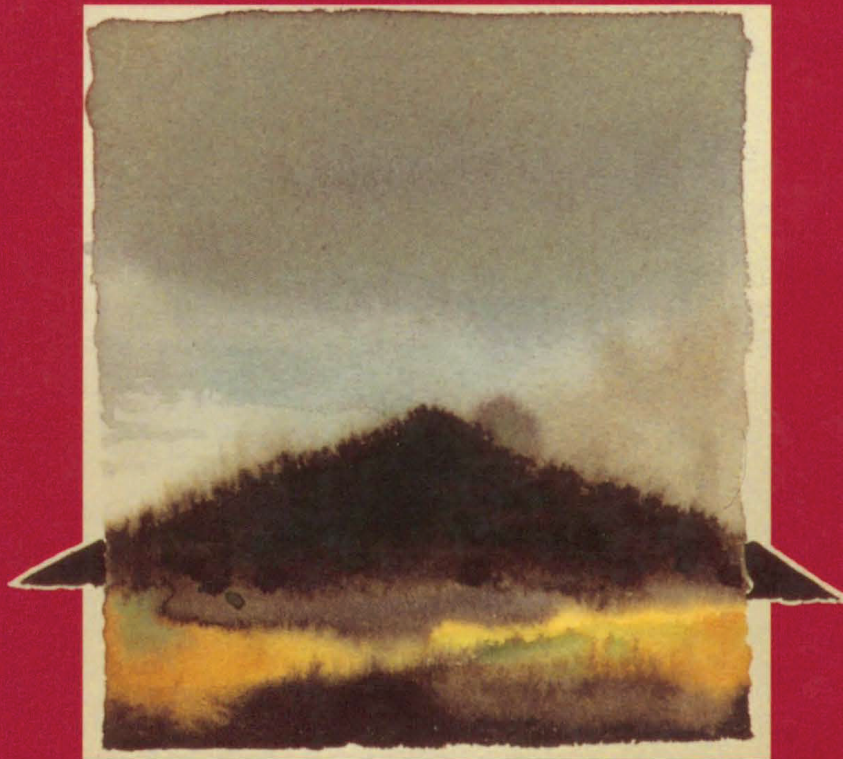


*"... the best introduction
to cryptography I've
ever seen.... The book
the National Security
Agency wanted never
to be published...."*

—Wired Magazine

**SECOND
EDITION**

APPLIED CRYPTOGRAPHY



**Protocols, Algorithms,
and Source Code in C**

BRUCE SCHNEIER

Contents in Brief

Publisher: Katherine Schowalter
Editor: Phil Suteland
Assistant Editor: Allison Egan
Managing Editor: Robert Arnold
Text Design & Composition: North Market Street Graphics

APPLIED CRYPTOGRAPHY, SECOND EDITION

PROTOCOLS, ALGORITHMS, AND SOURCE CODE IN C

BRUCE SCHNEIER

Disclaimers used by companies to distinguish their products are often claimed as trademarks. In all cases, the publisher and author disclaim any liability for any consequences arising from the use of the information in this book.

This text is published in the United States of America and is not to be distributed outside the United States of America.

Copyright © 1996 by John Wiley & Sons, Inc.
Published by John Wiley & Sons, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or by any information storage and retrieval system, without permission in writing from John Wiley & Sons, Inc.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought.

In no event will the publisher or author be liable for any damages, including consequential, incidental, or indirect damages, arising from the use or inability to use the protocols and algorithms in this book, even if the publisher or author has been advised of the possibility of such damages.

Some of the protocols and algorithms in this book are patented by others. It is the responsibility of the reader to obtain all necessary patent and copyright licenses before implementing any protocol or algorithm in this book. This book does not contain an exhaustive list of all applicable patents and copyrights.

Part III Cryptographic Algorithms

Some of the protocols and algorithms in this book are required under the United States Export Administration Regulations. It is the responsibility of the reader to obtain all necessary export licenses before implementing any protocol or algorithm in this book.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the 1976 United States Copyright Act without the permission of the copyright owner is prohibited. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

17 Other Stream Ciphers and Real Random-Sequence Generators

18 One-Way Hash Functions

19 Public-Key Algorithms

20 Public-Key Digital Signature Algorithms

21 Identification Schemes

22 Key-Exchange Algorithms

23 Special Algorithms for Protocols



John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Part IV The Real World

24 Example Implementations

25 Policies

Part V Source Code

References

Publisher: Katherine Schowalter
Editor: Phil Sutherland
Assistant Editor: Allison Roarty
Managing Editor: Robert Aronds
Text Design & Composition: North Market Street Graphics

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc. is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This text is printed on acid-free paper.

Copyright © 1996 by Bruce Schneier
Published by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought.

In no event will the publisher or author be liable for any consequential, incidental, or indirect damages (including damages for loss of business profits, business interruption, loss of business information, and the like) arising from the use or inability to use the protocols and algorithms in this book, even if the publisher or author has been advised of the possibility of such damages.

Some of the protocols and algorithms in this book are protected by patents and copyrights. It is the responsibility of the reader to obtain all necessary patent and copyright licenses before implementing in software any protocol or algorithm in this book. This book does not contain an exhaustive list of all applicable patents and copyrights.

Some of the protocols and algorithms in this book are regulated under the United States Department of State International Traffic in Arms Regulations. It is the responsibility of the reader to obtain all necessary export licenses before implementing in software for export any protocol or algorithm in this book.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging-in-Publication Data:

Schneier, Bruce

Applied Cryptography Second Edition : protocols, algorithms, and source code in C
/ Bruce Schneier.

p. cm.

