ADVANCES IN INFORMATION SECURITY

Hop Integrity in the Internet

Chin-Tser Huang
Mohamed G. Gouda

Springer

HOP INTEGRITY IN THE INTERNET

Advances in Information Security

Sushil Jajodia
Consulting Editor
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030-4444
email: jajodia@gmu.edu

The goals of the Springer International Series on ADVANCES IN INFORMATION SECURITY are, one, to establish the state of the art of, and set the course for future research in information security and, two, to serve as a central reference source for advanced and timely topics in information security research and development. The scope of this series includes all aspects of computer and network security and related areas such as fault tolerance and software assurance.

ADVANCES IN INFORMATION SECURITY aims to publish thorough and cohesive overviews of specific topics in information security, as well as works that are larger in scope or that contain more detailed background information than can be accommodated in shorter survey articles. The series also serves as a forum for topics that may not have reached a level of maturity to warrant a comprehensive textbook treatment.

Researchers, as well as developers, are encouraged to contact Professor Sushil Jajodia with ideas for books under this series.

Additional titles in the series:

BIOMETRIC USER AUTHENTICATION FOR IT SECURITY: From Fundamentals to Handwriting by Claus Vielhauer; ISBN-10: 0-387-26194-X

IMPACTS AND RISK ASSESSMENT OF TECHNOLOGY FOR INTERNET SECURITY:Enabled Information Small-Medium Enterprises (TEISMES) by Charles A. Shoniregun; ISBN-10: 0-387-24343-7

SECURITY IN E-LEARNING by Edgar R. Weippl; ISBN: 0-387-24341-0

IMAGE AND VIDEO ENCRYPTION: From Digital Rights Management to Secured Personal Communication by Andreas Uhl and Andreas Pommer; ISBN: 0-387-23402-0

INTRUSION DETECTION AND CORRELATION: Challenges and Solutions by Christopher Kruegel, Fredrik Valeur and Giovanni Vigna; ISBN: 0-387-23398-9

THE AUSTIN PROTOCOL COMPILER by Tommy M. McGuire and Mohamed G. Gouda; ISBN: 0-387-23227-3

ECONOMICS OF INFORMATION SECURITY by L. Jean Camp and Stephen Lewis; ISBN: 1-4020-8089-1

PRIMALITY TESTING AND INTEGER FACTORIZATION IN PUBLIC KEY CRYPTOGRAPHY by Song Y. Yan; ISBN: 1-4020-7649-5

SYNCHRONIZING E-SECURITY by Godfried B. Williams; ISBN: 1-4020-7646-0

INTRUSION DETECTION IN DISTRIBUTED SYSTEMS: An Abstraction-Based Approach by Peng Ning, Sushil Jajodia and X. Sean Wang; ISBN: 1-4020-7624-X

SECURE ELECTRONIC VOTING edited by Dimitris A. Gritzalis; ISBN: 1-4020-7301-1

DISSEMINATING SECURITY UPDATES AT INTERNET SCALE by Jun Li, Peter Reiher, Gerald J. Popek; ISBN: 1-4020-7305-4

SECURE ELECTRONIC VOTING by Dimitris A. Gritzalis; ISBN: 1-4020-7301-1

Additional information about this series can be obtained from http://www.springeronline.com

HOP INTEGRITY IN THE INTERNET

by

Chin-Tser Huang
University of South Carolina, USA

Mohamed G. Gouda
The University of Texas at Austin, USA

Springer

Prof. Chin-Tser Huang
University of Carolina
Dept. of Computer Science & Eng.
Columbia SC 29208

Dr. Mohamed G. Gouda
University of Texas at Austin
Dept. of Computer Sciences
Austin TX 78712-1188

Library of Congress Control Number: 2005933713

HOP INTEGRITY IN THE INTERNET
by Chin-Tser Huang and Mohamed G. Gouda

ISBN-13: 978-0-387-24426-6
ISBN-10: 0-387-24426-3
e-ISBN-13: 978-0-387-29444-5
e-ISBN-10: 0-387-29444-9

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11053828, 11570073

springeronline.com

Dedication

To my wife Cindy and our son

Austin for their support.

C.-T. H.

To A. R. G. and M. E. G.

for many fond memories.

M. G. G.

Contents

Preface

The subject of this monograph is a proposal called "Hop Integrity" intended to strengthen the security of the Internet against denial-of-service attacks.

Hop integrity provides three important guarantees whenever a message m makes one "hop" from a computer p to an adjacent computer q in the Internet. (Note that each of the two computers p and q can be a host or a router.) First, computer p does not send m (to q) unless it is certain that until recently q was up and running. Second, if both p and q are routers, then upon receiving message m, router q can check m and correctly conclude that m is "fresh" and has been sent recently by router p. In this case, q accepts m and proceeds to process it further. Third, if both p and q are routers and there is an adversary that modifies m into m′ or replaces m with an earlier message m′, then upon receiving message m′, router q can check m′ and conclude correctly that m′ is "modified" or "replayed". In this case, q discards m′.

The three guarantees of hop integrity constitute a defense against denial-of-service attacks as follows. If a message m, that is part of a denial-of-service attack, is originated by an adversarial host in the Internet and if the message header includes a wrong address for the originating host of m (in order to hide the true source of the attack), then message m will be classified as modified or replayed and will be discarded by the first router that receives m in the Internet.

To provide (the three guarantees of) hop integrity, the address resolution protocol of the Ethernet needs to be secured. Also, the IP header of each message in the Internet needs to have a "message digest" and a "sequence number". (The digest of a message is used by the receiving router to detect

whether or not the message is modified, and the sequence number of a message is used by the receiving router to detect whether or not the message is replayed. If a router receives a message and detects that the message is neither modified nor replayed, the router concludes correctly that the message is fresh.)

Also to provide hop integrity, each pair of adjacent routers p and q in the Internet need to share two security keys K and L. Router p uses key K to compute the digest of each message that p sends to q and uses key L to validate the digest of each message that p receives from q. Similarly, router q uses key L to compute the digest of each message that q sends to p and uses key K to validate the digest of each message that q receives from p. To enhance the security of their shared keys, each pair of adjacent routers need to update their shared keys regularly and relatively frequently, and so they use a light-weight key update protocol to update their shared keys.

In this monograph, a suite of protocols for providing hop integrity in the Internet is discussed in great detail. In particular, each protocol in this suite is specified and verified using an abstract and formal notation, called the Secure Protocol Notation. This notation is a variation of the Abstract Protocol Notation in the textbook "Elements of Network Protocol Design", written by the second author, Mohamed G. Gouda.

This monograph is primarily directed towards designers, reviewers, verifiers, and implementers of secure network protocols. It is also directed towards graduate students who are interested in network security and secure protocols.

Finally, the authors wish to thank their friends and colleagues in the Department of Computer Science and Engineering at the University of South Carolina at Columbia and in the Department of Computer Sciences at the University of Texas at Austin. The encouragement of our colleagues made this monograph possible.

Acknowledgments

We would like to thank Mootaz Elnozahy, Vijay Garg, Simon S. Lam, Aloysius K. Mok, and Tommy M. McGuire for their suggestions which have improved this monograph.

We also would like to thank several individuals in the Department of Computer Sciences at the University of Texas at Austin for many discussions concerning the subject of this monograph. These individuals are Eunjin (EJ) Jung, Alex X. Liu, Young-Ri Choi, and Hung-Ming Chen.

We also would like to thank several individuals in the Department of Computer Science and Engineering at the University of South Carolina at Columbia for many discussions centered on "Hop Integrity" and its potential role in improving the Security of the Internet. These individuals are Duncan Buell, Manton Matthews, Csilla Farkas, and Srihari Nelakuditi.

Chin-Tser Huang
Columbia, South Carolina

Mohamed G. Gouda
Austin, Texas

Chapter 1

INTRODUCTION

A network consists of computers connected to subnetworks. Examples of subnetworks include local area networks, telephone lines, and satellite links. The computers in a network are classified into hosts and routers. It is assumed that each host in a network is connected to one subnetwork, and each router is connected to two or more subnetworks via distinct interfaces.

Two computers in a network are called adjacent if both computers are connected to the same subnetwork. Two adjacent computers in a network can exchange messages over the common subnetwork(s) to which they are both connected. Two computers that are not adjacent to each other in a network can exchange messages through the help of intermediate routers as follows. Assume a message m is to be transmitted from a computer s to a faraway computer d in the same network. First, message m is forwarded in one hop from computer s to a router r.1 adjacent to s. Second, message m is forwarded in one hop from router r.1 to router r.2 adjacent to r.1, and so on. Finally, message m is forwarded in one hop from a router r.n that is adjacent to computer d to computer d.

Today, most computer networks in the Internet suffer from the following security problem. In a typical network, an adversary, that has an access to the network, can insert new messages, modify current messages, or replay old messages in the network. In many cases, the inserted, modified, or replayed messages can go undetected for some time until they cause severe damage to the network. More importantly, the physical location in the network where the adversary inserts new messages, modifies current messages, or replays old messages may never be determined.

One type of such malicious attacks is called denial-of-service attack [5], which manages to exhaust the communicating resources of a network or the computing resources of a host in order to largely reduce or completely deny

normal services provided by a network or a host. Two well-known examples of denial-of-service attacks in networks that support the Internet Protocol (or IP, for short) and the Transmission Control Protocol (or TCP, for short) are as follows.

I. *Smurf Attack*:

In an IP network, any computer can send a "ping" message to any other computer which replies by sending back a "pong" message to the first computer as required by Internet Control Message Protocol (or ICMP, for short) [43]. The ultimate destination in the pong message is the same as the original source in the ping message. An adversary can utilize these messages to attack a computer d in such a network as follows. First, the adversary inserts into the network a ping message whose original source is computer d and whose ultimate destination is a multicast address for every computer in the network. Second, a copy of the inserted ping message is sent to every computer in the network. Third, every computer in the network replies to its ping message by sending a pong message to computer d. Thus, computer d is flooded by pong messages that it had not requested.

II. *SYN Attack*:

To establish a TCP connection between two computers c and d, one of the two computers c sends a "SYN" message to the other computer d. When d receives the SYN message, it reserves some of its resources for the expected connection and sends a "SYN-ACK" message to c. When c receives the SYN-ACK message, it replies by sending back an "ACK" message to d. If d receives the ACK message, the connection is fully established and the two computers can start exchanging their data messages over the established connection. On the other hand, if d does not receive the ACK message for a specified time period of T seconds after it has sent the SYN-ACK message, d discards the partially established connection and releases all the resources reserved for that connection. The net effect of this scenario is that computer d has lost some of its resources for T seconds. An adversary can take advantage of such a scenario to attack computer d as follows [5, 51]. First, the adversary inserts into the network successive waves of SYN messages whose original sources are different (so that these messages cannot be easily detected and filtered out from the network) and whose ultimate destination is d. Second, d receives the SYN messages, reserves its resources for the expected connections, replies by sending SYN-ACK messages, then waits for the corresponding ACK messages which will never arrive. Third, the net effect of each wave of inserted SYN messages is that computer d loses all its resources for T seconds.

In these (and other [24]) types of attacks, an adversary inserts into a network messages with wrong original sources. These messages are accepted and forwarded by unsuspecting routers toward the computer under attack. To counter these attacks, each router p in the network should route a received message m only after it checks that the original source in m is a computer adjacent to p or m is forwarded to p by an adjacent router q. Performing the first check is straightforward, whereas performing the second check requires special protocols between adjacent routers. Filling in this void is the goal of this monograph.

In this monograph, we present the concept of hop integrity between adjacent routers as discussed in [14, 15, 19], and present the three protocols in the hop integrity protocol suite that are aimed to counter the aforementioned attacks and strengthen the security of the Internet. The basic idea of hop integrity is straightforward: whenever a router p receives a message m from an adjacent router q, p should be able to determine whether m was indeed sent by q or it was modified or replayed by an adversary that operates between p and q.

Next, we discuss the requirements of hop integrity. A network is said to provide hop integrity iff the following three conditions hold for every pair of adjacent routers p and q in the network.

I. *Detection of Next-Hop Failure*:
Router p does not send any message m to router q over the subnetwork connecting p and q unless router q has been up and reachable shortly before m is sent.

II. *Detection of Message Modification*:
Whenever router q receives a message m over the subnetwork connecting routers p and q, q can determine correctly whether message m was modified by an adversary after it was sent by p and before it was received by q.

III. *Detection of Message Replay*:
Whenever router q receives a message m over the subnetwork connecting routers p and q, and determines that message m was not modified, then q can determine correctly whether message m is another copy of a message that is received earlier by q.

The first condition infers sending integrity, in which a sender does not send a message to the receiver of the message unless the sender is sure the receiver has been up and reachable shortly before. The second and third conditions infer receiving integrity, in which whenever a receiver receives a message from a sender, the receiver can verify whether m was indeed sent by the sender or it was modified or replayed by an adversary that operates

between the receiver and the sender. Note that the sender and the receiver referred to in our presentation of hop integrity are one hop away from each other, i.e. they are connected to the same subnetwork.

For a network to provide hop integrity, we propose that the hop integrity protocol suite needs to be added to the protocol stack in each router in the network. The hop integrity protocol suite consists of the following three protocols:

I. *Secure Address Resolution Protocol*:
Secure address resolution protocol can detect next-hop failure. This protocol can be used to counter denial-of-service attacks that involve ARP spoofing [45, 52].

II. *Weak Hop Integrity Protocol*:
Weak hop integrity protocol can detect message modification. This protocol can be used to overcome denial-of-service attacks that involve message modification and do not involve message replay.

III. *Strong Hop Integrity Protocol*:
Strong hop integrity protocol is an enhanced version of weak hop integrity protocol in that besides detecting message modification, this protocol can also detect message replay. This protocol can be used to overcome denial-of-service attacks that involve message modification or message replay.

As discussed in [7] and [46], the protocol stack of each router (or host) in a network consists of four protocol layers. They are (from bottom to top): the subnetwork layer, the network layer, the transport layer, and the application layer. The secure address resolution protocol needs to be added to the subnetwork layer of this protocol stack, whereas the weak hop integrity protocol and the strong hop integrity protocol need to be added to the network layer.

Note that these proposed protocols are based on the following two assumptions:

I. *Local Area Network Assumption*:
The proposed protocols are based on local area networks, in particular Ethernets.

II. *Secure Router Assumption*:
The routers in the network and the software used by them are assumed to be secure and so they cannot be compromised by any adversary.

An adversary who wants to attack the network can compromise any group of hosts in the network and can cause them to execute actions on behalf of the adversary. However, under the Secure Router Assumption, the protocols of our hop integrity protocol suite can detect and defeat the adversarial actions.

It is instructive to compare hop integrity with secure routing [6, 37, 47], traceback [3, 48, 49], and IPsec [26]. In secure routing, for example [6], [37], and [47], the routing update messages that routers exchange are authenticated. This authentication ensures that every routing update message, that is modified or replayed, is detected and discarded. By contrast, hop integrity ensures that all messages (whether data or routing update messages), that are modified or replayed, are detected and discarded.

The purpose of traceback is for the destination under attack to reconstruct the path traversed by the attacking messages, so as to identify the real origin(s) of the messages responsible for the attack. Two schemes have been proposed to achieve traceback: message marking scheme [3, 49] and hash-based scheme [48]. In message marking scheme, when a router r receives a message m, it sends the traceback information, namely the pair (r, m), to the ultimate destination of the message. The traceback information for a message m is either sent in the ID field of IP header of message m itself [49] or sent in a separate ICMP message [3]. Due to the overhead incurred by sending traceback information, both Bellovin and Savage employ probabilistic methods rather than applying their methods to every message. In hash-based scheme, when a router r receives a message m, r stores the traceback information (r, m) in a hash table for some (relatively short) time. In these two schemes, a denial-of-service attack has to proceed for some time before the ultimate destination that is under the attack can detect the attack sources, if at all, and block them. In other words, these are detection-and-resolution schemes. By contrast, hop integrity is a prevention scheme. An attacking message, usually with a false source address, will be detected and discarded in its first hop. Thus, denial-of-service attacks will be prevented before they start.

