Survey", *Advances in Cryptology - ASIACRYPT 1994*, pp. 135-150.

[79] Merritt, M. J., "Cryptographic Protocols", PhD Thesis, Georgia Institute of Technology, 1983.

[80] Micali, Silvio, "Fair Public-Key Cryptosystems".

[81] Millen, Jonathan K., Clark, Sidney C., and Freedman, Sheryl B., "The Interrogator: Protocol Security Analysis", IEEE Transactions on Software Engineering, Volume SE-13, Number 2, February 1987, pp. 274-288.

[82] Millen, Jonathan K., "The Interrogator Model", *Proceedings of the 1995 IEEE Symposium On Research in Security and Privacy*, pp. 251-260.

[83] Millen, Jonathan K., "CAPSL: Common Authentication Protocol Specification Language", http://www.mitre.org/research/capsl

[84] Molva, Refik, Tsudik, Gene, Van Herreweghen, Els, and Zatti, Stefano, "*KryptoKnight* Authentication and Key Distribution System", *Proceedings of the 1992 European Symposium on Research in Computer Security - ESORICS 1992*, pp. 1-16.

[85] Molva, Refik, and Tsudik, Gene, "Authentication Method with Impersonal Token Cards", *Proceedings of the 1993 IEEE Symposium On Research in Security and Privacy*, pp. 56-65.

[86] Moore, J.H., "Protocol Failures in Cryptosystems", *Proceedings of the IEEE*, Volume 76, Number 5, May 1988, pp. 594-602.

[87] Morris, R.H., and Thompson, K., "UNIX password security", *Communications of the*

ACM, Volume 22, Number 11, November 1979, pp. 594-597.

[88] Moser, Louise E., “A Logic of Knowledge and Belief for Reasoning about Computer Security”, *Proceedings of the Computer Security Foundations Workshop II*, 1989, pp. 57-63.

[89] National Bureau of Standards, “Data Encryption Standard”, FIPS Publication 46, January 1977.

[90] Needham, Roger M., and Schroeder, Michael D., “Using Encryption for Authentication in Large Networks of Computers”, *Communications of the ACM*, Volume 21, Number 12, December 1978, pp. 993-999.

[91] Needham, Roger M., and Schroeder, Michael D., “Authentication Revisted”, *Operating Systems Review*, Volume 21, Number 1, January 1987, p. 7.

[92] Nessett, Dan M., “A Critique of the Burrows, Abadi, and Needham Logic”, *Operating Systems Review*, April 1990, pp. 35-38.

[93] Neufeld, G., and Voung, Son, “An Overview of ASN.1”, *Computer Networks and ISDN Systems*, Volume 23, 1992, pp. 393-415.

[94] Neuman, B.C., and Stubblebine, S.G., “A Note on the Use of Timestamps as Nonces”, *Operating Systems Review*, Volume 27, Number 2, April 1993, pp. 10-14.

[95] Otway, Dave, and Rees, Owen, “Efficient and Timely Mutual Authentication”, *Operating Systems Review*, Volume 21, Number 1, January 1987, pp. 8-10.

[96] Piessens, F., De Decker, B., and Janson, P., “Interconnecting Domains with Heterogeneous Key Distribution and Authentication Protocols”, *Proceedings of the 1993 IEEE Symposium On Research in Security and Privacy*, pp. 66-79.

[97] Rangan, P.V., “An Axiomatic Basis of Trust in Distributed Systems”, *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pp. 204-211.

[98] Rich, C., “A Formal Representation for Plans in the Programmer’s Aprentice”, *Proceddings of the 7th International Joint Conference on Artificial Intelligence*, 1981, pp. 1044-1052.

[99] Rich, Charles, and Waters, Richard C., “Automatic Programming: Myths and Prospects”, *IEEE Computer*, August, 1988, pp. 40-51.

[100] Rivest, R. L., Shamir, A., and Adleman, L., “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, *Communications of the ACM*, Volume 21, Number 2, February 1978, pp. 120-126.

[101] Rivest, R., and Shamir, A., “How to expose an eavesdropper”, *Communications of the ACM*, Volume 27, Number 4, April 1984, pp. 393-395.

[102] Rubin, Aviel D., and Honeyman, Peter, “Nonmonotonic Cryptographic Protocols”, *Proceedings of the Computer Security Foundations Workshop VII*, 1994, pp. 100-116.

[103] Satyanarayanan, M., “Integrating Security in a Large Distributed System”, *ACM Transactions on Computer Systems*, Volume 7, Number 3, August 1989,

[104] Shamir, Adi, “How to Share a Secret”, *Communications of the ACM*, Volume 22, Number 11, November 1979, pp. 612-613.

[105] Shor, Peter W., “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124-134.

[106] Simmons, Gustavus J., “How to (Selectively) Broadcast a Secret”, *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, pp. 108-113.

[107] Simmons, Gustavus J., “An Introduction to the Mathematics of Trust in Security Protocols”, *Proceedings of the Computer Security Foundations Workshop VI*, 1993, pp. 121-127.

[108] Simmons, Gustavus J., “Proof of Soundness (Integrity) of Cryptographic Protocols”, *Journal of Cryptology*, Volume 7, 1994, pp. 69-77.

[109] Simon, Daniel R., “On the Power of Quantum Computation”, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 116-123.