Includes bibliographical references (p. 675).

ISBN 0-471-12845-7 (cloth : acid-free paper). — ISBN

0-471-11709-9 (paper : acid-free paper)

1. Computer security. 2. Telecommunication—Security measures.

3. Cryptography. I. Title.

QA76.9.A25S35 1996

005.8'2—dc20

95-12398

CIP

Printed in the United States of America
10 9 8 7 6

Plaintext is denoted by M , for message, or P , for plaintext. It can be a stream of bits, a text file, a bitmap, a stream of digitized voice, a digital video image . . . whatever. As far as a computer is concerned, M is simply binary data. (After this chapter, this book concerns itself with binary data and computer cryptography.) The plaintext can be intended for either transmission or storage. In any case, M is the message to be encrypted.

Ciphertext is denoted by C . It is also binary data: sometimes the same size as M , sometimes larger. (By combining encryption with compression, C may be smaller than M . However, encryption does not accomplish this.) The encryption function E , operates on M to produce C . Or, in mathematical notation:

$$E(M) = C$$

In the reverse process, the decryption function D operates on C to produce M :

$$D(C) = M$$

Since the whole point of encrypting and then decrypting a message is to recover the original plaintext, the following identity must hold true:

$$D(E(M)) = M$$

Authentication, Integrity, and Nonrepudiation

In addition to providing confidentiality, cryptography is often asked to do other jobs:

- **Authentication.** It should be possible for the receiver of a message to ascertain its origin; an intruder should not be able to masquerade as someone else.
- **Integrity.** It should be possible for the receiver of a message to verify that it has not been modified in transit; an intruder should not be able to substitute a false message for a legitimate one.
- **Nonrepudiation.** A sender should not be able to falsely deny later that he sent a message.

These are vital requirements for social interaction on computers, and are analogous to face-to-face interactions. That someone is who he says he is . . . that someone's credentials—whether a driver's license, a medical degree, or a passport—are valid . . . that a document purporting to come from a person actually came from that person. . . . These are the things that authentication, integrity, and nonrepudiation provide.

Algorithms and Keys

A **cryptographic algorithm**, also called a **cipher**, is the mathematical function used for encryption and decryption. (Generally, there are two related functions: one for encryption and the other for decryption.)

ular encrypted key. The server doesn't have to be secure or trusted, since it will not be able to decrypt any of the messages.

3.2 AUTHENTICATION

When Alice logs into a host computer (or an automatic teller, or a telephone banking system, or any other type of terminal), how does the host know who she is? How does the host know she is not Eve trying to falsify Alice's identity? Traditionally, passwords solve this problem. Alice enters her password, and the host confirms that it is correct. Both Alice and the host know this secret piece of knowledge and the host requests it from Alice every time she tries to log in.

Authentication Using One-Way Functions

What Roger Needham and Mike Guy realized is that the host does not need to know the passwords; the host just has to be able to differentiate valid passwords from invalid passwords. This is easy with one-way functions [1599,526,1274,1121]. Instead of storing passwords, the host stores one-way functions of the passwords.

- (1) Alice sends the host her password.
- (2) The host performs a one-way function on the password.
- (3) The host compares the result of the one-way function to the value it previously stored.

Since the host no longer stores a table of everybody's valid password, the threat of someone breaking into the host and stealing the password list is mitigated. The list of passwords operated on by the one-way function is useless, because the one-way function cannot be reversed to recover the passwords.

Dictionary Attacks and Salt

A file of passwords encrypted with a one-way function is still vulnerable. In his spare time, Mallory compiles a list of the 1,000,000 most common passwords. He operates on all 1,000,000 of them with the one-way function and stores the results. If each password is about 8 bytes, the resulting file will be no more than 8 megabytes; it will fit on a few floppy disks. Now, Mallory steals an encrypted password file. He compares that file with his file of encrypted possible passwords and sees what matches.

This is a **dictionary attack**, and it's surprisingly successful (see Section 8.1). **Salt** is a way to make it more difficult. Salt is a random string that is concatenated with passwords before being operated on by the one-way function. Then, both the salt value and the result of the one-way function are stored in a database on the host. If the number of possible salt values is large enough, this practically eliminates a dictionary attack against commonly used passwords because Mallory has to generate the one-way hash for each possible salt value. This is a simple attempt at an initialization vector (see Section 9.3).

The point here is to make sure that Mallory has to do a trial encryption of each password in his dictionary every time he tries to break another person's password, rather than just doing one massive precomputation for all possible passwords.

A lot of salt is needed. Most UNIX systems use only 12 bits of salt. Even with that, Daniel Klein developed a password-guessing program that often cracks 40 percent of the passwords on a given host system within a week [847,848] (see Section 8.1). David Feldmeier and Philip Karn compiled a list of about 732,000 common passwords concatenated with each of 4096 possible salt values. They estimate that 30 percent of passwords on any given host can be broken with this list [561].

Salt isn't a panacea; increasing the number of salt bits won't solve everything. Salt only protects against general dictionary attacks on a password file, not against a concerted attack on a single password. It protects people who have the same password on multiple machines, but doesn't make poorly chosen passwords any better.

SKEY

SKEY is an authentication program that relies on a one-way function for its security. It's easy to explain.