The hop integrity protocol suite introduced in this monograph and the IPsec protocol suite presented in [26], [27], [28], [38], and [40] are both intended to provide security at the network layer. Nevertheless, these two protocol suites provide different, and somewhat complementary, services. On one hand, the hop integrity protocols are to be executed at all routers in a network, and they provide a minimum level of security for all communications between adjacent routers in that network. On the other hand, the IPsec protocols are to be executed at selected pairs of computers in the network, and they provide sophisticated levels of security for the communications between these selected computer pairs. Clearly, one can

envision networks where the hop integrity protocol suite and the IPsec protocol suite are both supported. When operating hand in hand, the hop integrity protocol suite can provide router authentication, router-to-router message integrity, and determination of the adversary location when the network is under attack, whereas the IPsec protocol suite can support source authentication, end-to-end message integrity, and confidentiality.

The rest of this monograph is organized as follows. In Chapter 2, we introduce the Abstract Protocol Notation that we use to specify all protocols in this monograph. In Chapter 3, we introduce more features of the AP notation that can be used to specify secure network protocols. In Chapter 4, we define denial-of-service attacks and discuss the role and use of hop integrity in countering these attacks. In the next four chapters, we introduce the three components of hop integrity protocol suite in order. First, in Chapter 5 we present a secure address resolution protocol that can achieve detection of next-hop failure. Second, in Chapter 6 we present the weak hop integrity protocol that can achieve detection of message modification. Third, in Chapters 7 and 8 we present the strong hop integrity protocol that can achieve detection of message replay in addition to achieving detection of message modification. In Chapter 7, we present the strong hop integrity protocol using soft sequence numbers, and in Chapter 8, we present a variation of the strong hop integrity protocol using hard sequence numbers. In Chapter 9, we discuss implementation considerations of hop integrity. Finally in Chapter 10, we illustrate four other applications of hop integrity besides overcoming most denial-of-service attacks.

Chapter 2

ABSTRACT PROTOCOL NOTATION

It is useful to specify network protocols using a formal notation. First, by using a formal notation to specify a network protocol, one can formally verify the correctness of this protocol and check that the protocol performs the function that it is intended to perform. Second, formal specification and verification is particularly important for secure network protocols. To verify the security guarantees of a protocol, one cannot depend only on some testing of the protocol because the tester may omit cases where vulnerabilities or weaknesses occur in the protocol. This is why we decided to specify all the secure protocols in this manuscript using a formal notation.

In this chapter, we present a variation of the Abstract Protocol Notation that is introduced in [13]. We use this variation to specify all the protocols presented in this manuscript.

The remainder of this chapter is organized as follows. In Section 2.1, we introduce the concept of processes and channels. In Section 2.2, we introduce the components of a process, namely constants, variables, and actions. In Section 2.3, we introduce the state transition diagram of a protocol, which is our tool to verify the correctness of the protocol. In Section 2.4, we introduce three more features of the AP notation, namely process arrays, parameters, and parameterized actions, that are used in our presentation.

1. PROCESSES AND CHANNELS

A protocol is defined by a collection of processes, and the channels between these processes. Processes in a protocol need to communicate with other processes in the same protocol by sending messages to and receiving

messages from the other processes. A message has a name (or a type) and can have zero or more fields that carry values to be used by the message receiver.

A message is transported from a sending process p to a receiving process q via the channel from p to q. The channel from p to q is the place where a message stays after it is sent by p and before it is received by q or before it is lost. Between each pair of adjacent processes p and q, there are two unidirectional channels: one from p to q, and the other from q to p.

Every channel in a protocol is both unbounded and FIFO (first-in, first-out). The unboundedness property means that an unbounded number of messages can reside simultaneously in a channel. The FIFO property means that messages are received from a channel in the same order in which they were sent into the channel. Messages that reside simultaneously in a channel form a sequence <m.1; m.2; ...; m.k> in accordance with the order in which messages m.1, m.2, ..., m.k have been sent by the sending process. The head message in the sequence, m.1, is the earliest sent, and the tail message in the sequence, m.k, is the latest sent. When the receiving process is ready to receive a message, it removes the head message, namely m.1, from the sequence. In this case, and the next message, namely m.2, becomes the next head message in the sequence, and so on. Therefore messages are to be received in the same order in which they were sent.

2. CONSTANTS, VARIABLES, AND ACTIONS

A process in a protocol is defined by a set of constants, a set of variables, and a set of actions. The protocol performs its designated function by executing the actions in its processes. In the next section, we explain how the actions in a process are executed. In this section, we discuss the constants, variables, and actions of a process.

A *constant* of a process has a name and a value, and can be one of the following four types: boolean, integer, range, and array. The constants of a process can be read but not updated by the actions of this process. Thus, the value of each constant of a process is either fixed or is updated by another process outside the protocol.

A *variable* of a process has a name and a value, and can be one of the following four types: boolean, integer, range, and array. The variables of a process can be read and updated by the actions of this process.

An *action* of a process consists of a guard, an arrow "→", and a statement:

<guard> → <statement>

The <guard> of an action is of one of the following two types: a local guard or a receiving guard.

A *local guard* is a boolean expression that involves the constants and variables of the process in which the local guard occurs.

A *receiving guard* is of the form:

rcv <message> **from** <process name>

A <statement> of an action is of one of the following six types: skip, assignment, send, selection, iteration, and sequence. Next, we describe the six types of statement and how to execute each of them.

A *skip* statement is of the form:

skip

The skip statement is executed by doing nothing.

An *assignment* statement is of the form:

v := E

where v is a variable of the process in which the assignment statement occurs, and E is an expression of the same type as v. The assignment statement is executed by assigning the current value of E to variable v.

A *send* statement is of the form:

send <message> **to** <name of another process>

This statement is executed by sending a message of the specified type to the specified process.

A *selection* statement is of the form:

if <boolean expression> → <statement>
...
[] <boolean expression> → <statement>
fi

This statement is executed by first computing the current value of each <boolean expression>, then arbitrarily selecting one <boolean expression> whose value is true and executing its corresponding <statement>.

An *iteration* statement is of the form:

do<boolean expression> → <statement>
od

This statement is executed by repeatedly computing the value of the <boolean expression> and then executing the <statement> when the value of the <boolean expression> is true. Execution of this statement terminates when the <boolean expression> becomes false.

A *sequence* statement is of the form:

<statement>; <statement>

This statement is executed by first executing the first <statement> and then executing the second <statement>.

Next, we use an example to illustrate the use of constants, variables, and actions. The following protocol consists of two processes p and q. In this protocol, process p can send a request message to process q, and then wait for a reply message from q before p can send the next request message to q. Process p can be specified as follows.

```
process p
var ready   :  boolean   {init. ready=true}
    txt, t  :  integer
begin
    ready →
        txt := any;
        send rqst(txt) to q;
        ready := false

[]  rcv rply(t) from q →
        {use text t in received message}
        ready := true
end
```

Process p has three variables: variable ready is used to remember whether process p is waiting for a rply message from process q or not, variable txt is used for keeping the content of the latest rqst message process p sends to process q, and variable t is used for keeping the content of the latest rply message process p receives from process q. There are two actions in process p. In the first action, if the value of ready is true, then p chooses a new value for txt, sends a rqst(txt) message to process q, and sets the value of ready to false. In the second action, if p receives a message rply(t) from q, then p sets the value of ready to true.

When process q receives a request message from process p, q returns a reply message to p. Process q can be specified as follows.

```
process q
var t  :  integer
begin
    rcv rqst(t) from p →
        t := any;
```

send rply(t) **to** p
end

Process q has one variable t, which is used for keeping the content of the latest message that process q receives from or sends to process p. There is one action in process q: if q receives a rqst(t) message from p, then q chooses a new value for t, and returns a rply(t) message to p.

3. STATE TRANSITION DIAGRAM

A state of a protocol is defined by one value for each constant and one value for each variable in each process in the protocol and by one sequence of messages for each channel in the protocol.

An action in a process p in a protocol is enabled at a state S of the protocol iff one of the following two conditions holds at S: the guard of the action is a local guard, or the guard is a receiving guard of the form **rcv** m **from** q and the head message in the channel from process q to process p is m at state S.

If one or more actions in the same process or in different processes in a protocol are enabled at a state S, then exactly one of the enabled actions is executed, yielding a next state S′ of the protocol. Likewise, if one or more actions are enabled at state S′, then exactly one of the enabled actions is executed, yielding a next state S″, and so on. An execution of a protocol may terminate when the protocol reaches a "deadlock state", where no action is enabled. If a protocol never reaches a deadlock state, then an execution of this protocol can continue endlessly.

Executing the actions (of different processes) in a protocol proceeds according to the following three rules:

I. *Atomicity*:
The actions in a protocol are executed one at a time.

II. *Nondeterminism*:
An action is executed only when its guard is true.

III. *Fairness*:
An action whose guard is continuously true is eventually executed.

To construct a state transition diagram of a protocol, we have to derive all the possible states that can be reached by the protocol. The derivation of reachable states begins with an initial state in which every constant and every variable is assigned an initial value and every channel in the network

is empty. Then, all the actions that are enabled at this state are identified. Execution of each of these enabled actions at the current state leads the network to a different next state. This procedure is continued at each of the next states until a deadlock state is reached or a previous state is reached.

After we derive all the reachable states of a protocol, we can draw the corresponding state transition diagram. In a state transition diagram, each node represents one network state, and each arrow from a node S to another node S′ represents an action execution that leads the network from state S to state S′.

Next, we use the protocol defined in the last section as an example for illustrating the construction of a state transition diagram. Assume that this network of process p and process q starts at a state defined by the following protocol predicate S.0.

S.0 : ready $\wedge$ txt = x $\wedge$ t.p = y $\wedge$ t.q = z $\wedge$
ch.p.q = < > $\wedge$ ch.q.p = < >

The first conjunct in S.0 asserts that variable ready in process p has the value true. The next three conjuncts assert that txt in process p has the value x, t in process p has the value y, and t in process q has the value z. The last two conjuncts assert that the two channels between processes p and q are empty.

At state S.0, exactly one action, namely the first action in process p, is enabled. Executing this action at state S.0 leads the network to the following state S.1.

S.1 : ~ready $\wedge$ txt = x $\wedge$ t.p = y $\wedge$ t.q = z $\wedge$
ch.p.q = <rqst(txt)> $\wedge$ ch.q.p = < >

At state S.1, only the sole action in process q is enabled. Assume that process q chooses a random value z′ for its variable t when executing this action at state S.1. Thus the network is led to the following state S.2.

S.2 : ~ready $\wedge$ txt = x $\wedge$ t.p = y $\wedge$ t.q = z′ $\wedge$
ch.p.q = < > $\wedge$ ch.q.p = <rply(t.q)>

At state S.2, only the second action in process p is enabled. Executing this action at S.2 leads the network to the following state S.3.

S.3 : ready $\wedge$ txt = x $\wedge$ t.p = z′ $\wedge$ t.q = z′ $\wedge$
ch.p.q = < > $\wedge$ ch.q.p = < >

At state S.3, only the first action in process p is enabled. Assume that process p chooses a random value x′ for its variable txt when executing this action at state S.3. Thus the network is led to the following state S.4.

S.4 : ~ready $\wedge$ txt = x′ $\wedge$ t.p = z′ $\wedge$ t.q = z′ $\wedge$

$$ch.p.q = \langle rqst(txt) \rangle \wedge ch.q.p = \langle\rangle$$

It turns out that this protocol has an infinite number of reachable states, because in each round process p chooses a new value for its variable txt before sending a message rqst(txt) to process q, and process q chooses a new value for its variable t before sending a message rply(t) to process p. Therefore, it is impossible to draw the corresponding state transition diagram in full.

To solve this problem, the definition of a state transition diagram for a protocol can be generalized as follows. Instead of each node in the diagram representing only one state of the protocol, some nodes in the diagram can be aggregated under a broader protocol predicate into one node that represents a nonempty subset of the protocol states.

For example, the initial state S.0 of this protocol can be found in an aggregated state that is defined by the following protocol predicate T.0.

$$T.0: \quad ready \wedge ch.p.q = \langle\rangle \wedge ch.q.p = \langle\rangle$$

State S.1 can be found in an aggregated state that is defined by the following protocol predicate T.1.

$$T.1: \quad \sim ready \wedge ch.p.q = \langle rqst(txt) \rangle \wedge ch.q.p = \langle\rangle$$

State S.2 can be found in an aggregated state that is defined by the following protocol predicate T.2.

$$T.2: \quad \sim ready \wedge ch.p.q = \langle\rangle \wedge ch.q.p = \langle rply(t.q) \rangle$$

Similarly, state S.3 can be found in the aggregated state defined by T.0, state S.4 can be found in the aggregated state defined by T.1, and so on.

We derive the following three inductions regarding the three aggregated states T.0, T.1, and T.2. First, at a state defined by T.0, only the first action in process p is enabled, and executing this action at a state defined by T.0 leads the protocol to a state defined by T.1. Second, at a state defined by T.1, only the sole action in process q is enabled, and executing this action at a state defined by T.1 leads the protocol to a state defined by T.2. Third, at a state defined by T.2, only the second action in process p is enabled, and executing this action at a state defined by T.2 leads the protocol to a state defined by T.0. Therefore, the sequence of transitions from T.0 to T.1, from T.1 to T.2, and from T.2 to T.0 forms a cycle in which the network performs progress. In this case, a state transition diagram for the protocol can be drawn as shown in Figure 2.1.

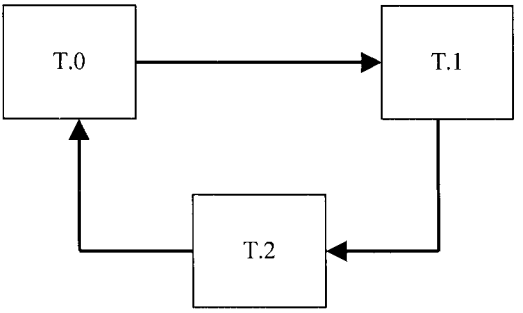


Figure 2-1. A state transition diagram for the example protocol.

4. PROCESS ARRAYS, PARAMETERS, AND PARAMETERIZED ACTIONS

In this section, we introduce two extensions of the AP notation. First, we introduce process arrays which allow one to define a set of identical processes by defining only one representative process. Second, we introduce parameters and parameterized actions which allow one to define a finite set of actions as a single parameterized action in a process.

A process array is a finite set of processes: each of them has the same set of constants, the same set of variables, and the same set of actions. Thus, all the processes in a process array can be specified by specifying only one representative process of the array. For example, let p be an array of n processes named p[0], p[1], ..., p[n-1] respectively. A representative process of this array can be p[i], where i is an index whose value is in the range between 0 and n-1.

A parameter has a name and is of type range. This implies that each parameter has a finite number of values.

A parameterized action is an action that refers to one or more parameters. A parameterized action is a shorthand notation for a finite set of actions: each of them can be obtained from the parameterized action by first selecting for each parameter i in the parameterized action a value v.i from the domain of i, and then replacing every occurrence of i in the parameterized action by the selected value v.i.

Next, we extend the example shown in Section 2.2 to illustrate the use of process arrays, parameters, and parameterized actions. In the extended example, process p communicates with an array of n processes q[i: 0 .. n-1].

In this protocol, process p can send a request message to any process q[i], and then wait for a reply message from q[i] before p can send the next request message to the same q[i]. While process p is waiting for a reply message from q[i], p can send a request message to any other process q[i'] that is different from q[i], provided that p is not waiting for a reply message from q[i']. Process p in the extended protocol can be specified as follows.

```
process p
const  n      :  integer       {number of processes in process array q}
var    ready  :  array [0 .. n-1] of boolean    {init. ready=true}
       txt, t :  integer
par    i      :  0 .. n-1
begin
     ready[i] →
          txt := any;
          send rqst(txt) to q[i];
          ready[i] := false

[]   rcv rply(t) from q[i] →
          {use text t in received message}
          ready[i] := true
end
```

Process p in the extended protocol has one constant n, which specifies the number of processes in process array q, and one parameter i, which stands for the index of process array q. Variable ready in process p is changed to an array of n booleans to remember whether process p is waiting for a rply message from each of the n processes in process array q or not. Both actions in process p are parameterized actions; each action is a shorthand notation of n actions as there are n possible values for parameter i. In the first parameterized action, if the value of ready[i] is true, then p chooses a new value for txt, sends a rqst(txt) message to process q[i], and sets the value of ready[i] to false. In the second parameterized action, if p receives a message rply(t) from q[i], then p sets the value of ready[i] to true.