[110] Snekkenes, Einar, “Exploring the BAN Approach to Protocol Analysis”, *Proceedings of the 1991 IEEE Symposium On Research in Security and Privacy*, pp. 171-181.

[111] Snekkenes, Einar, “Roles in Cryptographic Protocols”, *Proceedings of the 1992 IEEE Symposium On Research in Security and Privacy*, pp. 105-118.

[112] Steiner, Michael, Tsudik, Gene, Waidner, Michael, “Refinement and Extension of *Encrypted Key Exchange*”, *Operating Systems Review*, Volume 29, Number 3, July 1995, pp. 22-30.

[113] Stubblebine, Stuart G., and Gligor, Virgil D., “On Message Integrity in Cryptographic Protocols”, *Proceedings of the 1992 IEEE Symposium On Research in Security and Privacy*, pp. 85-104.

[114] Stubblebine, Stuart G., and Gligor, Virgil D., “Protocol Design for Integrity Protection”, *Proceedings of the 1993 IEEE Symposium On Research in Security and Privacy*, pp. 41-53.

[115] Syverson, Paul, “Formal Semantics for Logics of Cryptographic Protocols”, *Proceedings of the Computer Security Foundations Workshop III*, 1990, pp. 32-41.

[116] Syverson, Paul, “The Use of Logic in the Analysis of Cryptographic Protocols”, *Proceedings of the 1991 IEEE Symposium On Research in Security and Privacy*, pp. 156-170.

[117] Syverson, Paul, “Knowledge, Belief, and Semantics in the Analysis of Cryptographic Protocols”, *Journal of Computer Security*, Volume 1, Number 3, 1992, pp. 317-334.

[118] Syverson, Paul, and Meadows, Catherine, “A Logical Language for Specifying Cryptographic Protocol Requirements”, *Proceedings of the 1993 IEEE Symposium On Research in Security and Privacy*, pp. 165-177.

[119] Syverson, Paul, “On Key Distribution Protocols for Repeated Authentication”, *Operating Systems Review*, Volume 27, Number 4, October 1993, pp. 24-30.

[120] Syverson, Paul, “Adding Time to a Logic of Authentication”, *Proceedings of the 1st ACM Conference on Computer and Communication Security*, November 1993.

[121] Syverson, Paul, “A Taxonomy of Replay Attacks”, *Proceedings of the Computer Security Foundations Workshop VII*, 1994, pp. 187-191.

[122] Syverson, Paul, and van Oorschot, Paul C., “On Unifying Some Cryptographic Protocol Logics”, *Proceedings of the 1994 IEEE Symposium On Research in Security and Privacy*, pp. 14-28.

[123] Tanenbaum, Andrew S., and van Renesse, Robbert, “Using Sparse Capabilities in a Distributed Operating System”, *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 558-563.

[124] Tatebayashi, M., Matsuzaki, N., and Newman, D. B., “Key Distribution Protocol for Digital Mobile Communications Systems”, *Advances in Cryptology - CRYPTO ‘89*, pp. 324-333.

[125] Thompson, K., “Reflections on Trusting Trust”, *Communications of the ACM*, Volume 27, Number 8, August, 1984, pp. 761-763.

[126] Toussaint, M. J., “Verification of Cryptographic Protocols”, PhD Thesis. Universite de Liege (Belgium), 1991.

[127] Toussaint, M.J., “Deriving the Complete Knowledge of Participants in Cryptographic Protocols”, *Advances in Cryptology - CRYPTO ‘91.*

[128] Varadharajan, V., Allen, P., and Black, S., “An analysis of the proxy problem in distributed systems”, *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pp. 255-275.

[129] Voydock, Victor L., and Kent, Stephen T., “Security in High-Level Network Protocols”, *IEEE Communications Magazine*, Volume 23, Number 7, July 1985, pp. 12-24.

[130] Walker, Stephen T., “Network Security: The Parts of the Sum”, *Proceedings of the 1989 IEEE Symposium On Research in Security and Privacy*, pp. 2-9.

[131] Waters, R.C., “The Programmer’s Apprentice: A Session with KBEacs”, *IEEE Transactions on Software Engineering*, Volume 11, Number 11, November 1985, pp. 1296-1320.

[132] Wobber, E., Abadi, M., Burrows, M., and Lampson, B., “Authentication in the Taos Operating System”, *Proceedings of the 14th ACM Symposium on Operating System Principles,* 1993, pp. 256-269.

[133] Woo, Thomas Y. C., and Lam, Simon S., “Authentication for Distributed Systems”, *Computer*, Volume 25, Number 1, January 1992, pp. 39-52.

[134] Woo, Thomas Y. C., and Lam, Simon S., “A Semantic Model for Authentication Protocols”, *Proceedings of the 1993 IEEE Symposium On Research in Security and Privacy*, pp. 178-194.

[135] Yahalom, R., Klein, B., and Beth, Th., “Trust Relationships in Secure Systems - A Distributed Authentication Perspective”, *Proceedings of the 1993 IEEE Symposium On*

Research in Security and Privacy, pp. 150-164.

[136] Yao, A., “Protocols for Secure Computations”, *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, November 1982, pp. 160-164.

[137] Yasinsac, A.F., “Evaluating Cryptographic Protocols”, Ph.D. Dissertation, University of Virginia, 1996.