To set up the system, Alice enters a random number, R . The computer computes $f(R)$, $f(f(R))$, $f(f(f(R)))$, and so on, about a hundred times. Call these numbers x_1 , x_2 , x_3 , ..., x_{100} . The computer prints out this list of numbers, and Alice puts it in her pocket for safekeeping. The computer also stores x_{101} , in the clear, in a login database next to Alice's name.

The first time Alice wants to log in, she types her name and x_{100} . The computer calculates $f(x_{100})$ and compares it with x_{101} ; if they match, Alice is authenticated. Then, the computer replaces x_{101} with x_{100} in the database. Alice crosses x_{100} off her list.

Every time Alice logs in, she enters the last uncrossed number on her list: x_i . The computer calculates $f(x_i)$ and compares it with x_{i+1} stored in its database. Eve can't get any useful information because each number is only used once, and the function is one-way. Similarly, the database is not useful to an attacker. Of course, when Alice runs out of numbers on her list, she has to reinitialize the system.

Authentication Using Public-Key Cryptography

Even with salt, the first protocol has serious security problems. When Alice sends her password to her host, anyone who has access to her data path can read it. She might be accessing her host through a convoluted transmission path that passes through four industrial competitors, three foreign countries, and two forward-thinking universities. Eve can be at any one of those points, listening to Alice's login sequence. If Eve has access to the processor memory of the host, she can see the password before the host hashes it.

Public-key cryptography can solve this problem. The host keeps a file of every user's public key; all users keep their own private keys. Here is a naïve attempt at a protocol. When logging in, the protocol proceeds as follows:

- (1) The host sends Alice a random string.
- (2) Alice encrypts the string with her private key and sends it back to the host, along with her name.
- (3) The host looks up Alice's public key in its database and decrypts the message using that public key.
- (4) If the decrypted string matches what the host sent Alice in the first place, the host allows Alice access to the system.

No one else has access to Alice's private key, so no one else can impersonate Alice. More important, Alice never sends her private key over the transmission line to the host. Eve, listening in on the interaction, cannot get any information that would enable her to deduce the private key and impersonate Alice.

The private key is both long and non-mnemonic, and will probably be processed automatically by the user's hardware or communications software. This requires an intelligent terminal that Alice trusts, but neither the host nor the communications path needs to be secure.

It is foolish to encrypt arbitrary strings—not only those sent by untrusted third parties, but under any circumstances at all. Attacks similar to the one discussed in Section 19.3 can be mounted. Secure proof-of-identity protocols take the following, more complicated, form:

- (1) Alice performs a computation based on some random numbers and her private key and sends the result to the host.
- (2) The host sends Alice a different random number.
- (3) Alice makes some computation based on the random numbers (both the ones she generated and the one she received from the host) and her private key, and sends the result to the host.
- (4) The host does some computation on the various numbers received from Alice and her public key to verify that she knows her private key.
- (5) If she does, her identity is verified.

If Alice does not trust the host any more than the host trusts Alice, then Alice will require the host to prove its identity in the same manner.

Step (1) might seem unnecessary and confusing, but it is required to prevent attacks against the protocol. Sections 21.1 and 21.2 mathematically describe several algorithms and protocols for proving identity. See also [935].

Mutual Authentication Using the Interlock Protocol

Alice and Bob are two users who want to authenticate each other. Each of them has a password that the other knows: Alice has P_A and Bob has P_B . Here's a protocol that will *not* work:

- (1) Alice and Bob trade public keys.
- (2) Alice encrypts P_A with Bob's public key and sends it to him.

- (3) Bob encrypts P_B with Alice's public key and sends it to her.
- (4) Alice decrypts what she received in step (2) and verifies that it is correct.
- (5) Bob decrypts what he received in step (3) and verifies that it is correct.

Mallory can launch a successful man-in-the-middle attack (see Section 3.1):

- (1) Alice and Bob trade public keys. Mallory intercepts both messages. He substitutes his public key for Bob's and sends it to Alice. Then he substitutes his public key for Alice's and sends it to Bob.
- (2) Alice encrypts P_A with "Bob's" public key and sends it to him. Mallory intercepts the message, decrypts P_A with his private key, re-encrypts it with Bob's public key and sends it on to him.
- (3) Bob encrypts P_B with "Alice's" public key and sends it to her. Mallory intercepts the message, decrypts P_B with his private key, re-encrypts it with Alice's public key, and sends it on to her.
- (4) Alice decrypts P_B and verifies that it is correct.
- (5) Bob decrypts P_A and verifies that it is correct.

Alice and Bob see nothing different. However, Mallory knows both P_A and P_B .

Donald Davies and Wyn Price describe how the interlock protocol (described in Section 3.1) can defeat this attack [435]. Steve Bellovin and Michael Merritt discuss ways to attack this protocol [110]. If Alice is a user and Bob is a host, Mallory can pretend to be Bob, complete the beginning steps of the protocol with Alice, and then drop the connection. True artistry demands Mallory do this by simulating line noise or network failure, but the final result is that Mallory has Alice's password. He can then connect with Bob and complete the protocol, thus getting Bob's password, too.

The protocol can be modified so that Bob gives his password before Alice, under the assumption that the user's password is much more sensitive than the host's password. This falls to a more complicated attack, also described in [110].

SKID

SKID2 and SKID3 are symmetric cryptography identification protocols developed for RACE's RIPE project [1305] (See Section 25.7). They use a MAC (see Section 2.4) to provide security and both assume that both Alice and Bob share a secret key, K .