Next, we specify process array q in the extended protocol as follows. Each new process q[i], when receiving a request message from p, will return a reply message to p. As discussed previously in this section, we can specify process array q by specifying one representative process q[i] in the array.

```
process q[i : 0 .. n-1]
const  n  :  integer    {number of processes in process array q}
var    t  :  integer
```

```
begin
    rcv rqst(t) from p →
        t := any;
        send rply(t) to p
end
```

Each process q[i] has one new constant n, which is the same as the constant n in the new process p. There is one action in process q[i]: if q[i] receives a rqst(t) message from p, then q[i] chooses a new value for t, and returns a rply(t) message to p.

Chapter 3

ABSTRACT SECURE PROTOCOLS

We have presented in Chapter 2 the basic features of the Abstract Protocol notation, and the state transition diagram of a protocol. In this chapter, we proceed to introduce more features of this AP notation. These features can be used to specify secure network protocols that can encounter adversarial attacks.

This chapter is organized as follows. In Section 3.1, we specify four types of adversarial actions, namely message snooping, message modification, message replay, and message loss, that an adversary can execute to disrupt the communications between any two processes in a protocol. In the next four sections, we introduce four security features of the AP notation and discuss how these features can be used to counter the four types of adversarial actions. In Section 3.2, we introduce the concept of security key, and discuss how to use security keys to counter message snooping actions. In Section 3.3, we introduce the concept of a message digest and discuss how to use message digests to counter message modifications. In Section 3.4, we introduce the concept of a nonce and discuss how to use nonces to counter message replays. In Section 3.5, we introduce timeout actions and discuss how to use these actions to counter message losses. Finally in Section 3.6, we use an example protocol to illustrate the use of security keys, message digests, nonces, and timeout actions.

1. ASSUMPTIONS ABOUT THE ADVERSARY

We assume that an adversary exists between any two processes p and q in a protocol. We also assume that this adversary can disrupt the communication between p and q by executing the following four types of actions, a finite number of times each.

I. *Message Snooping*:
The adversary can snoop on the messages exchanged between processes p and q by making a copy of each head message in each channel between p and q.

II. *Message Modification*:
The adversary can arbitrarily modify the content of the head message in one of the two channels between p and q according to the following restriction. Suppose a message has n fields f.0, f.1, f.2, ..., f.(n-1) that satisfy a relationship f.0 = F(f.1, f.2, ..., f.(n-1)) where F is a "security function". After the message is arbitrarily modified by the adversary, the n fields f.0′, f.1′, f.2′, ..., f.(n-1)′ of the modified message no longer satisfy the previous relationship, i.e. f.0′ ≠ F(f.1′, f.2′, ..., f.(n-1)′).

III. *Message Replay*:
The adversary can replace the head message in one of the two channels between p and q by a copy of a message that is of the same type and was sent earlier.

IV. *Message Loss*:
The adversary can discard a message by removing the head message from one of the two channels between p and q.

For simplicity, we assume that each head message in one of the two channels between p and q is affected by at most one adversarial action.

2. SECURITY KEYS

We assume that each key is a non-negative integer. We also assume that each data item is a non-negative integer and vice versa. Therefore, each key is also a data item.

We assume the existence of an appropriate encryption function NCR and an appropriate decryption function DCR. Each of these two functions takes a key and a data item as arguments and produces a data item as a result. Let K be a key and d be a data item. Then, the data item NCR(K, d) is called the

encryption of data item d using key K and the data item DCR(K, d) is called the decryption of data item d using key K.

A pair of keys (K, L) is called secure iff the following three conditions hold:

I. *Restoration*:
For every data item d,
$d = DCR(L, NCR(K, d))$, and
$d = DCR(K, NCR(L, d))$.

II. *Hiding*:
For every key K′ other than K and every key L′ other than L, there is a data item d such that
$d \neq DCR(L', NCR(K, d))$, and
$d \neq DCR(K', NCR(L, d))$.

III. *Secrecy*:
If K = L, then there is no efficient algorithm to deduce L from the set of data items $\{NCR(K, d) \mid d \text{ is a data item}\}$. If $K \neq L$, then there is no efficient algorithm to deduce L from K.

A secure key pair (K, L) is called *asymmetric* iff $K \neq L$, and is called *symmetric* iff K = L.

For an asymmetric secure key pair (K, L) that belongs to a process p in a network, one key K is called a *public key* of process p and is denoted as B_p, while the other key L is called a *private key* of process p and is denoted as R_p. The public key B_p is known to every process in the network and the adversary, while the private key R_p is only known to process p.

For a symmetric secure key pair (K, K) that belongs to a process pair p and q in a network, key K is called a *shared key* of processes p and q and is denoted as $S_{\{p, q\}}$. The shared key $S_{\{p, q\}}$ is known to processes p and q only.

An asymmetric secure key pair (B_p, R_p) can be used to counter snooping on the messages from a process q to process p as follows. For q to send p a message msg(txt), with one field txt that should be kept confidential, q computes

$$t := NCR(B_p, txt)$$

and sends a message msg(t) instead. When p receives msg(t), p recovers txt from t by computing

$$txt := DCR(R_p, t)$$

A symmetric secure key $S_{\{p, q\}}$ can be used to counter snooping on the messages exchanged between processes p and q as follows. For q to send p a

message msg(txt), with one field txt that should be kept confidential, q computes

$$t := NCR(S_{\{p, q\}}, txt)$$

and sends p a message msg(t) instead. When p receives msg(t), p recovers txt from t by computing

$$txt := DCR(S_{\{p, q\}}, t)$$

3. MESSAGE DIGESTS

A message digest function MD is a function that computes for any data item d a fixed-length data item MD(d) such that the following condition is satisfied.

Finger Printing: There is no efficient algorithm that computes, for any data item d, another data item d′ such that MD(d) = MD(d′).

Common message digest functions include MD5 [44], SHA [39], or HMAC [29].

Message digests can be used to counter message modification actions as follows. Assume that a message msg(txt) is to be sent from a process p to another process q. Assume also that p and q share a secret S. Before p sends q the message, p computes an integrity check d for this message as follows:

$$d := MD(txt; S)$$

where MD is a message digest function, and "txt; S" is a concatenation of the txt field and the shared secret. Then p adds d to the message and sends q a message msg(txt, d) instead. If the message is arbitrarily modified in its transit to become msg(txt′, d′), then q can detect the modification by computing MD(txt′; S) and checking that d′ is not equal to MD(txt′; S).

4. NONCES

A nonce is a non-negative integer, and so each nonce is also a data item. During the execution of a security protocol, each process in the protocol can generate a nonce by executing a function NONCE.

The sequence of nonces generated by a process satisfies the following two conditions:

I. *Non-repetition*: The value of a generated nonce is different from the values of all previously generated nonces in the sequence.

II. *Unpredictability*: The value of a generated nonce cannot be deduced from the values of all previously generated nonces in the sequence.

Nonces can be used to counter message replay actions as follows. Assume that a message msg(txt) that requires a reply is to be sent from a process p to another process q. Before p sends this message to q, p adds a nonce nc to the message and sends q a message msg(txt, nc) instead. When q receives the message and prepares a reply, q adds the same nonce nc to the reply. Finally, when p receives the reply and checks that the nonce is the same as that in the original message, it concludes correctly that neither the original message nor the reply was replaced by earlier messages.

5. TIMEOUT ACTIONS

A timeout action is an action that begins with a timeout guard. A timeout guard is of the form:

timeout <time expression>

The <time expression> is a boolean expression that involves the constants and variables of the process in which the timeout guard occurs. It can also refer to a time period that has passed since some action of the process has executed. This implies that each process has a real-time clock. The clocks in different processes do not need to be synchronized, but they have the same rate.

Timeout actions can be used to counter message loss actions as follows. If a process p sends a message to another process q and does not receive from q a reply for this message for a relatively long time, then p executes a timeout action to send q another copy of the same message or another message.

6. AN EXAMPLE PROTOCOL WITH SECURITY FEATURES

In this section, we extend the example shown in Section 2.2 to illustrate the use of security keys, message digests, nonces, and timeout actions. Recall that in the example protocol in Section 2.2, process p can send a request message to process q, and then wait for a reply message from q

before p can send the next request message to q. However, that protocol does not provide any security features introduced in this chapter. Therefore, it is vulnerable to the message snooping actions, message modification actions, message replay actions, and message loss actions executed by an adversary.

We extend this protocol to counter the four types of actions by an adversary as follows. In the extended protocol, we assume that each of processes p and q has an asymmetric secure key pair, and that p and q share a secret. The following four changes are made on the protocol. First, to counter message snooping actions, the text field of each message is encrypted using the public key of the receiving process. Second, to counter message replay action, a nonce is attached to each message. Third, to counter a message modification action, a message digest computed using the concatenation of the nonce, the message text, and the shared secret is attached to each message. Fourth, to counter message loss actions, p executes a timeout action to resend the same request message to q if p detects that a request message is lost in transit. Process p in the extended protocol can be specified as follows.

```
process p
const  R_p      :  integer     {private key of p}
       B_q      :  integer     {public key of q}
       S        :  integer     {shared secret between p and q}
var    ready    :  boolean     {init. ready=true}
       nc, c    :  integer     {nonce}
       txt, t   :  integer     {text}
       d        :  integer     {message digest}
begin
    ready →
        txt := any;
        send rqst(nc, NCR(B_q, txt), MD(nc; txt; S)) to q;
        ready := false

[]  rcv rply(c, t, d) from q →
        t := DCR(R_p, t);
        if ~ ready ∧ nc = c ∧ MD(nc; t; S) = d →
            {use decrypted text t in received message}
            ready := true
        [] ready ∨ nc ≠ c ∨ MD(nc; t; S) ≠ d →
            {discard received message}
            skip
        fi
```

```
[]  timeout (~ready ∧ #ch.p.q + #ch.q.p = 0) →
        send rqst(nc, NCR(B_q, txt), MD(nc; txt; S)) to q
end
```

Process p in the extended protocol has one constant R_p, B_q, and S, and three new variables nc, c, and d. Constant R_p is the private key of p, and constant B_q is the public key of q. Constant S specifies the shared secret between process p and process q. Variable nc specifies a nonce chosen for a new request message, variable c is used for keeping the value of the nonce in the last received reply message, and variable d is used for keeping the value of the message digest in the last received reply message. There are three actions in process p. In the first action, if the value of ready is true, then p randomly chooses a value for txt, sends a rqst(nc, txt, MD(nc; txt; S)) message to process q, and sets the value of ready to false. In the second action, if p receives a message rply(c, t, d) from q, then p verifies that c is equal to the last used nonce nc, and d is equal to the message digest MD(nc; t; S). If so, p sets the value of ready to true; otherwise p discards the received message and skips. The third action is a timeout action. In this action, if the value of ready is false and the number of messages that are currently in both the channel from p to q and the channel from q to p is 0 (which is an indication that the sent request message is lost in transit), then p resend the request message rqst(nc, NCR(B_q, txt), MD(nc; txt; S)) to q.

Next, we specify process q in the extended protocol as follows. The new process q, when receiving a request message from p, will first decrypt the text and verify that the received message was not modified in transit, and then return a reply message to p. Process q in the extended protocol can be specified as follows.

```
process q
const  R_q :  integer    {private key of q}
       B_p :  integer    {public key of p}
       S   :  integer    {shared secret between p and q}
var    c   :  integer    {nonce}
       t   :  integer    {text}
       d   :  integer    {message digest}
begin
    rcv rqst(c, t, d) from p →
        t := DCR(R_q, t);
        if MD(c; t; S) = d →
            t := any;
            d := MD(c; t; S);
            t := NCR(B_p, t);
```

```
            send rply(c, t, d) to p
        [] MD(c; t; S) ≠ d →
            {discard received message}
            skip
        fi
end
```

Process q in the extended protocol has three constants R_q, B_p, and S, and two new variables c and d. Constant R_q is the private key of q that corresponds to the public key B_q known to p, and constant B_p is the public key of p that corresponds to the private key R_p owned by p. Constant S is the same as the constant S in p. Variable c is used for keeping the value of the nonce in the last received request message, and variable d is used for keeping the value of the message digest in the last received request message. There is one action in process q: if q receives a rqst(c, t, d) message from p, then q decrypts t using its private key R_q, and verifies that the message was not modified in transit. If so, q chooses an arbitrary value for t, computes an integrity check d, encrypts t using p's public key B_p, and returns a rply(c, t, d) message to p; otherwise q discards the received message and skips.

Chapter 4

DENIAL-OF-SERVICE ATTACKS

A series of denial-of-service attacks that occurred in the past few years have caused severe problems to many Internet Service Providers (ISP) and online services, and have also posed new challenges to network security experts. According to a survey conducted by Computer Security Institute (CSI) and Federal Bureau of Investigation (FBI), the estimated financial losses caused by denial-of-service attacks amounted to more than $65M in the year of 2003 and more than $26M in the year of 2004 [54].

Most of the success of denial-of-service attacks can be attributed to the two-sided nature of these attacks: they are quite easy to launch but extremely hard to defend against. Denial-of-service attacks are easy to launch because generating messages of these attacks takes as few as just one computer and some handy tools that can be downloaded from the Internet. They are hard to defend against because the messages generated by denial-of-service attacks are almost indistinguishable from those normal messages generated by legitimate users.

The aim of denial-of-service attacks is to largely reduce or completely deny normal services provided by a network or a host. According to the ways these attacks achieve their goal, denial-of-service attacks can be divided into two categories [45]. The first category is called communication-stopping attacks: attacks in this category stop the communication of the target host with the outside world, for example ARP spoofing attack. The second category is called resource-exhausting attacks: attacks in this category exhaust the communicating resources of the target network or the computing resources of the target host, for example Smurf attack, SYN attack, and distributed denial-of-service attack. In this chapter, we discuss how the attacks in each of the two categories prevail, and why hop integrity is needed to counter these attacks.

1. COMMUNICATION-STOPPING ATTACKS

In this type of attacks, an adversary manages to stop the communications between the target host and the outside world, such that the target host cannot get normal services provided by the outside world, and the outside world cannot get normal services provided by the target host. ARP spoofing attack [45, 52] is an attack that is often used to achieve this goal.

We first give an introduction to ARP before we discuss the mechanism and defenses of ARP spoofing attack. The Address Resolution Protocol [42], or ARP for short, is a protocol for mapping an IP address to a hardware address that is recognized in the local network, in particular an Ethernet. To illustrate the operation of ARP, consider the following scenario in which a network consists of n computers h[0], h[1], ... , h[n-1]. These n computers are connected to the same Ethernet. Before any computer h[i] can send a message m to any other computer h[j] in this network, h[i] needs to obtain the hardware address of h[j]. This can be accomplished using ARP as follows. First, the ARP process in h[i] broadcasts a rqst(ipa) message over the Ethernet to every other computer in the network, where ipa is the IP address of the destination computer h[j]. Second, when the ARP process in any computer other than h[j] receives the rqst(ipa) message, it detects that ipa is not its own IP address and discards the message. Third, when the ARP process in computer h[j] receives the rqst(ipa) message, it detects that ipa is its own IP address, and sends a rply(ipa, hda) message over the Ethernet to computer h[i], where hda is the required hardware address of computer h[j]. When computer h[i] receives the rply(ipa, hda) message, it attaches hda to message m, sends m(hda) over the Ethernet to computer h[j], and keeps this mapping of ipa and hda (of computer h[j]) in an ARP cache for some time. Next time, if computer h[i] wants to send another message m′ to computer h[j], h[i] first checks its ARP cache to see whether the entry of h[j]'s ipa and hda has expired. If the entry has not expired yet, h[i] sends m′(hda) over the Ethernet to h[j]. Otherwise, h[i] repeats the process described above to obtain the hardware address of h[j].

This scenario demonstrates that there are three functions for ARP:

I. *Resolving IP Addresses*:

Using ARP, each computer can obtain the hardware address of any other computer (using the IP address of that other computer) on the same Ethernet.

II. *Supporting Dynamic Assignment of Addresses*:

ARP can be used to resolve the IP addresses of computers on the same Ethernet even if the IP addresses assigned to these computers

change over time. For example, consider the case where a mobile computer visits an Ethernet. In this case, the mobile computer can be assigned a temporary IP address through some configuration protocol like DHCP [9]. Then, the other computers on the Ethernet can use ARP to resolve this temporary IP address to the hardware address of the mobile computer, and so can send messages to that computer.

III. *Detecting Destination Failures*:

Consider the case where a computer h[i] needs to resolve the IP address ipa of another computer h[j] on the same Ethernet. Computer h[i] broadcasts a rqst(ipa) message over the Ethernet. If h[j] happens to be down at this time, then no rply(ipa, hda) message will be returned to h[i] and h[i] will not send an m(hda) message over the Ethernet. Thus, ARP ensures that no m(hda) message is sent over the Ethernet unless the destination computer of this message has been up shortly before m(hda) is sent.

The simplicity of ARP has made it widely used in the Internet. Unfortunately, this simplicity makes ARP vulnerable to two types of spoofing attacks. To describe these two types of ARP spoofing attack, consider a scenario where an adversary computer h[k], which is on the same Ethernet as computer h[i], wants to stop the communication of h[i] with the outside world. Thus, h[k] sends forged ARP reply messages to poison the ARP caches of h[i] and all other computers on the Ethernet. There are two cases to consider.

I. *Stopping Inbound Traffic*:

In this case, h[k] sends to all the computers of the Ethernet except h[i] a spoofed rply(ipa, hda), in which ipa is the IP address of h[i], and hda is a nonexistent hardware address. Every computer that receives this spoofed rply(ipa, hda) message caches this nonexistent hda for h[i], and as a result, all future messages destined for h[i] will not be delivered to h[i].

II. *Stopping Outbound Traffic*:

In this case, h[k] sends to h[i] a spoofed rply(ipa, hda), in which ipa is the IP address of the default router of the Ethernet, and hda is the hardware address of h[k]. Once computer h[i]'s cache is poisoned by this spoofed rply(ida, hda), all future outbound messages of computer h[i] are delivered to h[k] rather than to the default router. (The adversary h[k] can also forward these outbound messages of h[i] to the default router after it reads them. This constitutes a man-in-the-middle attack [52].)

In both cases, the adversary h[k] poisons the ARP caches of other computers such that the real next-hop destinations of their messages become unreachable. Therefore, the Detection of Next-Hop Failure condition of hop integrity is violated in this network.

In order to counter these ARP spoofing attacks two solutions have been proposed recently. In one solution, a tool called ARPWATCH [32] is proposed to monitor the activities over the Ethernet (such as the transmission of rqst(ipa) and rply(ipa, hda) messages over the Ethernet) and check these activities against a database of (IP address, hardware address) pairings. In another solution, permanent entries for trusted hosts [1, 51] are permanently stored in the ARP caches in all computers on the Ethernet, so that rqst(ipa) and rply(ipa, hda) messages are not sent over the Ethernet and ARP spoofing is prevented. Both of these solutions suffer from some problems. ARPWATCH supports two functions of ARP, namely resolving IP addresses and detecting destination failures, but it does not support the dynamic assignment of IP addresses. In the case of permanent entries for trusted hosts, detecting destination failures and dynamically assigning addresses are not supported. Moreover, neither of the two solutions can overcome transmission inducement attack as discussed in [16].

By contrast, our secure address resolution protocol, which will be presented in Chapter 5, can support all the three functions of ARP, and can defeat both ARP spoofing attack and transmission inducement attack.

2. RESOURCE-EXHAUSTING ATTACKS

Most known denial-of-service attacks belong to the fashion of exhausting the resources of the target systems. In this type of attacks, an adversary sends successive huge waves of messages to the target host in order to exhaust its computing resources and the bandwidth of its connection link. Smurf attack and SYN attack, as described in Chapter 1, both belong to this type.

A common characteristic of attacks of this type is that messages inserted by the adversary carry wrong original sources. However, adversaries of these attacks put wrong original sources in their attacking messages for different reasons. In Smurf attack, the original source that an adversary puts in the ping messages is the IP address of the target host, such that each computer that receives a copy of this ping message sends a pong message to the target host. In SYN attack, the original sources that an adversary puts in the SYN messages are IP addresses of hosts that are either down or unreachable at present. This is because if the original sources in these attacking SYN messages belong to some up and reachable hosts, then these hosts will

receive a SYN-ACK message from the target host, and will return a RESET message to the target host so as to inform it that they did not send any SYN message to the target host before. As a result, the target host is able to release the resources that were reserved for the expected connections and foil the attack. Moreover, an adversary of any denial-of-service attack tends to put a forged source address in its attacking messages, such that the identity and location of the adversary will not be easily determined.

The recent years have seen the emergence of distributed denial-of-service attack [45], an even nastier type of denial-of-service attacks. This attack is called "distributed" because an adversary does not send out the attacking messages by itself. Instead, the adversary intrudes a multitude of unprotected hosts over the Internet and installs its attacking software in these unprotected hosts. These intruded hosts are called "zombies". The adversary can launch an attack against a computer d on the Internet as follows. First, the adversary sends a command to all the zombies at the same time to initiate the software it installed in the zombies previously. Second, after receiving the command from the adversary, each zombie launches a denial-of-service attack, for example Smurf attack or SYN attack, against computer d. As a result, computer d is flooded by messages from all the zombies.

In order to curb this type of denial-of-service attacks that involve messages with wrong original sources, Ferguson and Senie proposed a technique called ingress filtering [12]. Using ingress filtering, each router checks whether the recorded source in each received message is consistent with the subnetwork from which the router received the message. (A router is connected to two or more subnetworks. It can determine which subnetwork a message comes from by the incoming interface of the message.) When a router receives a message, there are two cases for the router to consider: the received message is from a subnetwork with no other router connected to it, or the received message is from a subnetwork with one or more adjacent routers.

If the received message is from a subnetwork with no other router connected to it, then the router checks if the recorded message source is consistent with the address prefix of the subnetwork. If so, then the message is supposedly from a host on that subnetwork and the router forwards the message as usual. Otherwise, the router discards the message. Therefore, if an adversary inserts messages with forged sources into a subnetwork with only one router connected to it, then these inserted messages will be detected and discarded by ingress filtering.

However, if the received message is from a subnetwork with one or more adjacent routers, the situation is more complex. If the router finds that the recorded source of the received message is not consistent with the address prefix of the subnetwork from which the message is received, then there are

two possible cases to consider: either the message is forwarded by an adjacent router, or the message is inserted by a host that is connected to the subnetwork and is compromised by an adversary. (An adversary may try to insert its messages with forged sources through a compromised host on the subnetwork, hoping to convince the receiving router that this message is forwarded by an adjacent router.) Ingress filtering cannot distinguish the above two cases, therefore it is not effective in stopping denial-of-service attacks that insert messages into a subnetwork with two or more routers.

In order for a network to counter such denial-of-service attacks that insert messages with forged sources into a subnetwork with two or more routers, the network needs to satisfy the second condition of hop integrity: Detection of Message Modification. That is, whenever a router q on the network receives a message m supposedly from an adjacent router p, router q can correctly determine whether message m was modified or inserted by an adversary. Our weak hop integrity protocol and strong hop integrity protocol, which will be presented in Chapters 6 and 7, can detect message modification, and therefore can detect and discard the messages inserted by an adversary.

Chapter 5

SECURE ADDRESS RESOLUTION PROTOCOL

In this chapter, we present the secure address resolution protocol. The secure address resolution protocol requires a secure server connected to the Ethernet, and consists of two sub-protocols: an invite-accept protocol and a request-reply protocol.

This chapter is organized as follows. In Section 5.1, we introduce the architecture of secure address resolution, and show that this architecture can counter ARP spoofing attacks discussed in Section 4.1. Then, in Sections 5.2 and 5.3, we present the invite-accept protocol and the request-reply protocol respectively. Finally, we discuss four extensions to the secure address resolution protocol in Section 5.4.

1. ARCHITECTURE OF SECURE ADDRESS RESOLUTION

To perform secure address resolution in an Ethernet, a secure server s is added to the Ethernet. Then, every communication concerning address resolution in this Ethernet is either from s to some computer in the Ethernet, or from some computer in the Ethernet to s.

The secure address resolution protocol between s and a computer h[i] in the Ethernet consists of two sub-protocols: the invite-accept protocol and the request-reply protocol. The function of the invite-accept protocol is to allow the secure server s to invite the different computers in the Ethernet to register, periodically and securely, their IP addresses and hardware addresses in the secure server. The function of the request-reply protocol is to allow each computer in the Ethernet to request the secure server s to resolve an IP address of some other computer in the same Ethernet to its hardware

address. As shown in Figure 5.1, the invite-accept protocol is between process sn in server s and process hn[i] in computer h[i], and the request-reply protocol is between process sr in server s and process hr[i] in computer h[i].

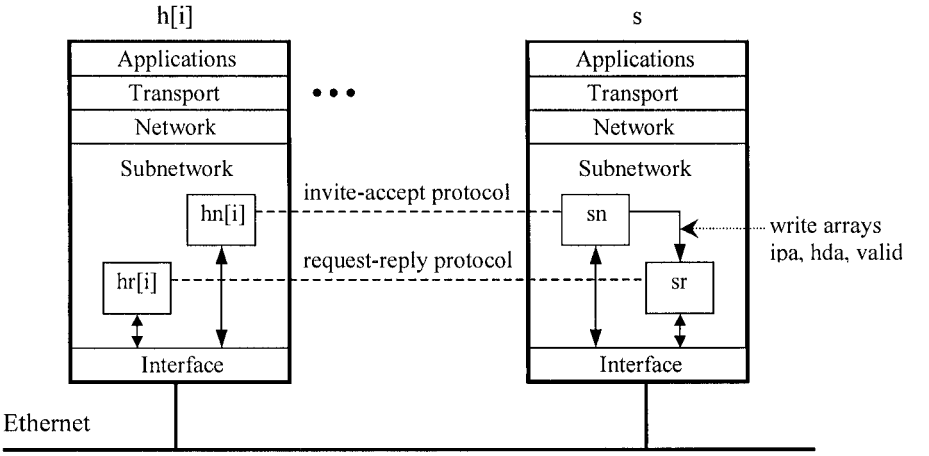


Figure 5-1. Architecture of secure address resolution.

Both the invite-accept protocol and the request-reply protocol are designed to tolerate the actions of any adversary that happens to be on the Ethernet. We assume that an adversary can perform the following three types of actions a finite number of times to disrupt the communications between server s and any computer h[i] on the Ethernet.

I. *Message Loss*:
After a message is sent (by a process in s or h[i]), the message is discarded by the adversary, and is never received (by the intended process in h[i] or s, respectively).

II. *Message Modification*:
After a message is sent and before it is received, the message fields are arbitrarily modified by the adversary.

III. *Message Replay*:
After a message is sent and before it is received, the message is replaced by a copy of an earlier message of the same type by the adversary.

Note that by executing a sequence of these adversarial actions, the adversary can launch the ARP spoofing attacks presented in Section 2.1. Let

us consider again the scenario where an adversary computer h[k], which is on the same Ethernet as its target computer h[i], wants to stop the communication of h[i] with the outside world. First, in order to stop the inbound traffic of h[i], h[k] modifies some ARP reply messages that are destined to all the computers of the Ethernet except h[i] such that the modified ARP reply messages become rply(ipa, hda), in which ipa is the IP address of h[i], and hda is a nonexistent hardware address. Also, h[k] discards a finite number of ARP reply messages that contain the IP address and the correct hardware address of h[i]. The net effect is that every computer that receives the spoofed rply(ipa, hda) message caches this nonexistent hda for h[i] for some time, and as a result, all messages destined for h[i] will not be delivered to h[i] for some time. Second, in order to stop the outbound traffic of h[i], h[k] modifies an ARP reply message destined to h[i] such that the message becomes rply(ipa, hda), in which ipa is the IP address of the default router of the Ethernet, and hda is the hardware address of h[k]. Also, h[k] discards a finite number of ARP reply messages that contain the IP address and the correct hardware address of the default router of the Ethernet. Once computer h[i]'s cache is poisoned by this spoofed rply(ida, hda), all future outbound messages of computer h[i] are delivered to h[k] rather than to the default router until the poison entry in h[i]'s cache expires.

Next, we illustrate how our secure address resolution architecture counters the adversarial actions. In our design, the invite-accept protocol and the request-reply protocol use the following three mechanisms to tolerate the three types of adversarial actions:

I. *Timeout Actions to Counter Message Loss*:

If a process (in s or h[i]) sends a message and does not receive a reply for this message for a relatively long time, the process times out and sends another copy of the same message or sends another message.

II. *Shared Secrets to Counter Message Modification*:

Server s shares a unique secret scr[i] with each computer h[i] on the Ethernet. This secret is used to compute a piece of integrity check information to be added to each message that is sent between s and h[i]. For example, assume that a message acpt(c, ip, hd), with three fields c, ip, and hd, is to be sent between s and h[i]. Then an integrity check d for this message can be computed as follows:

$$d := MD(c; ip; hd; scr[i])$$

where MD is a message digest function, and "c; ip; hd; scr[i]" is a concatenation of the three message fields and the shared secret. This

integrity check d is added to the message, to become acpt(c, ip, hd, d), before sending it so that if the message fields are arbitrarily modified (by the adversary) to become acpt(c′, ip′, hd′, d′), then d′ is no loger equal to MD(c′; ip′; hd′; scr[i]). Thus, arbitrarily modifying the fields of a message can be detected by the message receiver.

Note that shared secrets used by the protocols in our secure address resolution architecture are based on the following assumption: Every computer on the Ethernet has secure access to the secret it shares with the secure server and does not reveal the shared secret to any other computer on this Ethernet. Otherwise, if an adversary gets to know the secret shared between server s and computer h[i] on the Ethernet, then this adversary can impersonate h[i] to communicate with s, or it can impersonate s to communicate with h[i], and messages sent from the adversary to s or h[i] will not be detected.

III. *Nonces to Counter Message Replay*:

Before a process (in s or h[i]) sends a message that requires a reply to another process (in h[i] or s, respectively), the sending process attaches to the message a unique integer nc, called the message nonce. When the receiving process receives the message and prepares a reply, it attaches the message nonce nc to the reply. Finally, when the sending process receives the reply and checks that the message nonce is the same as that in the original message, it concludes correctly that neither the original message nor the reply were replaced by earlier messages (by the adversary).

We argue that ARP spoofing attack cannot succeed under our secure address resolution architecture. Note that using our secure address resolution architecture, all messages regarding address resolution are exchanged between server s and computer h[i], rather than between computer h[i] and other computers on the Ethernet. Therefore, in order to launch an ARP spoofing attack against computer h[i], adversary h[k] has to try to modify address resolution messages between s and h[i]. However, the attempt by h[k] will not succeed because of the following two reasons. First, each message of the invite-accept protocol between s and h[i] is protected by an integrity check computed using the secret shared between s and h[i]. Thus, h[k] cannot poison the hardware address of h[i] stored in server s because h[k] does not know the secret shared between s and h[i]. Second, each message of the request-reply protocol between s and h[i] is protected by an integrity check computed using the secret shared between s and h[i]. Thus, h[k] cannot fool h[i] by sending h[i] a forged reply message because h[k] does not know the secret shared between s and h[i].

In the next two sections, we describe in some detail the two protocols: the invite-accept protocol and the request-reply protocol, and discuss their correctness proofs.

2. THE INVITE-ACCEPT PROTOCOL

The invite-accept protocol consists of process sn in server s and every process hn[i] in computer h[i]. Process sn shares a unique secret scr[i] with every process hn[i], and it stores the shared secrets in a constant array scr[0 .. n-1]. This array is defined as a constant in process sn because the actions of sn can read this array but cannot update it. (The initial shared secret of a host can be assigned to this host along with its IP address when the host is added to the Ethernet. The shared secret can be renewed once in a long period, for example a month.)