SKID2 allows Bob to prove his identity to Alice. Here's the protocol:

- (1) Alice chooses a random number, R_A . (The RIPE document specifies a 64-bit number). She sends it to Bob.
- (2) Bob chooses a random number, R_B . (The RIPE document specifies a 64-bit number). He sends Alice:

$$R_B, H_K(R_A, R_B, B)$$

H_K is the MAC. (The RIPE document suggests the RIPE-MAC function—see Section 18.14.) B is Bob's name.

- (3) Alice computes $H_K(R_A, R_B, B)$ and compares it with what she received from Bob. If the results are identical, then Alice knows that she is communicating with Bob.

SKID3 provides mutual authentication between Alice and Bob. Steps (1) through (3) are identical to SKID2, and then the protocol proceeds with:

- (4) Alice sends Bob:

$$H_K(R_B, A)$$

A is Alice's name.

- (5) Bob computes $H_K(R_B, A)$, and compares it with what he received from Alice. If the results are identical, then Bob knows that he is communicating with Alice.

This protocol is not secure against a man-in-the-middle attack. In general, a man-in-the-middle attack can defeat any protocol that doesn't involve a secret of some kind.

Message Authentication

When Bob receives a message from Alice, how does he know it is authentic? If Alice signed her message, this is easy. Alice's digital signature is enough to convince anyone that the message is authentic.

Symmetric cryptography provides some authentication. When Bob receives a message from Alice encrypted in their shared key, he knows it is from Alice. No one else knows their key. However, Bob has no way of convincing a third party of this fact. Bob can't show the message to Trent and convince him that it came from Alice. Trent can be convinced that the message came from either Alice or Bob (since no one else shared their secret key), but he has no way of knowing which one.

If the message is unencrypted, Alice could also use a MAC. This also convinces Bob that the message is authentic, but has the same problems as symmetric cryptography solutions.

3.3 AUTHENTICATION AND KEY EXCHANGE

These protocols combine authentication with key exchange to solve a general computer problem: Alice and Bob are on opposite ends of a network and want to talk securely. How can Alice and Bob exchange a secret key and at the same time each be sure that he or she is talking to the other and not to Mallory? Most of the protocols assume that Trent shares a different secret key with each participant, and that all of these keys are in place before the protocol begins.

The symbols used in these protocols are summarized in Table 3.1.

Wide-Mouth Frog

The Wide-Mouth Frog protocol [283,284] is probably the simplest symmetric key-management protocol that uses a trusted server. Both Alice and Bob share a secret

TABLE 3.1
Symbols used in authentication and key exchange protocols

A	Alice's name
B	Bob's name
E_A	Encryption with a key Trent shares with Alice
E_B	Encryption with a key Trent shares with Bob
I	Index number
K	A random session key
L	Lifetime
T_A, T_B	A timestamp
R_A, R_B	A random number, sometimes called a nonce , chosen by Alice and Bob respectively

key with Trent. The keys are just used for key distribution and not to encrypt any actual messages between users. Just by using two messages, Alice transfers a session key to Bob:

- (1) Alice concatenates a timestamp, Bob's name, and a random session key and encrypts the whole message with the key she shares with Trent. She sends this to Trent, along with her name.

$$A, E_A(T_A, B, K)$$

- (2) Trent decrypts the message from Alice. Then he concatenates a new timestamp, Alice's name, and the random session key; he encrypts the whole message with the key he shares with Bob. Trent sends to Bob:

$$E_B(T_B, A, K)$$

The biggest assumption made in this protocol is that Alice is competent enough to generate good session keys. Remember that random numbers aren't easy to generate; it might be more than Alice can be trusted to do properly.

Yahalom

In this protocol, both Alice and Bob share a secret key with Trent [283,284].

- (1) Alice concatenates her name and a random number, and sends it to Bob.

$$A, R_A$$

- (2) Bob concatenates Alice's name, Alice's random number, his own random number, and encrypts it with the key he shares with Trent. He sends this to Trent, along with his name.

$$B, E_B(A, R_A, R_B)$$

- (3) Trent generates two messages. The first consists of Bob's name, a random session key, Alice's random number, and Bob's random number, all encrypted with the key he shares with Alice. The second consists of

Alice's name and the random session key, encrypted with the key he shares with Bob. He sends both messages to Alice.

$$E_A(B, K, R_A, R_B), E_B(A, K)$$

- (4) Alice decrypts the first message, extracts K , and confirms that R_A has the same value as it did in step (1). Alice sends Bob two messages. The first is the message received from Trent, encrypted with Bob's key. The second is R_B , encrypted with the session key.

$$E_B(A, K), E_K(R_B)$$

- (5) Bob decrypts the message encrypted with his key, extracts K , and confirms that R_B has the same value as it did in step (2).

At the end, Alice and Bob are each convinced that they are talking to the other and not to a third party. The novelty here is that Bob is the first one to contact Trent, who only sends one message to Alice.

Needham-Schroeder

This protocol, invented by Roger Needham and Michael Schroeder [1159], also uses symmetric cryptography and Trent.