Process sn also maintains three variable arrays ipa[0 .. n-1], hda[0 .. n-1], and valid[0 .. n-1]. Array ipa[0 .. n-1] and array hda[0 .. n-1] are used to record the IP addresses and hardware addresses of all computers on the Ethernet. Array valid[0 .. n-1] is the validity count for the entries in arrays ipa[0 .. n-1] and hda[0 .. n-1]. When sn writes ipa[i] and hda[i], valid[i] is assigned its highest possible value vmax. Periodically, sn decrements valid[i] by one. If the value of valid[i] ever becomes zero, then the current values of ipa[i] and hda[i] are no longer valid.

There are two types of messages in the invite-accept protocol: invite and accept messages. The invite messages are sent from process sn to every process hn[i], whereas the accept messages are sent from every process hn[i] to process sn. Every T seconds, process sn sends an invite message to every process hn[i]. Then every hn[i] replies by sending an accept message to s.

Each invite message is of the form invt(nc, md), where nc is the unique nonce of the message and md is a list md[0], ... , md[n-1] of message digests. Before sending an invt(nc, md) msg, process sn computes a unique value for nc, and computes every md[i] as follows:

nc := NONCE;
for every i, $0 \leq i < n$, md[i] := MD(nc; scr[i])

where NONCE is a function that when invoked returns a fresh nonce.

When a process hn[i] receives an invt(nc, md) message, it computes the value MD(nc; sc) and compares the computed value with the received value md[i] in the message. If they are equal, then hn[i] concludes correctly that this message was indeed sent by sn, and sends an accept message to sn. Otherwise, hn[i] discards the received invite message.

Each accept message, sent by a process hn[i], is of the form acpt(c, x, y, d), where c is the message nonce that hn[i] found in the last received invite message, x is the IP address of hn[i], y is the hardware address of hn[i], and d is the message digest computed by hn[i] as follows:

d := MD(c; x; y; sc)

where sc is the secret that h[i] shares with server s.

When process sn receives an acpt(c, x, y, d) message from a process hn[i], it checks that c equals the nonce nc in the last invite message sent by sn and that d is a correct digest for the accept message. If so, sn concludes correctly that the accept message was indeed sent by hn[i] and stores x in ipa[i] and stores y in hda[i]. Otherwise, sn discards the accept message. Process sn can be defined as follows.

```
process sn
const  scr      :  array [0 .. n-1] of integer  {shared secrets}
       T        :  integer     {T ≥ round trip delay between}
                               {sn and each hn[i]}
       vmax     :  integer
var    ipa      :  array [0 .. n-1] of integer
       hda      :  array [0 .. n-1] of integer
       valid    :  array [0 .. n-1] of 0 .. vmax
       md       :  array [0 .. n-1] of integer
       nc, c, d :  integer
       x, y     :  integer
       j        :  0 .. n
par    i        :  0 .. n-1
begin
    timeout (T seconds passed since this action executed last) →
        nc := NONCE;
        j := 0;
        do j < n →
            md[j] := MD(nc; scr[j]);
            valid[j] := max(0, valid[j] – 1);
            j := j + 1
        od;
        send invt(nc, md) to hn

[]  rcv acpt(c, x, y, d) from hn[i] →
        if  c = nc  ∧  d = MD(c; x; y; scr[i]) →
            ipa[i] := x;
            hda[i] := y;
```

```
                valid[i] := vmax
            [] c ≠ nc ∨ d ≠ MD(c; x; y; scr[i]) →
                {discard message}
                skip
            fi
end
```

Process sn has two actions. In the first action, sn broadcasts an invite message to every process hn[i] on the Ethernet every T seconds. In the second action, process sn receives an accept message from a process hn[i], checks that the message is correct, and if so, it stores the IP address and hardware address contained in the accept message in ipa[i] and hda[i].

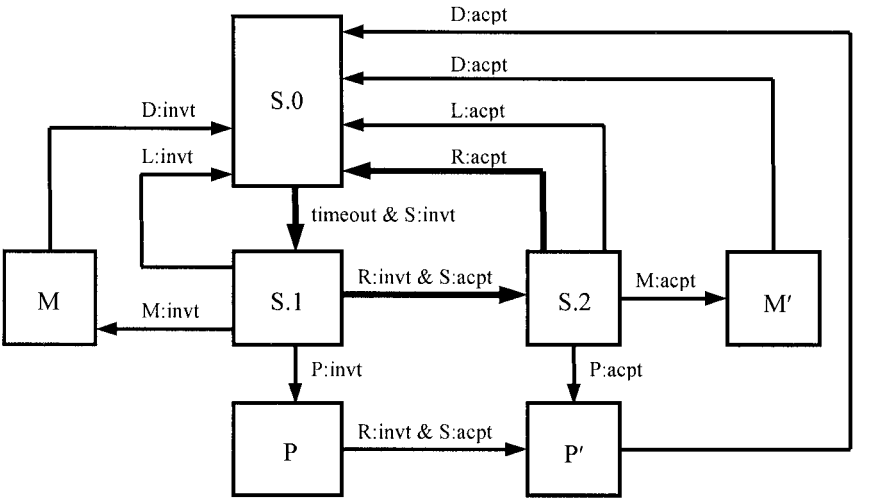
Note that when sn broadcasts an invite message, it decrements the value of every valid[i] by one, and when sn receives an accept message from hn[i] and checks that the message is correct, it resets the value of valid[i] to vmax. Thus, if sn does not receive any accept message from hn[i] for vmax * T seconds, then valid[i] becomes 0 in sn.

Process hn[i] stores the secret it shares with process sn in a constant named sc. (Thus, the value of sc in hn[i] equals the value of scr[i] in sn.) Process hn[i] has two other constants, namely ip and hd, that stores the IP address and the hardware address of computer h[i], respectively. Process hn[i] can be defined as follows.

```
process hn[i : 0 .. n-1]
const  sc       :  integer        {sc in hn[i] = scr[i] in sn}
       ip, hd   :  integer
var    e        :  array [0 .. n-1] of integer
       c, d     :  integer
begin
    rcv invt(c, e) from sn →
        d := MD(c; sc);
        if d = e[i] →
            d := MD(c; ip; hd; sc);
            send acpt(c, ip, hd, d) to sn
        [] d ≠ e[i] →
            {discard message}
            skip
        fi
end
```

To verify the correctness of the invite-accept protocol, we can use the state transition diagram of this protocol in Figure 5.2. This diagram has

seven nodes that represent all possible reachable states of the protocol. Every transition in the diagram stands for either a legitimate action (of process sn or process hn[i]), or an illegitimate action of the adversary.



S.0 = ch.sn.hn[i] = < > ∧ ch.hn[i].sn = < >

S.1 = ch.sn.hn[i] = <invt(c, e)> ∧ c = nc ∧ e[i] = md[i] ∧ e[i] = MD(c; scr[i]) ∧
ch.hn[i].sn = < >

S.2 = ch.sn.hn[i] = < > ∧
ch.hn[i].sn = <acpt(c, x, y, d)> ∧ c = nc ∧ d = MD(c; x; y; sc)

M = ch.sn.hn[i] = <invt(c, e)> ∧ e[i] ≠ MD(c; scr[i]) ∧
ch.hn[i].sn = < >

M′ = ch.sn.hn[i] = < > ∧
ch.hn[i].sn = <acpt(c, x, y, d)> ∧ d ≠ MD(c; x; y; sc)

P = ch.sn.hn[i] = <invt(c, e)> ∧ c ≠ nc ∧ e[i] ≠ md[i] ∧ e[i] = MD(c; scr[i]) ∧
ch.hn[i].sn = < >

P′ = ch.sn.hn[i] = < > ∧
ch.hn[i].sn = <acpt(c, x, y, d)> ∧ c ≠ nc

Figure 5-2. State transition diagram of the invite-accept protocol.

For convenience, each transition is labeled by the message event that is executed during the transition. In particular, each transition has a label of the form

<event type> : <message type>

where <event type> is one of the following:

S stands for sending a message of the specified type
R stands for receiving and accepting a message of the specified type
D stands for receiving and discarding a message of the specified type
L stands for losing a message of the specified type
M stands for modifying a message of the specified type
P stands for replaying a message of the specified type

Initially, the network starts at a state S.0 where the two channels between processes sn and hn[i] are empty. This state can be defined by the following predicate

S.0 : ch.sn.hn[i] = < > $\wedge$ ch.hn[i].sn = < >

At state S.0, exactly one action, namely the timeout action in process sn, is enabled for execution. Executing this action at state S.0 leads the network to state S.1 defined as follows.

S.1 : ch.sn.hn[i] = <invt(c, e)> $\wedge$ c = nc $\wedge$ e[i] = md[i] $\wedge$
e[i] = MD(c; scr[i]) $\wedge$ ch.hn[i].sn = < >

Note that in state S.1, the channel from process sn to process hn[i] has only one message: invt(c, e), where the following three conditions hold. First, the value of field c in the message equals the value of variable nc in sn. Second, the i^{th} element in array e in the message equals the i^{th} element in array md in sn. Third, the i^{th} element in array e equals the message digest of the concatenation of the value of field c and the i^{th} element in array scr in sn.

At state S.1, exactly one legitimate action, namely the receive action in process hn[i], is enabled for execution. Executing this action at state S.1 leads the network to state S.2 defined as follows.

S.2 : ch.sn.hn[i] = < > $\wedge$
ch.hn[i].sn = <acpt(c, x, y, d)> $\wedge$ c = nc $\wedge$ d = MD(c; x; y; sc)

Note that in state S.2, the channel from process hn[i] to process sn has only one message: acpt(c, x, y, d), where the following two conditions hold. First, the value of field c in the message equals the value of variable nc in sn. Second, the value of field d in the message equals the message digest of the concatenation of the values of fields c, x, y, and the value of constant sc in hn[i].

At state S.2, exactly one legitimate action, namely the receive action in process sn, is enabled for execution. Executing this action at S.2 leads the network back to S.0 defined above.

States S.0, S.1 and S.2 are called *good states* because the transitions between these states only involve the legitimate actions of processes sn and hn[i]. The sequence of the transitions from state S.0 to state S.1, from state S.1 to state S.2, and from state S.2 to state S.0, constitutes the *good cycle* in which the network performs progress. If only legitimate actions of processes sn and hn[i] are executed, the network will stay in this good cycle indefinitely. Next, we discuss the bad effects caused by the actions of an adversary, and how the network can recover from bad states to good states.

First, the adversary can execute a message loss action at state S.1 or S.2. If the adversary executes a message loss action at S.1, the only message in the channel from process sn to process hn[i] is removed. If the adversary executes a message loss action at S.2, the only message in the channel from hn[i] to sn is removed. In either case, the network returns to state S.0 where both channels are empty.

Second, the adversary can execute a message modification action at state S.1 or S.2. If the adversary executes a message modification action at S.1, the network moves to state M where the i^{th} element of array e in message invt(c, e) is not equal to the message digest of the concatenation of c and scr[i]. This message invt(c, e) will be received and discarded by hn[i] because it cannot pass the integrity check in the receive action of hn[i]. If the adversary executes a message modification action at S.2, the network moves to state M′ where the value of field d in message acpt(c, x, y, d) is not equal to the message digest of the concatenation of the values of fields c, x, y in the message and constant sc in hn[i]. This message acpt(c, x, y, d) will be received and discarded by sn because it cannot pass the integrity check in the receive action of sn. In either case, the network returns to state S.0.

Third, the adversary can execute a message replay action at state S.1 or S.2. If the adversary executes a message replay action at S.1, the network moves to state P where the value of field c in message invt(c,e) is not equal to the value of variable nc in sn, the i^{th} element of array e in the message is not equal to the i^{th} element of array md in sn, but the i^{th} element of array e is equal to the message digest of the concatenation of the values of field c in the message and the i^{th} element of constant array scr in sn. This message invt(c, e) will be received by hn[i] and it will pass the integrity check in the receive action of hn[i]. Then, hn[i] sends a message acpt(c, x, y, d) to sn, and the network enters state P′ where the value of field c in message acpt(c, x, y, d) is not equal to the value of variable nc in sn. This message acpt(c, x, y, d) will be received and discarded by sn because it cannot pass the integrity check in the receive action of sn, and the network returns to state S.0 where

both channels are empty. If the adversary executes a message replay action at S.2, the network moves to state P′ as described above. Then, the message acpt(c, x, y, d) will be received and discarded by sn, and the network returns to S.0.

From the state transition diagram, it is clear that each imposed illegitimate action by the adversary will eventually lead the network back to S.0, which is a good state. Once the network enters a good state, the network can make progress in the good cycle. Hence the following two theorems about the invite-accept protocol are proved.

Theorem 5.1 *In the absence of an adversary, a network that executes the invite-accept protocol will follow the good cycle, consisting of the transitions from state S.0 to state S.1, from state S.1 to state S.2, and from state S.2 to state S.0, and will stay in this good cycle indefinitely.*

Theorem 5.2 *In the presence of an adversary, a network that executes the invite-accept protocol will converge to the good cycle in a finite number of steps after the adversary finishes executing the message loss, message modification, and message replay actions.*

3. THE REQUEST-REPLY PROTOCOL

Next, we outline the operation of the request-reply protocol as follows. When a computer h[i] wants to send a message m to any other computer h[j] on the same Ethernet and thus needs to resolve the IP address of h[j] into its corresponding hardware address, h[i] can use the request-reply protocol to send a request message to server s. Then server s replies by sending a reply message to h[i]. If valid[j] in s is positive, which indicates h[j] has been up shortly before s receives the request message, s sends h[i] a reply message that contains the hardware address of h[j]. Otherwise, s sends h[i] a reply message with no hardware address in it. Therefore, h[i] does not send a message to h[j] over the Ethernet unless h[j] has been up shortly before the message is sent. Similarly, with the secure address resolution protocol suite installed in the subnetwork, router p does not send any message to router q over the Ethernet connecting p and q unless router q has been up shortly before the message is sent. Consequently, the Detection of Next-Hop Failure condition is attained.

The request-reply protocol consists of process sr in server s and every process hr[i] in computer h[i]. Process sr in server s shares the same unique secret with process hr[i] in computer h[i] as shared between processes sn and hn[i] in the invite-accept protocol.

There are two types of messages in the request-reply protocol: request and reply messages. The request messages are sent from process hr[i] to process sr, whereas the reply messages are sent from process sr to process hr[i]. When process hr[i] needs to resolve an IP address into its corresponding hardware address, and hr[i] is not waiting for a reply message for a previous request message, hr[i] sends a request message to process sr. Then sr replies by sending a reply message to process hr[i].

Each request message is of the form rqst(nc, dst, d), where nc is the unique nonce of the message, dst is the IP address of the destination computer process hr[i] needs to resolve, and d is a message digest computed by hr[i]. Before sending a rqst(nc, dst, d) msg, process hr[i] computes a unique value for nc, and computes d as follows:

nc := NONCE;
d := MD(nc; dst; sc)

When process sr receives a rqst(nc, dst, d) message, it computes the value MD(nc; dst; scr[i]) and compares the computed value with the received value d in the message. If they are equal, then sr concludes correctly that this message was indeed sent by hr[i], searches its database for the corresponding hardware address of dst, and sends a reply message to hr[i]. Otherwise, sr discards the received request message.

Each reply message, sent by process sr, is of the form rply(c, x, y, d), where c is the message nonce that sr found in the last received request message, x is the IP address of the destination computer requested by hr[i], y is the corresponding hardware address of x, and d is the message digest computed by sr as follows:

d := MD(c; x; y; scr[i])

where scr[i] is the secret that server s shares with computer h[i].

When process hr[i] receives a rply(c, x, y, d) message from process sr, it checks that c equals the nonce nc in the last request message sent by hr[i], that x equals dst in the last request message sent by hr[i], and that d is a correct digest for the reply message. If so, hr[i] concludes correctly that the reply message was indeed sent by sr and takes y as the hardware address of the destination computer. Otherwise, hr[i] discards the reply message. Process hr[i] can be defined as follows.

```
process hr[i : 0 .. n-1]
const  sc       : integer      {sc in hr[i] = scr[i] in sr}
       t        : integer
var    nc, c, d : integer
       dst, x, y: integer
```

```
        wait    :  boolean
begin
    ~ wait →
        wait := true;
        nc := NONCE;
        dst := any;
        d := MD(nc; dst; sc);
        send rqst(nc, dst, d) to sr

[]  rcv rply(c, x, y, d) from sr →
        if nc = c ∧ dst = x ∧ d = MD(c; x; y; sc) →
            {y is requested information about x}
            wait := false
        [] nc ≠ c ∨ dst ≠ x ∨ d ≠ MD(c; x; y; sc) →
            {discard message}
            skip
        fi

[]  timeout wait ∧ (t seconds passed since first action executed last) →
        d := MD(nc; dst; sc);
        send rqst(nc, dst, d) to sr
end
```

Process hr[i] has three actions. In the first action, process hr[i] sends a request message to process sr while not waiting. In the second action, hr[i] receives a reply message from sr, and derives the hardware address of the destination computer. In the third action, hr[i] times out after waiting for t seconds, and resends the same request message to sr.

Note that in the second action, process hr[i] checks both field c and field x in message rply(c, x, y, d) to see if they are equal to the values of nc and dst respectively. The purpose of this double-checking is to make sure that the reply message corresponds to the request message for which hr[i] is waiting for a reply, and that the hardware address contained in the reply message corresponds to the IP address hr[i] needs to resolve, and also to make it harder for the adversary to modify the message.

Process sr can read (but not write) the three arrays ipa[0 .. n-1], hda[0 .. n-1], and valid[0 .. n-1] that are updated regularly by process sn of the invite-accept protocol. Process sr can be defined as follows.

```
process sr
const  scr  :  array [0 .. n-1] of integer
       ipa  :  array [0 .. n-1] of integer
```

```
        hda   :  array [0 .. n-1] of integer
        valid :  array [0 .. n-1] of integer
var     c, d  :  integer
        x     :  integer
        j     :  0 .. n
par     i     :  0 .. n-1
begin
    rcv rqst(c, x, d) from hr[i] →
        if d = MD(c; x; scr[i]) →
            j := 0;
            do ipa[j] ≠ x ∧ j < n →
                j := j + 1
            od;
            if j < n ∧ valid[j] > 0 →
                d := MD(c; x; hda[j]; scr[i]);
                send rply(c, x, hda[j], d) to hr[i]
            [] j = n ∨ valid[j] = 0 →
                d := MD(c; x; 0; scr[i]);
                send rply(c, x, 0, d) to hr[i]
            fi
        [] d ≠ MD(c; x; scr[i]) →
            {discard message}
            skip
        fi
end
```

Process sr has only one action, in which sr receives a request message from a process hr[i] and sends a reply message to hr[i].

Note that when process sr receives a request message from process hr[i], it first checks the integrity of the message. Then, sr searches array ipa for the IP address that hr[i] requests to resolve. If the requested IP address exists in array ipa and the validity count for it is larger than 0, then sr sends a reply message, containing the corresponding hardware address, to hr[i]. If the requested IP address does not exist in array ipa or the validity count is equal to 0, then sr sends a reply message, containing an empty hardware address, to hr[i].

To verify the correctness of the request-reply protocol, we can use the state transition diagram as shown in Figure 5.3. This diagram has eight states that represent all possible reachable states of the protocol. The predicates for each state in the diagram are shown in Figure 5.4.

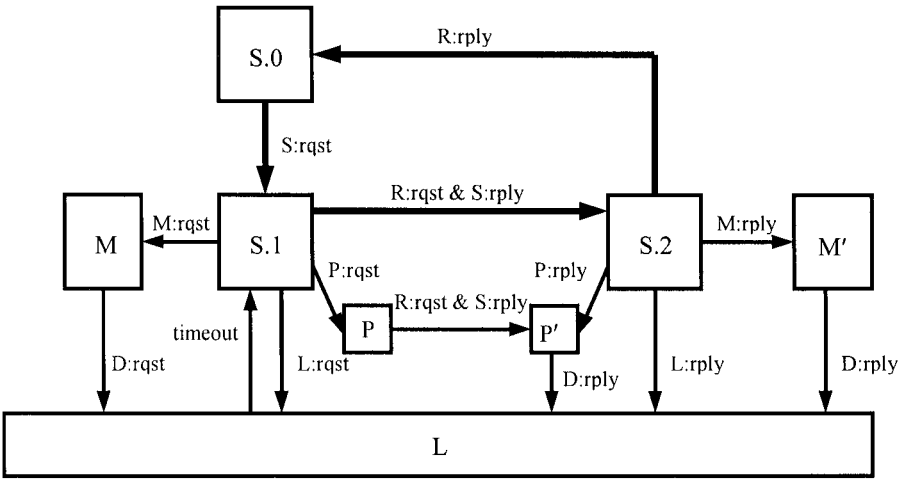


Figure 5-3. State transition diagram of the request-reply protocol.

Initially, the network starts at a state S.0 where the value of variable wait in process hr[i] is false and the two channels between processes hr[i] and sr are empty. At S.0, exactly one action, namely the first action in hr[i], is enabled for execution. Executing this action at S.0 leads the network to state S.1, where the channel from hr[i] to sr has only one message rqst(c, x, d). In this message, the value of field c equals the value of variable nc in hr[i], the value of field x equals the value of variable dst in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c, x, and the value of constant sc in hr[i].

At state S.1, exactly one legitimate action, namely the receive action in process sr, is enabled for execution. Executing this action at S.1 leads the network to state S.2, where the channel from sr to hr[i] has only one message rply(c, x, y, d). In this message, the value of field c equals the value of variable nc in hr[i], the value of field x equals the value of variable dst in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c, x, y, and the i^{th} element of constant array scr in sr.

At state S.2, exactly one legitimate action, namely the receive action in hr[i], is enabled for execution. Executing this action at S.2 leads the network back to S.0.

States S.0, S.1 and S.2 are the good states of the request-reply protocol, and the sequence of the transitions from S.0 to S.1, from S.1 to S.2, and from S.2 to S.0, constitutes the good cycle in which the network performs progress. Next, we discuss the bad effects caused by the actions of the adversary, and how the network can recover from bad states to good states.

S.0 = ~wait ∧ ch.hr[i].sr = < > ∧ ch.sr.hr[i] = < >

S.1 = wait ∧
ch.hr[i].sr = <rqst(c, x, d)> ∧ c = nc ∧ x = dst ∧ d = MD(c; x; sc) ∧
ch.sr.hr[i] = < >

S.2 = wait ∧
ch.hr[i].sr = < > ∧
ch.sr.hr[i] = <rply(c, x, y, d)> ∧ c = nc ∧ x = dst ∧ d = MD(c; x; y; scr[i])

M = wait ∧
ch.hr[i].sr = <rqst(c, x, d)> ∧ d ≠ MD(c; x; sc) ∧
ch.sr.hr[i] = < >

M′ = wait ∧
ch.hr[i].sr = < > ∧
ch.sr.hr[i] = <rply(c, x, y, d)> ∧ d ≠ MD(c; x; y; scr[i])

P = wait ∧
ch.hr[i].sr = <rqst(c, x, d)> ∧ c ≠ nc ∧ d = MD(c; x; sc) ∧
ch.sr.hr[i] = < >

P′ = wait ∧
ch.hr[i].sr = < > ∧
ch.sr.hr[i] = <rply(c, x, y, d)> ∧ c ≠ nc ∧ d = MD(c; x; y; scr[i])

L = wait ∧ ch.hr[i].sr = < > ∧ ch.sr.hr[i] = < >

Figure 5-4. Predicates for the states in the state transition diagram of the request-reply protocol.

First, the adversary can execute a message loss action at state S.1 or S.2. If the adversary executes a message loss action at S.1 or S.2, the network moves to state L where the value of variable wait in hr[i] is true and the two channels between hr[i] and sr are empty. After the timeout action, the network returns to S.1.

Second, the adversary can execute a message modification action at state S.1 or S.2. If the adversary executes a message modification action at S.1, the network moves to state M where the value of field d in message rqst(c, x, d) is not equal to the message digest of the concatenation of the values of fields c, x in the message and constant sc in hr[i]. This message rqst(c, x, d) will be received and discarded by sr because it cannot pass the integrity check. If the adversary executes a message modification action at S.2, the

network moves to state M′ where the value of field d in message rply(c, x, y, d) is not equal to the message digest of the concatenation of the values of fields c, x, y in the message and the i^{th} element of constant array scr in sr. This message rply(c, x, y, d) will be received and discarded by hr[i] because it cannot pass the integrity check. In either case, the network moves to state L next and eventually returns to S.1.

Third, the adversary can execute a message replay action at state S.1 or S.2. If the adversary executes a message replay action at S.1, the network moves to state P where the value of field c in message rqst(c, x, d) is not equal to the value of variable nc in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c and x in the message and constant sc in hr[i]. This message rqst(c, x, d) will be received and accepted by sr because it can pass the integrity check. Thus sr sends to hr[i] a message rply(c, x, y, d), and the network moves to state P′ where the value of field c in message rply(c, x, y, d) is not equal to the value of variable nc in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c, x, y, and the i^{th} element of constant array scr in sr. If the adversary executes a message replay action at S.2, the network moves to state P′ as well. From state P′, message rply(c, x, y, d) will be received and discarded by hr[i] because it cannot pass the integrity check, and the network moves to state L. Eventually, the network returns to S.1.

From the state transition diagram, it is clear that each imposed illegitimate action by the adversary will eventually lead the network back to S.1, which is a good state. Once the network enters a good state, the network can make progress in the good cycle. Hence the following two theorems about the request-reply protocol are proved.

Theorem 5.3 *In the absence of an adversary, a network that executes the request-reply protocol will follow the good cycle, consisting of the transitions from state S.0 to state S.1, from state S.1 to state S.2, and from state S.2 to state S.0, and will stay in this good cycle indefinitely.*

Theorem 5.4 *In the presence of an adversary, a network that executes the request-reply protocol will converge to the good cycle in a finite number of steps after the adversary finishes executing the message loss, message modification, and message replay actions.*

4. EXTENSIONS

In this section, we outline four extensions of the secure address resolution protocol. First, we extend the protocol to support insecure address

resolution for mobile computers that may visit an Ethernet but share no secrets with the secure server in that Ethernet. Second, we make the protocol more reliable by adding a backup server to its architecture. Third, we make the protocol perform some system diagnosis tasks. Fourth, we make the secure server act as a server for several Ethernets to which the server is attached.

4.1 Insecure Address Resolution

Consider an Ethernet that has several computers h[0 .. n-1] and a secure server s. Assume that these computers and server use the secure address resolution protocol (discussed above) to resolve IP addresses to hardware addresses. Assume also that mobile computers h[n .. r-1] visit this Ethernet but do not share any secret with the secure servers. In order that computers h[n .. r-1] can exchange messages with the other computers on this Ethernet, h[n .. r-1] need to use an "insecure" version of the address resolution protocol. Thus, server s needs to support two versions of the address resolution protocol: secure and insecure. If a message is due to insecure version of the address resolution protocol, then the information in the message is insecure. In particular, if a message comes from or will be sent to one of computers h[n .. r-1], or contains resolved address of one of computers h[n .. r-1], then the information in the message is insecure. Otherwise, if the message is due to secure version of the protocol, then the information in the message is secure.

The insecure version of the invite-accept protocol proceeds as follows. Whenever server s sends a invt(nc, md) to every computer in the Ethernet, computer h[i], where $n \leq i < r$, replies by sending back acpt(nc, x, y, d) message, where d has an arbitrary value, to server s. When server s receives a acpt(nc, x, y, d) message from computer h[i] and notices that h[i] is one of the mobile computers h[n .. r-1], it concludes that the message is insecure and so it does not attempt to check the correctness of the message digest d. Nevertheless, s stores in its database the IP address x and the hardware address y of computer h[i] along with an indication that this information is unreliable.

Process sn, process hn[0 .. n-1], and process hn[n .. r-1] in the invite-accept protocol with extension for insecure address resolution can be specified as follows.

```
process sn
const  scr  :  array [0 .. n-1] of integer  {shared secrets}
       T    :  integer                      {T ≥ round trip delay between}
                                            {sn and each hn[i]}
```

```
        vmax    :  integer
var     ipa     :  array [0 .. r-1] of integer  {r > n}
        hda     :  array [0 .. r-1] of integer
        valid   :  array [0 .. n-1] of 0 .. vmax
        md      :  array [0 .. n-1] of integer
        nc, c, d :  integer
        x, y    :  integer
        j       :  0 .. n
par     i       :  0 .. r-1
begin
    timeout (T seconds passed since this action executed last) →
        nc := NONCE;
        j := 0;
        do j < n →
            md[j] := MD(nc; scr[j]);
            valid[j] := max(0, valid[j] – 1);
            j := j + 1
        od;
        send invt(nc, md) to hn

[]  rcv acpt(c, x, y, d) from hn[i] →
        if i < n →
            if c = nc ∧ d = MD(c; x; y; scr[i]) →
                ipa[i] := x;
                hda[i] := y;
                valid[i] := vmax
            [] c ≠ nc ∨ d ≠ MD(c; x; y; scr[i]) →
                {discard message}
                skip
            fi
        [] n ≤ i < r →
            if c = nc →
                ipa[i] := x;
                hda[i] := y;
            [] c ≠ nc →
                {discard message}
                skip
            fi
        fi
end

process hn[i : 0 .. n-1]
```

```
const  sc       :  integer        {sc in hn[i] = scr[i] in sn}
       ip, hd   :  integer
var    e        :  array [0 .. n-1] of integer
       c, d     :  integer
begin
    rcv invt(c, e) from sn →
        d := MD(c; sc);
        if d = e[i] →
            d := MD(c; ip; hd; sc);
            send acpt(c, ip, hd, d) to sn
        [] d ≠ e[i] →
            {discard message}
            skip
        fi
end

process hn[i : n .. r-1]
const  ip, hd   :  integer
var    e        :  array [0 .. n-1] of integer
       c        :  integer
begin
    rcv invt(c, e) from sn →
        send acpt(c, ip, hd, 0) to sn
end
```

Note that the proof of correctness of secure version of the invite-accept protocol (between process sn and processes hn[0 .. n-1]) remains the same as we have shown in Figure 5.2. No proof can be derived for the insecure version (between process sn and processes hn[n .. r-1]), however, because nothing can be guaranteed for the messages exchanged between sn and hn[n .. r-1].

The insecure version of the request-reply protocol proceeds as follows. There are two cases to consider. First, server s may receive a rqst(nc, x, d) message from a computer h[i], where x is the IP address of computer h[j], and $0 \leq i < n$. In this case, s replies by sending a rply(nc, x, y, d) message to computer h[i], where y is the hardware address of computer h[j], and d is computed as follows: if $0 \leq j < n$, then d = MD(nc; x; y; scr[i]; 1) (the last bit "1" is used to indicate that y is secure information); if $n \leq j < r$, then d = MD(nc; x; y; scr[i]; 0) (the last bit "0" is used to indicate that y is insecure information). Second, server s may also receive a rqst(nc, x, d) message from a computer h[i], where x is the IP address of computer h[j], and $n \leq i < r$. In this case, s replies by sending a rply(nc, x, y, d) message to computer

h[i], where y is the hardware address of computer h[j], and d has an arbitrary value.