- (1) Alice sends a message to Trent consisting of her name, Bob's name, and a random number.

$$A, B, R_A$$

- (2) Trent generates a random session key. He encrypts a message consisting of a random session key and Alice's name with the secret key he shares with Bob. Then he encrypts Alice's random value, Bob's name, the key, and the encrypted message with the secret key he shares with Alice. Finally, he sends her the encrypted message:

$$E_A(R_A, B, K, E_B(K, A))$$

- (3) Alice decrypts the message and extracts K . She confirms that R_A is the same value that she sent Trent in step (1). Then she sends Bob the message that Trent encrypted in his key.

$$E_B(K, A)$$

- (4) Bob decrypts the message and extracts K . He then generates another random value, R_B . He encrypts the message with K and sends it to Alice.

$$E_K(R_B)$$

- (5) Alice decrypts the message with K . She generates $R_B - 1$ and encrypts it with K . Then she sends the message back to Bob.

$$E_K(R_B - 1)$$

- (6) Bob decrypts the message with K and verifies that it is $R_B - 1$.

All of this fussing around with R_A and R_B and $R_B - 1$ is to prevent **replay attacks**. In this attack, Mallory can record old messages and then use them later in an attempt to subvert the protocol. The presence of R_A in step (2) assures Alice that

Trent's message is legitimate and not a replay of a response from a previous execution of the protocol. When Alice successfully decrypts R_B and sends Bob $R_B - 1$ in step (5), Bob is ensured that Alice's messages are not replays from an earlier execution of the protocol.

The major security hole in this protocol is that old session keys are valuable. If Mallory gets access to an old K , he can launch a successful attack [461]. All he has to do is record Alice's messages to Bob in step (3). Then, once he has K , he can pretend to be Alice:

- (1) Mallory sends Bob the following message:

$$E_B(K, A)$$

- (2) Bob extracts K , generates R_B , and sends Alice:

$$E_K(R_B)$$

- (3) Mallory intercepts the message, decrypts it with K , and sends Bob:

$$E_K(R_B - 1)$$

- (4) Bob verifies that "Alice's" message is $R_B - 1$.

Now, Mallory has Bob convinced that he is Alice.

A stronger protocol, using timestamps, can defeat this attack [461, 456]. A timestamp is added to Trent's message in step (2) encrypted with Bob's key: $E_B(K, A, T)$. Timestamps require a secure and accurate system clock—not a trivial problem in itself.

If the key Trent shares with Alice is ever compromised, the consequences are drastic. Mallory can use it to obtain session keys to talk with Bob (or anyone else he wishes to talk to). Even worse, Mallory can continue to do this even after Alice changes her key [90].

Needham and Schroeder attempted to correct these problems in a modified version of their protocol [1160]. Their new protocol is essentially the same as the Otway-Rees protocol, published in the same issue of the same journal.

Otway-Rees

This protocol also uses symmetric cryptography [1224].

- (1) Alice generates a message consisting of an index number, her name, Bob's name, and a random number, all encrypted in the key she shares with Trent. She sends this message to Bob along with the index number, her name, and his name:

$$I, A, B, E_A(R_A, I, A, B)$$

- (2) Bob generates a message consisting of a new random number, the index number, Alice's name, and Bob's name, all encrypted in the key he shares with Trent. He sends it to Trent, along with Alice's encrypted message, the index number, her name, and his name:

$$I, A, B, E_A(R_A, I, A, B), E_B(R_B, I, A, B)$$

- (3) Trent generates a random session key. Then he creates two messages. One is Alice's random number and the session key, encrypted in the key he shares with Alice. The other is Bob's random number and the session key, encrypted in the key he shares with Bob. He sends these two messages, along with the index number, to Bob:

$$I, E_A(R_A, K), E_B(R_B, K)$$

- (4) Bob sends Alice the message encrypted in her key, along with the index number:

$$I, E_A(R_A, K)$$

- (5) Alice decrypts the message to recover her key and random number. She then confirms that both have not changed in the protocol.

Assuming that all the random numbers match, and the index number hasn't changed along the way, Alice and Bob are now convinced of each other's identity, and they have a secret key with which to communicate.

Kerberos

Kerberos is a variant of Needham-Schroeder and is discussed in detail in Section 24.5. In the basic Kerberos Version 5 protocol, Alice and Bob each share keys with Trent. Alice wants to generate a session key for a conversation with Bob.

- (1) Alice sends a message to Trent with her identity and Bob's identity.

$$A, B$$

- (2) Trent generates a message with a timestamp, a lifetime, L , a random session key, and Alice's identity. He encrypts this in the key he shares with Bob. Then he takes the timestamp, the lifetime, the session key, and Bob's identity, and encrypts these in the key he shares with Alice. He sends both encrypted messages to Alice.

$$E_A(T, L, K, B), E_B(T, L, K, A)$$

- (3) Alice generates a message with her identity and the timestamp, encrypts it in K , and sends it to Bob. Alice also sends Bob the message encrypted in Bob's key from Trent.

$$E_K(A, T), E_B(T, L, K, A)$$

- (4) Bob creates a message consisting of the timestamp plus one, encrypts it in K , and sends it to Alice.

$$E_K(T + 1)$$

This protocol works, but it assumes that everyone's clocks are synchronized with Trent's clock. In practice, the effect is obtained by synchronizing clocks to within a few minutes of a secure time server and detecting replays within the time interval.

Neuman-Stubblebine

Whether by system faults or by sabotage, clocks can become unsynchronized. If the clocks get out of sync, there is a possible attack against most of these protocols

[644]. If the sender's clock is ahead of the receiver's clock, Mallory can intercept a message from the sender and replay it later when the timestamp becomes current at the receiver's site. This attack is called **suppress-replay** and can have irritating consequences.

This protocol, first presented in [820] and corrected in [1162] attempts to counter the suppress-replay attack. It is an enhancement to Yahalom and is an excellent protocol.