Process hr[0 .. n-1], process hr[n .. r-1], and process sr in the request-reply protocol with extension for insecure address resolution can be specified as follows.

```
process hr[i : 0 .. n-1]
const  sc      : integer      {sc in hr[i] = scr[i] in sr}
       t       : integer
var    nc, c, d : integer
       dst, x, y: integer
       wait    : boolean
begin
    ~ wait →
          wait := true;
          nc := NONCE;
          dst := any;
          d := MD(nc; dst; sc);
          send rqst(nc, dst, d) to sr

[]  rcv rply(c, x, y, d) from sr →
          if nc = c ∧ dst = x ∧ d = MD(c; x; y; sc; 1) →
                {y is secure information about x}
                wait := false
          [] nc = c ∧ dst = x ∧ d = MD(c; x; y; sc; 0) →
                {y is insecure information about x}
                wait := false
          [] nc ≠ c ∨ dst ≠ x ∨
             (d ≠ MD(c; x; y; sc; 1) ∧ d ≠ MD(c; x; y; sc; 0)) →
                {discard message}
                skip
          fi

[]  timeout wait ∧ (t seconds passed since first action executed last) →
          d := MD(nc; dst; sc);
          send rqst(nc, dst, d) to sr
end

process hr[i : n .. r-1]
const  t       : integer
var    nc, c, d : integer
       dst, x, y: integer
```

```
        wait    :  boolean
begin
    ~ wait →
        wait := true;
        nc := NONCE;
        dst := any;
        send rqst(nc, dst, 0) to sr

[]  rcv rply(c, x, y, d) from sr →
        if nc = c ∧ dst = x →
            {y is requested information about x}
            wait := false
        [] nc ≠ c ∨ dst ≠ x →
            {discard message}
            skip
        fi

[]  timeout wait ∧ (t seconds passed since first action executed last) →
        send rqst(nc, dst, 0) to sr
end

process sr
const  scr   :  array [0 .. n-1] of integer
       ipa   :  array [0 .. r-1] of integer
       hda   :  array [0 .. r-1] of integer
       valid :  array [0 .. n-1] of integer
var    c, d  :  integer
       x     :  integer
       j     :  0 .. r
par    i     :  0 .. r-1
begin
    rcv rqst(c, x, d) from hr[i] →
        if i < n →
            if d = MD(c; x; scr[i]) →
                j := 0;
                do ipa[j] ≠ x ∧ j < r →
                    j := j + 1
                od;
                if j < n ∧ valid[j] > 0 →
                    d := MD(c; x; hda[j]; scr[i]; 1);
                    send rply(c, x, hda[j], d) to hr[i]
                [] n ≤ j < r →
```

```
                        d := MD(c; x; hda[j]; scr[i]; 0);
                        send rply(c, x, hda[j], d) to hr[i]
                    [] j = r ∨ valid[j] = 0 →
                        d := MD(c; x; 0; scr[i]; 1);
                        send rply(c, x, 0, d) to hr[i]
                    fi
                [] d ≠ MD(c; x; scr[i]) →
                    {discard message}
                    skip
                fi
            [] n ≤ i < r →
                j := 0;
                do ipa[j] ≠ x ∧ j < r →
                    j := j + 1
                od;
                if (j < n ∧ valid[j] > 0) ∨ n ≤ j < r →
                    send rply(c, x, hda[j], 0) to hr[i]
                [] n ≤ j < r →
                    send rply(c, x, hda[j], 0) to hr[i]
                [] j = r ∨ valid[j] = 0 →
                    send rply(c, x, 0, 0) to hr[i]
                fi
            fi
end
```

Note that the proof of correctness of secure version of the request-reply protocol (between process sr and processes hr[0 .. n-1]) remains the same as we have shown in Figure 5.3, except that each appearance of conjunct "d = MD(c; x; y; scr[i])" in the predicates of S.2 and P′ needs to be replaced by "d = MD(c; x; y; scr[i]; 1) ∨ d = MD(c; x; y; scr[i]; 0)", and conjunct "d ≠ MD(c; x; y; scr[i])" in the predicate of M′ needs to be replaced by "d ≠ MD(c; x; y; scr[i]; 1) ∧ d ≠ MD(c; x; y; scr[i]; 0)". No proof can be derived for the insecure version (between process sr and processes hr[n .. r-1]), however, because nothing can be guaranteed for the messages exchanged between sr and hr[n .. r-1].

4.2 A Backup Server

The main problem of the secure address resolution protocol discussed above is that its secure server s represents a single point of failure. This problem can be resolved somewhat by adding a backup server bs to the Ethernet. Initially server bs is configured in a promiscuous mode so that it

receives a copy of every message sent over the Ethernet. Because server bs receives copies of all accept messages sent over the Ethernet, bs keeps its database up-to-date in the same way server s keeps its database up-to-date. (This necessitates that server bs is provided with all the secrets that server s shares with the computers on the Ethernet.)

Server bs sends no message as long as server s continues to send invite messages every T seconds over the Ethernet. If server bs observes that server s has not sent an invite message for vmax * T seconds, it concludes that server s has failed. In this case, bs reports the failure, and assumes the duties of s: it starts to send invite messages every T seconds and to send a reply message for every received request message.

4.3 System Diagnosis

In the secure address resolution protocol, the secure server s may conclude that some computer h[i] on the Ethernet has failed. This happens when s sends vmax consecutive invite messages and does not receive an accept message for any of them from computer h[i]. Thus, server s can be designed to report computer failures to the system administrator, whenever s detects such failures. In this case, system diagnosis becomes a side task of the secure address resolution protocol.

4.4 Serving Multiple Ethernets

The architecture of the secure address resolution protocol can be extended to allow s to act as a secure server for several Ethernets (rather than a single Ethernet) to which s is attached [8]. With this extension, the computers h[0 .. n-1] can be distributed over several Ethernets and n can become large. In the extended architecture, server s sends invite messages over the different Ethernets at the same time, then waits to receive accept messages over the different Ethernets. Also, each computer on an Ethernet can request (from server s) the hardware address of any other computer on the same Ethernet or on a different Ethernet.

Chapter 6

WEAK HOP INTEGRITY PROTOCOL

In this and the next two chapters, we present the hop integrity protocols. The hop integrity protocols belong to two thin layers, namely the secret exchange layer and the integrity check layer, that need to be added to the network layer of the protocol stack of each router in a network. The function of the secret exchange layer is to allow adjacent routers to periodically generate and exchange (and so share) new secrets. The exchanged secrets are made available to the integrity check layer, which uses them to compute and verify the integrity check for every data message transmitted between the adjacent routers.

Figure 6.1 shows the protocol stacks in two adjacent routers p and q. The secret exchange layer has one protocol: the secret exchange protocol. This protocol consists of the two processes pe and qe in routers p and q, respectively. The integrity check layer has two protocols: the weak integrity check protocol and the strong integrity check protocol. The weak version consists of the two processes pw and qw in routers p and q, respectively. This version can detect message modification, but not message replay. The strong version of the integrity check layer consists of the two processes ps and qs in routers p and q, respectively. This version can detect both message modification and message replay.

In this chapter, we present the weak hop integrity protocol, which is the combination of the secret exchange protocol and the weak integrity check protocol. In the next chapter, we present the strong hop integrity protocol, which is the combination of the secret exchange protocol and the strong integrity check protocol.

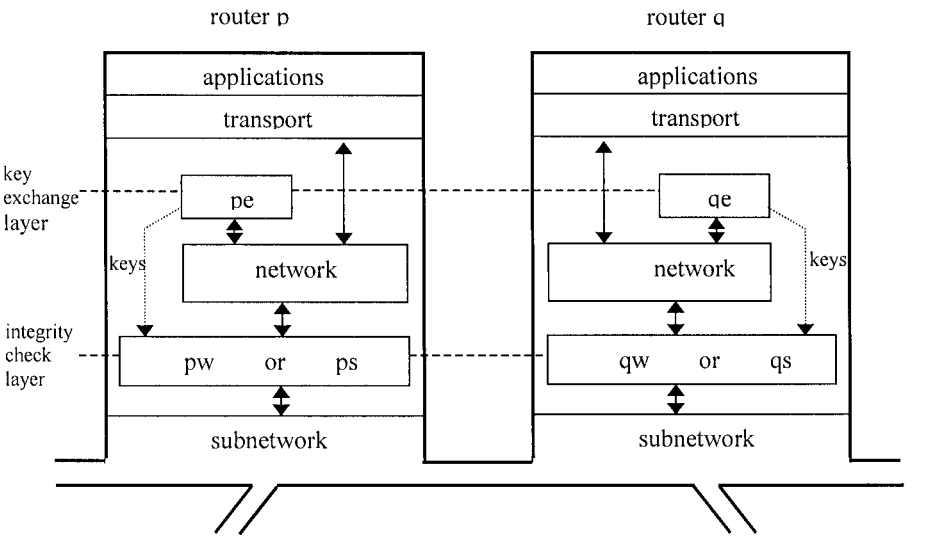


Figure 6-1. Protocol stack for hop integrity protocols.

This chapter is organized as follows. First, we present the secret exchange protocol, and verify its correctness. Then, we present the weak integrity check protocol, and verify its correctness.

1. SECRET EXCHANGE PROTOCOL

In the secret exchange protocol, the two processes pe and qe maintain two shared secrets sp and sq. Secret sp is used by router p to compute the integrity check for each data message sent by p to router q, and it is also used by router q to verify the integrity check for each data message received by q from router p. Similarly, secret sq is used by q to compute the integrity checks for data messages sent to p, and it is used by p to verify the integrity checks for data messages received from q.

As part of maintaining the two secrets sp and sq, processes pe and qe need to change these secrets periodically, say every te hours, for some chosen value te. Process pe is to initiate the change of secret sq, and process qe is to initiate the change of secret sp. Processes pe and qe each has a public key and a private key that they use to encrypt and decrypt the messages that

carry the new secrets between pe and qe. A public key is known to all processes (in the same layer), whereas a private key is known only to its owner process. The public and private keys of process pe are named B_p and R_p respectively; similarly, the public and private keys of process qe are named B_q and R_q respectively.

For process pe to change secret sq, the following four steps need to be performed. First, pe generates a new sq, and encrypts the concatenation of the old sq and the new sq using qe's public key B_q, and sends the result in a rqst message to qe. Second, when qe receives the rqst message, it decrypts the message contents using its private key R_q and obtains the old sq and the new sq. Then, qe checks that its current sq equals the old sq from the rqst message, and installs the new sq as its current sq, and sends a rply message containing the encryption of the new sq using pe's public key B_p. Third, pe waits until it receives a rply message from qe containing the new sq encrypted using B_p. Receiving this rply message indicates that qe has received the rqst message and has accepted the new sq. Fourth, if pe sends the rqst message to qe but does not receive the rply message from qe for some tr seconds, indicating that either the rqst message or the rply message was lost before it was received, then pe resends the rqst message to qe. Thus tr is an upper bound on the round trip time between pe and qe.

Note that the old secret (along with the new secret) is included in each rqst message and the new secret is included in each rply message to ensure that if an adversary modifies or replays rqst or rply messages, then each of these messages is detected and discarded by its receiving process (whether pe or qe).

Process pe has two variables sp and sq declared as follows.

```
var  sp :  integer
     sq :  array [0 .. 1] of integer
```

Similarly, process qe has an integer variable sq and an array variable sp.

In process pe, variable sp is used for storing the secret sp, variable sq[0] is used for storing the old sq, and variable sq[1] is used for storing the new sq. The assertion sq[0] ≠ sq[1] indicates that process pe has generated and sent the new secret sq, and that qe may not have received it yet. The assertion sq[0] = sq[1] indicates that qe has already received and accepted the new secret sq. Initially,

sq[0] in pe = sq[1] in pe = sq in qe, and
sp[0] in qe = sp[1] in qe = sp in pe.

Process pe can be defined as follows. (Process qe can be defined in the same way except that each occurrence of R_p in pe is replaced by an

occurrence of R_q in qe, each occurrence of B_q in pe is replaced by an occurrence of B_p in qe, each occurrence of sp in pe is replaced by an occurrence of sq in qe, and each occurrence of sq[0] or sq[1] in pe is replaced by an occurrence of sp[0] or sp[1], respectively, in qe.)

```
process pe
const  Rp  : integer      {private key of pe}
       Bq  : integer      {public key of qe}
       te  : integer      {time between secret exchanges}
       tr  : integer      {upper bound on round trip time}
var    sp  : integer
       sq  : array [0 .. 1] of integer{initially sq[0] = sq[1] = sq in
qe}
       d, e : integer
begin
    timeout  (sq[0] = sq[1] ∧
                 (te hours passed since rqst message sent last)) →
          sq[1] := NEWSCR;
          e := NCR(Bq, (sq[0]; sq[1]));
          send rqst(e) to qe

[]  rcv rqst(e) from qe →
          (d, e) := DCR(Rp, e);
          if sp = d ∨ sp = e →
                sp := e;
                e := NCR(Bq, sp);
                send rply(e) to qe
          [] sp ≠ d ∧ sp ≠ e →
                {detect adversary}
                skip
          fi

[]  rcv rply(e) from qe →
          d := DCR(Rp, e);
          if sq[1] = d →
                sq[0] := sq[1]
          [] sq[1] ≠ d →
                {detect adversary}
                skip
          fi

[] timeout  (sq[0] ≠ sq[1] ∧
```

```
            (tr seconds passed since rqst message sent last)) →
        e := NCR(Bq, (sq[0]; sq[1]));
        send rqst(e) to qe
end
```

The four actions of process pe use three functions NEWSCR, NCR, and DCR defined as follows. Function NEWSCR takes no arguments, and when invoked, it returns a fresh secret that is different from any secret that was returned in the past. Function NCR is an encryption function that takes two arguments, a key and a data item, and returns the encryption of the data item using the key. For example, execution of the statement

$$e := NCR(B_q, (sq[0]; sq[1]))$$

causes the concatenation of sq[0] and sq[1] to be encrypted using the public key Bq, and the result to be stored in variable e. Function DCR is a decryption function that takes two arguments, a key and an encrypted data item, and returns the decryption of the data item using the key. For example, execution of the statement

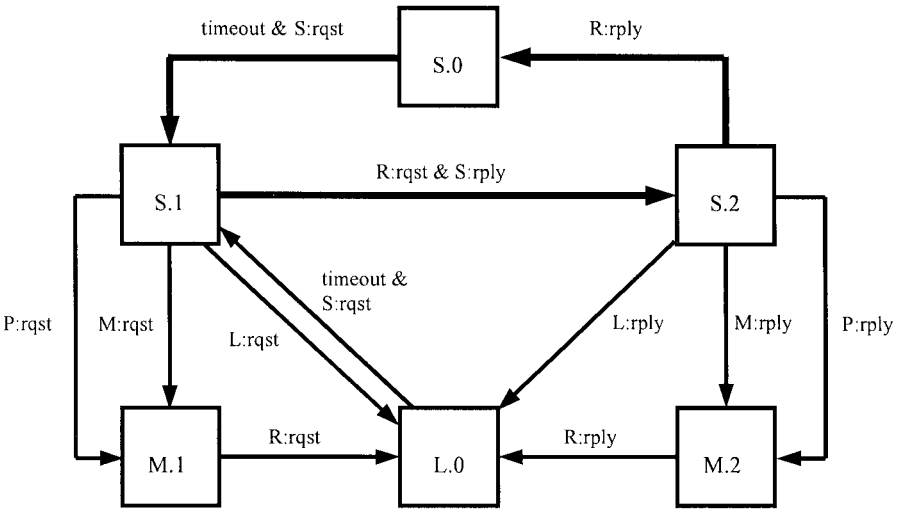
$$d := DCR(R_p, e)$$

causes the (encrypted) data item e to be decrypted using the private key Rp, and the result to be stored in variable d. As another example, consider the statement

$$(d, e) := DCR(R_p, e)$$

This statement indicates that the value of e is the encryption of the concatenation of two values (v_0; v_1) using key R_p. Thus, executing this statement causes e to be decrypted using key R_p, and the resulting first value v_0 to be stored in variable d, and the resulting second value v_1 to be stored in variable e.

To verify the correctness of the secret exchange protocol, we can use the state transition diagram of this protocol in Figure 6.2. This diagram has six nodes that represent all possible reachable states of the protocol. Every transition in the diagram stands for either a legitimate action (of process pe or process qe), or an illegitimate action of the adversary.



S.0 = ch.pe.qe = < > ∧ ch.qe.pe = < > ∧ sq[0] in pe = sq[1] in pe = sq in qe

S.1 = ch.pe.qe = <rqst(e)> ∧ ch.qe.pe = < > ∧ e = NCR(B_q, (sq[0]; sq[1])) ∧
sq[0] in pe ≠ sq[1] in pe ∧ sq[0] in pe = sq in qe

S.2 = ch.pe.qe = < > ∧ ch.qe.pe = <rply(e)> ∧ e = NCR(B_p, sq) ∧
sq[0] in pe ≠ sq[1] in pe ∧ sq[1] in pe = sq in qe

M.1 = ch.pe.qe = <rqst(e)> ∧ ch.qe.pe = < > ∧ e ≠ NCR(B_q, (sq[0]; sq[1])) ∧
sq[0] in pe ≠ sq[1] in pe ∧ (sq[0] in pe = sq in qe ∨ sq[1] in pe = sq in qe)

M.2 = ch.pe.qe = < > ∧ ch.qe.pe = <rply(e)> ∧ e ≠ NCR(B_p, sq) ∧
sq[0] in pe ≠ sq[1] in pe ∧ (sq[0] in pe = sq in qe ∨ sq[1] in pe = sq in qe)

L.0 = ch.pe.qe = < > ∧ ch.qe.pe = < > ∧
sq[0] in pe ≠ sq[1] in pe ∧ (sq[0] in pe = sq in qe ∨ sq[1] in pe = sq in qe)

Figure 6-2. State transition diagram of the secret exchange protocol.