- (1) Alice concatenates her name and a random number and sends it to Bob.

$$A, R_A$$

- (2) Bob concatenates Alice's name, her random number, and a timestamp, and encrypts with the key he shares with Trent. He sends it to Trent along with his name and a new random number.

$$B, R_B, E_B(A, R_A, T_B)$$

- (3) Trent generates a random session key. Then he creates two messages. The first is Bob's name, Alice's random number, a random session key, and the timestamp, all encrypted with the key he shares with Alice. The second is Alice's name, the session key, and the timestamp, all encrypted with the key he shares with Bob. He sends these both to Alice, along with Bob's random number.

$$E_A(B, R_A, K, T_B), E_A(A, K, T_B), R_B$$

- (4) Alice decrypts the message encrypted with her key, extracts K , and confirms that R_A has the same value as it did in step (1). Alice sends Bob two messages. The first is the message received from Trent, encrypted with Bob's key. The second is R_B , encrypted with the session key.

$$E_B(A, K, T_B), E_K(R_B)$$

- (5) Bob decrypts the message encrypted with his key, extracts K , and confirms that T_B and R_B have the same value they did in step (2).

Assuming both random numbers and the timestamp match, Alice and Bob are convinced of one another's identity and share a secret key. Synchronized clocks are not required because the timestamp is only relative to Bob's clock; Bob only checks the timestamp he generated himself.

One nice thing about this protocol is that Alice can use the message she received from Trent for subsequent authentication with Bob, within some predetermined time limit. Assume that Alice and Bob completed the above protocol, communicated, and then terminated the connection. Alice and Bob can reauthenticate in three steps, without having to rely on Trent.

- (1) Alice sends Bob the message Trent sent her in step (3) and a new random number.

$$E_B(A, K, T_B), R'_A$$

- (2) Bob sends Alice another new random number, and Alice's new random number encrypted in their session key.

$$R'_B, E_K(R'_A)$$

- (3) Alice sends Bob his new random number, encrypted in their session key.

$$E_K(R'_B)$$

The new random numbers prevent replay attacks.

DASS

The Distributed Authentication Security Service (DASS) protocols, developed at Digital Equipment Corporation, also provide for mutual authentication and key exchange [604,1519,1518]. Unlike the previous protocols, DASS uses both public-key and symmetric cryptography. Alice and Bob each have a private key. Trent has signed copies of their public keys.

- (1) Alice sends a message to Trent, consisting of Bob's name.

$$B$$

- (2) Trent sends Alice Bob's public key, K_B , signed with Trent's private key, T . The signed message includes Bob's name.

$$S_T(B, K_B)$$

- (3) Alice verifies Trent's signature to confirm that the key she received is actually Bob's public key. She generates a random session key, and a random public-key/private-key key pair: K_p . She encrypts a timestamp with K . Then she signs a key lifetime, L , her name, and K_p with her private key, K_A . Finally, she encrypts K with Bob's public key, and signs it with K_p . She sends all of this to Bob.

$$E_K(T_A), S_{K_A}(L, A, K_p), S_{K_p}(E_{K_B}(K))$$

- (4) Bob sends a message to Trent (this may be a different Trent), consisting of Alice's name.

$$A$$

- (5) Trent sends Bob Alice's public key, signed in Trent's private key. The signed message includes Alice's name.

$$S_T(A, K_A)$$

- (6) Bob verifies Trent's signature to confirm that the key he received is actually Alice's public key. He then verifies Alice's signature and recovers K_p . He verifies the signature and uses his private key to recover K . Then he decrypts T_A to make sure this is a current message.

- (7) If mutual authentication is required, Bob encrypts a new timestamp with K , and sends it to Alice.

$$E_K(T_B)$$

- (8) Alice decrypts T_B with K to make sure that the message is current.

SPX, a product by DEC, is based on DASS. Additional information can be found in [34].

Denning-Sacco

This protocol also uses public-key cryptography [461]: Trent keeps a database of everyone's public keys.

- (1) Alice sends a message to Trent with her identity and Bob's identity:
 A, B
- (2) Trent sends Alice Bob's public key, K_B , signed with Trent's private key, T . Trent also sends Alice her own public key, K_A , signed with his private key.
 $S_T(B, K_B), S_T(A, K_A)$
- (3) Alice sends Bob a random session key and a timestamp, signed in her private key and encrypted in Bob's public key, along with both signed public keys.
 $E_B(S_A(K, T_A)), S_T(B, K_B), S_T(A, K_A)$
- (4) Bob decrypts Alice's message with his private key and then verifies Alice's signature with her public key. He checks to make sure that the timestamp is still valid.

At this point both Alice and Bob have K , and can communicate securely.

This looks good, but it isn't. After completing the protocol with Alice, Bob can then masquerade as Alice [5]. Watch:

- (1) Bob sends his name and Carol's name to Trent
 B, C
- (2) Trent sends Bob both Bob's and Carol's signed public keys.
 $S_T(B, K_B), S_T(C, K_C)$
- (3) Bob sends Carol the signed session key and timestamp he previously received from Alice, encrypted with Carol's public key, along with Alice's certificate and Carol's certificate.
 $E_C(S_A(K, T_A)), S_T(A, K_A), S_T(C, K_C)$
- (4) Carol decrypts Alice's message with her private key and then verifies Alice's signature with her public key. She checks to make sure that the timestamp is still valid.

Carol now thinks she is talking to Alice; Bob has successfully fooled her. In fact, Bob can fool everyone on the network until the timestamp expires.

This is easy to fix. Add the names inside the encrypted message in step (3):

$$E_B(S_A(A, B, K, T_A)), S_T(A, K_A), S_T(B, K_B)$$

Now Bob can't replay the old message to Carol, because it is clearly meant for communication between Alice and Bob.

Woo-Lam

This protocol also uses public-key cryptography [1610, 1611]:

- (1) Alice sends a message to Trent with her identity and Bob's identity:

$$A, B$$

- (2) Trent sends Alice Bob's public key, K_B , signed with Trent's private key, T .

$$S_T(K_B)$$

- (3) Alice verifies Trent's signature. Then she sends Bob her name and a random number, encrypted with Bob's public key.

$$E_{K_B}(A, R_A)$$

- (4) Bob sends Trent his name, Alice's name, and Alice's random number encrypted with Trent's public key, K_T .

$$A, B, E_{K_T}(R_A)$$

- (5) Trent sends Bob Alice's public key, K_A , signed with Trent's private key. He also sends him Alice's random number, a random session key, Alice's name, and Bob's name, all signed with Trent's private key and encrypted with Bob's public key.

$$S_T(K_A), E_{K_B}(S_T(R_A, K, A, B))$$

- (6) Bob verifies Trent's signatures. Then he sends Alice the second part of Trent's message from step (5) and a new random number—all encrypted in Alice's public key.

$$E_{K_A}(S_T(R_A, K, A, B), R_B)$$

- (7) Alice verifies Trent's signature and her random number. Then she sends Bob the second random number, encrypted in the session key.

$$E_K(R_B)$$

- (8) Bob decrypts his random number and verifies that it unchanged.

Other Protocols

There are many other protocols in the literature. The X.509 protocols are discussed in Section 24.9, KryptoKnight is discussed in Section 24.6, and Encrypted Key Exchange is discussed in Section 22.5.

Another new public-key protocol is Kuperee [694]. And work is being done on protocols that use **beacons**, a trusted node on a network that continuously broadcasts authenticated nonces [783].

Lessons Learned

There are some important lessons in the previous protocols, both those which have been broken and those which have not:

- Many protocols failed because the designers tried to be too clever. They optimized their protocols by leaving out important pieces: names, random numbers, and so on. The remedy is to make everything explicit [43,44].
- Trying to optimize is an absolute tar pit and depends a whole lot on the assumptions you make. For example: If you have authenticated time, you can do a whole lot of things you can't do if you don't.

- The protocol of choice depends on the underlying communications architecture. Do you want to minimize the size of messages or the number of messages? Can all parties talk with each other or can only a few of them?

It's questions like these that led to the development of formal methods for analyzing protocols.

3.4 FORMAL ANALYSIS OF AUTHENTICATION AND KEY-EXCHANGE PROTOCOLS

The problem of establishing secure session keys between pairs of computers (and people) on a network is so fundamental that it has led to a great deal of research. Some of the research focused on the development of protocols like the ones discussed in Sections 3.1, 3.2, and 3.3. This, in turn, has led to a greater and more interesting problem: the formal analysis of authentication and key-exchange protocols. People have found flaws in seemingly secure protocols years after they were proposed, and researchers wanted tools that could prove a protocol's security from the start. Although much of this work can apply to general cryptographic protocols, the emphasis in research is almost exclusively on authentication and key exchange.

There are four basic approaches to the analysis of cryptographic protocols [1045]:

1. Model and verify the protocol using specification languages and verification tools not specifically designed for the analysis of cryptographic protocols.
2. Develop expert systems that a protocol designer can use to develop and investigate different scenarios.
3. Model the requirements of a protocol family using logics for the analysis of knowledge and belief.
4. Develop a formal method based on the algebraic term-rewriting properties of cryptographic systems.

A full discussion on these four approaches and the research surrounding them is well beyond the scope of this book. See [1047,1355] for a good introduction to the topic; I am only going to touch on the major contributions to the field.

The first approach treats a cryptographic protocol as any other computer program and attempts to prove correctness. Some researchers represent a protocol as a finite-state machine [1449,1565], others use extensions of first-order predicate calculus [822], and still others use specification languages to analyze protocols [1566]. However, proving correctness is not the same as proving security and this approach fails to detect many flawed protocols. Although it was widely studied at first, most of the work in this area has been redirected as the third approach gained popularity.

The second approach uses expert systems to determine if a protocol can reach an undesirable state (the leaking of a key, for example). While this approach better identifies flaws, it neither guarantees security nor provides techniques for develop-

ing attacks. It is good at determining whether a protocol contains a given flaw, but is unlikely to discover unknown flaws in a protocol. Examples of this approach can be found in [987,1521]; [1092] discusses a rule-based system developed by the U.S. military, called the Interrogator.

The third approach is by far the most popular, and was pioneered by Michael Burrows, Martin Abadi, and Roger Needham. They developed a formal logic model for the analysis of knowledge and belief, called **BAN logic** [283,284]. BAN logic is the most widely used logic for analyzing authentication protocols. It assumes that authentication is a function of integrity and freshness, and uses logical rules to trace both of those attributes through the protocol. Although many variants and extensions have been proposed, most protocol designers still refer back to the original work.

BAN logic doesn't provide a proof of security; it can only reason about authentication. It has a simple, straightforward logic that is easy to apply and still useful for detecting flaws. Some of the statements in BAN logic include:

Alice believes X. (Alice acts as though X is true.)

Alice sees X. (Someone has sent a message containing X to Alice, who can read and repeat X—possibly after decrypting it.)

Alice said X. (At some time, Alice sent a message that includes the statement X. It is not known how long ago the message was sent or even that it was sent during the current run of the protocol. It is known that Alice believed X when she said it.)

X is fresh. (X has not been sent in a message at any time before the current run of the protocol.)

And so on. BAN logic also provides rules for reasoning about belief in a protocol. These rules can then be applied to the logical statements about the protocol to prove things or answer questions about the protocol. For example, one rule is the message-meaning rule:

IF Alice believes that Alice and Bob share a secret key, K , and Alice sees X , encrypted under K , and Alice did not encrypt X under K , THEN Alice believes that Bob once said X .

Another rule is the nonce-verification rule:

IF Alice believes that X could have been uttered only recently and that Bob once said X , THEN Alice believes that Bob believes X .

There are four steps in BAN analysis:

- (1) Convert the protocol into idealized form, using the statements previously described.
- (2) Add all assumptions about the initial state of the protocol.

- (3) Attach logical formulas to the statements: assertions about the state of the system after each statement.
- (4) Apply the logical postulates to the assertions and assumptions to discover the beliefs held by the parties in the protocol.

The authors of BAN logic “view the idealized protocols as clearer and more complete specifications than traditional descriptions found in the literature. . . .” [283,284]. Others are not so impressed and criticize this step because it may not accurately reflect the real protocol [1161,1612]. Further debate is in [221,1557]. Other critics try to show that BAN logic can deduce characteristics about protocols that are obviously false [1161]—see [285,1509] for a rebuttal—and that BAN logic deals only with trust and not security [1509]. More debate is in [1488, 706,1002].

Despite these criticisms, BAN logic has been a success. It has found flaws in several protocols, including Needham-Schroeder and an early draft of a CCITT X.509 protocol [303]. It has uncovered redundancies in many protocols, including Yahalom, Needham-Schroeder, and Kerberos. Many published papers use BAN logic to make claims about their protocol’s security [40,1162,73].

Other logic systems have been published, some designed as extensions to BAN logic [645,586,1556,828] and others based on BAN to correct perceived weaknesses [1488,1002]. The most successful of these is GNY [645], although it has some shortcomings [40]. Probabilistic beliefs were added to BAN logic, with mixed success, by [292,474]. Other formal logics are [156,798,288]; [1514] attempts to combine the features of several logics. And [1124,1511] present logics where beliefs can change over time.

The fourth approach to the analysis of cryptographic protocols models the protocol as an algebraic system, expresses the state of the participants’ knowledge about the protocol, and then analyzes the attainability of certain states. This approach has not received as much attention as formal logics, but that is changing. It was first used by Michael Merritt [1076], who showed that an algebraic model can be used to analyze cryptographic protocols. Other approaches are in [473,1508,1530,1531,1532, 1510,1612].

The Navy Research Laboratory’s (NRL) Protocol Analyzer is probably the most successful application of these techniques [1512,823,1046,1513]; it has been used to discover both new and known flaws in a variety of protocols [1044,1045,1047]. The Protocol Analyzer defines the following actions:

- Accept (Bob, Alice, M , N). (Bob accepts the message M as from Alice during Bob’s local round N .)
- Learn (Eve, M). (Eve learns M .)
- Send (Alice, Bob, Q , M). (Alice sends M to Bob in response to query, Q .)
- Request (Bob, Alice, Q , N). (Bob sends Q to Alice during Bob’s local round N .)

From these actions, requirements can be specified. For example:

- If Bob accepted message M from Alice at some point in the past, then Eve did not learn M at some point in the past.
- If Bob accepted message M from Alice in Bob's local round N , then Alice sent M to Bob as a response to a query in Bob's local round N .

To use the NRL Protocol Analyzer, a protocol must be specified using the previous constructs. Then, there are four phases of analysis: defining transition rules for honest participants, describing operations available to all—honest and dishonest—participants, describing the basic building blocks of the protocol, and describing the reduction rules. The point of all this is to show that a given protocol meets its requirements. Tools like the NRL Protocol Analyzer could eventually lead to a protocol that can be proven secure.

While much of the work in formal methods involves applying the methods to existing protocols, there is some push towards using formal methods to design the protocols in the first place. Some preliminary steps in this direction are [711]. The NRL Protocol Analyzer also attempts to do this [1512,222,1513].

The application of formal methods to cryptographic protocols is still a fairly new idea and it's really hard to figure out where it is headed. At this point, the weakest link seems to be the formalization process.

3.5 MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY

Public-key cryptography uses two keys. A message encrypted with one key can be decrypted with the other. Usually one key is private and the other is public. However, let's assume that Alice has one key and Bob has the other. Now Alice can encrypt a message so that only Bob can decrypt it, and Bob can encrypt a message so that only Alice can read it.

This concept was generalized by Colin Boyd [217]. Imagine a variant of public-key cryptography with three keys: K_A , K_B , and K_C , distributed as shown in Table 3.2.

Alice can encrypt a message with K_A so that Ellen, with K_B and K_C , can decrypt it. So can Bob and Carol in collusion. Bob can encrypt a message so that Frank can read

TABLE 3.2
Three-Key Key Distribution

Alice	K_A
Bob	K_B
Carol	K_C
Dave	K_A and K_B
Ellen	K_B and K_C
Frank	K_C and K_A