Initially, the protocol starts at a state S.0, where the two channels between processes pe and qe are empty and the values of variables sq[0], sq[1] in pe and variable sq in qe are the same. This state can be defined by the following predicate

S.0 : ch.pe.qe = < > ∧ ch.qe.pe = < > ∧

sq[0] in pe = sq[1] in pe = sq in qe

At state S.0, exactly one action, namely the first timeout action in process pe, is enabled for execution. Executing this action at state S.0 leads the protocol to state S.1 defined as follows.

S.1 : ch.pe.qe = <rqst(e)> $\wedge$ ch.qe.pe = < > $\wedge$
e = NCR(B_q, (sq[0]; sq[1])) $\wedge$
sq[0] in pe $\neq$ sq[1] in pe $\wedge$ sq[0] in pe = sq in qe

At state S.1, exactly one legitimate action, namely the receive action (that receives a rqst message) in process qe, is enabled for execution. Executing this action at state S.1 leads the protocol to state S.2 defined as follows.

S.2 : ch.pe.qe = < > $\wedge$ ch.qe.pe = <rply(e)> $\wedge$
e = NCR(B_p, sq) $\wedge$
sq[0] in pe $\neq$ sq[1] in pe $\wedge$ sq[1] in pe = sq in qe

At state S.2, exactly one legitimate action, namely the receive action (that receives a rply message) in process pe, is enabled for execution. Executing this action at state S.2 leads the protocol back to state S.0 defined above.

States S.0, S.1 and S.2 are called good states because the transitions between these states consist of executing the legitimate actions of the two processes. The sequence of transitions from state S.0 to state S.1, to state S.2, and back to state S.0 constitutes the good cycle of the protocol. If only legitimate actions of processes pe and qe are executed, the protocol will stay in this good cycle indefinitely. Next, we discuss the bad effects caused by the actions of an adversary, and how the protocol can recover from these effects.

First, the adversary can execute a message loss action at state S.1 or S.2. If the adversary executes a message loss action at state S.1 or S.2, the network moves to a state L.0 defined as follows.

L.0 :ch.pe.qe = < > $\wedge$ ch.qe.pe = < > $\wedge$
sq[0] in pe $\neq$ sq[1] in pe $\wedge$
(sq[0] in pe = sq in qe $\vee$ sq[1] in pe = sq in qe)

At state L.0, only the second timeout action in pe is enabled for execution, and executing this action leads the network back to state S.1.

Second, the adversary can execute a message modification action at state S.1 or S.2. If the adversary executes a message modification action at state S.1, the network moves to state M.1 defined as follows.

M.1 : ch.pe.qe = <rqst(e)> $\wedge$ ch.qe.pe = < > $\wedge$
e $\neq$ NCR(B_q, (sq[0]; sq[1])) $\wedge$
sq[0] in pe $\neq$ sq[1] in pe $\wedge$

(sq[0] in pe = sq in qe $\vee$ sq[1] in pe = sq in qe)

If the adversary executes a message modification action at state S.2, the network moves to state M.2 defined as follows.

M.2 : ch.pe.qe = < > $\wedge$ ch.qe.pe = <rply(e)> $\wedge$
e $\neq$ NCR(B_p, sq) $\wedge$
sq[0] in pe $\neq$ sq[1] in pe $\wedge$
(sq[0] in pe = sq in qe $\vee$ sq[1] in pe = sq in qe)

In either case, the protocol moves next to state L.0 and eventually returns to state S.1.

Third, the adversary can execute a message replay action at state S.1 or S.2. If the adversary executes a message replay action at state S.1, the network moves to state M.1. If the adversary executes a message replay action at state S.2, the network moves to state M.2. As shown above, the protocol eventually returns to state S.1.

From the state transition diagram in Figure 6.2, it is clear that each illegitimate action by the adversary will eventually lead the network back to state S.1, which is a good state. Once the network is in a good state, the network can progress in the good cycle. Hence the following two theorems about secret exchange protocol are proved.

Theorem 6.1 *In the absence of an adversary, a network that executes the secret exchange protocol will follow the good cycle, consisting of the transitions from state S.0 to state S.1, from state S.1 to state S.2, and from state S.2 to state S.0, and will stay in this good cycle indefinitely.*

Theorem 6.2 *In the presence of an adversary, a network that executes the secret exchange protocol will converge to the good cycle in a finite number of steps after the adversary finishes executing the message loss, message modification, and message replay actions.*

2. WEAK INTEGRITY CHECK PROTOCOL

The main idea of the weak integrity check protocol is simple. Consider the case where a data(t) message, with t being the message text, is generated at a source src then transmitted through a sequence of adjacent routers r.1, r.2, ..., r.n to a destination dst. When data(t) reaches the first router r.1, r.1 computes a digest d for the message as follows:

d := MD(t; scr)

where MD is the message digest function, (t; scr) is the concatenation of the message text t and the shared secret scr between r.1 and r.2 (provided by the secret exchange protocol in r.1). Then, r.1 adds d to the message before transmitting the resulting data(t, d) message to router r.2.

When the second router r.2 receives the data(t, d) message, r.2 computes the message digest using the secret shared between r.1 and r.2 (provided by the secret exchange process in r.2), and checks whether the result equals d. If they are unequal, then r.2 concludes that the received message has been modified, discards it, and reports an adversary. If they are equal, then r.2 concludes that the received message has not been modified and proceeds to prepare the message for transmission to the next router r.3. Preparing the message for transmission to r.3 consists of computing d using the shared secret between r.2 and r.3 and storing the result in field d of the data(t, d) message.

When the last router r.n receives the data(t, d) message, it computes the message digest using the shared secret between r.(n-1) and r.n and checks whether the result equals d. If they are unequal, r.n discards the message and reports an adversary. Otherwise, r.n sends the data(t) message to its destination dst.

Note that this protocol detects and discards every modified message. More importantly, it also determines the location where each message modification has occurred.

Process pw in the weak integrity check protocol has two constants sp and sq that pw reads but never updates. These two constants in process pw are also variables in process pe, and pe updates them periodically, as discussed in the previous section. Process pw can be defined as follows. (Process qw is defined in the same way except that each occurrence of p, q, pw, qw, sp, and sq is replaced by an occurrence of q, p, qw, pw, sq, and sp, respectively.)

```
process pw
const  sp    :  integer
       sq    :  array [0 .. 1] of integer
var    t, d  :  integer
begin
    rcv data(t, d) from qw →
        if MD(t; sq[0]) = d  ∨  MD(t; sq[1]) = d →
            {accept message}
            RTMSG
        [] MD(t; sq[0]) ≠ d  ∧  MD(t; sq[1]) ≠ d →
            {report an adversary}
            skip
        fi
```

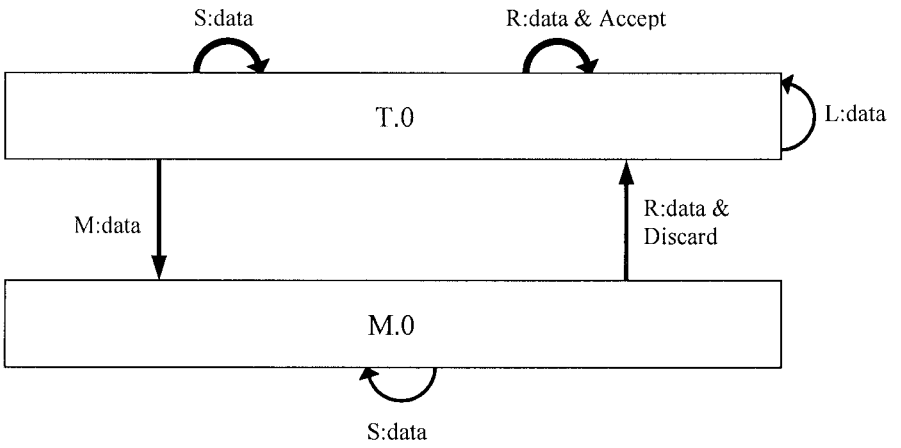
```
[]  true →
        {p receives data(t, d) from router other than q}
        {and checks that its message digest is correct}
        RTMSG

[]  true →
        {either p receives data(t) from an adjacent host or}
        {p generates the text t for the next data message}
        RTMSG
end
```

In the first action of process pw, if pw receives a data(t, d) message from qw while sq[0] ≠ sq[1], then pw cannot determine beforehand whether qw computed d using sq[0] or using sq[1]. In this case, pw needs to compute two message digests using both sq[0] and sq[1] respectively, and compare the two digests with d. If either digest equals d, then pw accepts the message. Otherwise, pw discards the message and reports the detection of an adversary.

The three actions of process pw use two functions named MD and NXT, and one statement named RTMSG. Function MD takes one argument, namely the concatenation of the text of a message and the appropriate secret, and computes a digest for that argument. Function NXT takes one argument, namely the text of a message (which we assume includes the message header), and computes the next router to which the message should be forwarded. Statement RTMSG is defined as follows.

```
if NXT(t) = p →
    {accept message}
    skip
[] NXT(t) = q →
    d := MD(t; sp);
    send data(t, d) to qw
[] NXT(t) ≠ p  ∧  NXT(t) ≠ q →
    {compute d as the message digest of}
    {the concatenation of t and the secret}
    {for sending data to NXT(t); forward}
    {data(t, d) to router NXT(t)}
    skip
fi
```

To verify the correctness of the weak integrity protocol, we can use the state transition diagram of this protocol in Figure 6.3, which considers the channel from process qw to process pw. (The channel from pw to qw, and the channels from pw to any other weak integrity process in an adjacent router of p, can be verified in the same way.) This diagram has two nodes that represent all possible reachable states of the protocol. Every transition in the diagram stands for either a legitimate action (of process pw or process qw), or an illegitimate action of the adversary.



$T.0 = I \wedge (\forall data(t, d)\ message\ in\ ch.qw.pw,\ d = MD(t; sq))$

$M.0 = I \wedge (\forall data(t, d)\ message\ in\ ch.qw.pw,$
$(\neg Head(data(t, d)) \Rightarrow d = MD(t; sq)) \wedge$
$(\ Head(data(t, d)) \Rightarrow d \neq MD(t; sq)))$

where

$I = sq\ in\ qw = sq[0]\ in\ pw \vee sq\ in\ qw = sq[1]\ in\ pw$

Figure 6-3. State transition diagram of the weak integrity check protocol.

Note that because the weak integrity check protocol operates below the secret exchange protocol in the protocol stack, we can assert that (sq in qw = sq[0] in pw $\vee$ sq in qw = sq[1] in pw) is an invariant in every state of the weak integrity protocol. We denote this invariant as I in the specification in Figure 6.3. Also note that the notation Head(data(t, d)) in the specification in

Figure 6.3 is a predicate whose value is true iff data(t, d) is the head message of the specified channel.

Initially, the protocol starts at state T.0. At state T.0, two legitimate actions, namely the send action in qw that sends a data message, and the receive action in pw that receives a data message, can be executed. Executing either one of the two actions at state T.0 keeps the protocol in state T.0.

States T.0 is the only good state in the weak integrity protocol. The sequence of the transitions from state T.0 to state T.0 constitutes the good cycle of the protocol. If only legitimate actions of processes pw and qw are executed, the protocol will stay in this good cycle indefinitely. Next, we discuss the bad effects caused by the actions of an adversary, and how the protocol can recover from these effects.

First, the adversary can execute a message loss action at state T.0. If the adversary executes a message loss action at state T.0, the predicate that for every data message data(t, d) in the channel from qw to pw, d = MD(t; sq), still holds. Therefore, the protocol stays at state T.0.

Second, the adversary can execute a message modification action at state T.0. If the adversary executes a message modification at state T.0, the protocol moves to state M.0. The receive and discard action executed by pw at state M.0 leads the protocol back to state T.0.

From the state transition diagram, it is clear that each illegitimate action by the adversary will eventually lead the protocol back to T.0, which is a good state. Once the protocol is in a good state, the protocol can progress in the good cycle. Hence the following two theorems about the weak integrity check protocol are proved.

Theorem 6.3 *In the absence of an adversary, a network that executes the weak integrity check protocol follows the good cycle, consisting of the single transition from state T.0 to state T.0, and will stay in this good cycle indefinitely.*

Theorem 6.4 *In the presence of an adversary, a network that executes the weak integrity check protocol will converge to the good cycle in a finite number of steps after the adversary finishes executing the message loss and message modification actions.*

However, the weak integrity check protocol, while being able to detect and discard all modified messages, cannot detect some replayed messages. In the next chapter, we introduce the strong integrity protocol that is capable of detecting and discarding all modified and replayed messages.

Chapter 7

STRONG HOP INTEGRITY USING SOFT SEQUENCE NUMBERS

The weak hop integrity protocol presented in the previous chapter can detect message modification but not message replay. In this chapter, we discuss how to strengthen this protocol to make it detect message replay as well. We present the strong hop integrity protocol in two steps. First, we present a protocol that uses "soft sequence numbers" to detect and discard replayed data messages. Second, we show how to integrate this soft sequence number protocol into the weak integrity check protocol presented in the previous chapter to construct the strong integrity check protocol. The combination of the secret exchange protocol and the strong integrity check protocol is the strong hop integrity protocol.

1. SOFT SEQUENCE NUMBER PROTOCOL

Before we introduce the soft sequence number protocol, we use a simple protocol to illustrate the need for sequence numbers in detecting message replay. Consider a protocol that consists of two processes u and v. Process u continuously sends data messages to process v. Assume that there is an adversary that attempts to disrupt the communication between u and v by inserting (i.e. replaying) old messages in the message stream from u to v. In order to overcome this adversary, process u attaches an integer sequence number s to every data message sent to process v. To keep track of the sequence numbers, process u maintains a variable nxt that stores the sequence number of the next data message to be sent by u and process v maintains a variable exp that stores the sequence number of the expected data message to be received by v. We call this protocol "hard sequence number protocol", because process u always remembers the next sequence

number to be sent, and process v always remembers the next sequence number it expects to receive.

To send the next data(s) message, process u assigns s the current value of variable nxt, then increments nxt by one. When process v receives a data(s) message, v compares its variable exp with s. If exp ≤ s, then v accepts the received data(s) message and assigns exp the value s + 1; otherwise v discards the data(s) message. Processes u and v of this protocol can be specified as follows.

```
process u
var nxt : integer    {sequence number of next sent message}
begin
    true →
        send data(nxt) to v;
        nxt := nxt + 1
end

process v
var s   : integer {sequence number of received message}
    exp : integer {sequence number expected next}
begin
    rcv data(s) from u →
        if s < exp →
            {reject message; report an adversary}
            skip
        [] exp ≤ s →
            {accept message}
            exp := s + 1
        fi
end
```

Correctness of this protocol is based on the observation that the predicate exp ≤ nxt holds at each (reachable) state of the protocol. However, if due to some fault (for example an accidental resetting of the values of variable nxt) the value of exp becomes larger than value of nxt, then all the data messages that u sends from this point, and until the value of nxt becomes equal to the value of exp, will be wrongly discarded by v. Next, we describe how to modify this protocol such that the number of messages, that can be wrongly discarded when the synchronization between u and v is lost due to some fault, is at most N, for some chosen integer N that is larger than one.

The modification consists of adding to process v two variables c and cmax, whose values are in the range 0..N-1. When process v receives a

data(s) message, v compares the values of c and cmax. If c ≠ cmax, then process v increments c by one (mod N) and proceeds as before, namely either accepts the data(s) message if exp ≤ s, or discards the message if exp > s. Otherwise, if c = cmax, then v accepts the message, assigns c the value 0, and assigns cmax a random integer in the range 0..N-1. We call this modified protocol "soft sequence number protocol" because process v at some instants "forgets" the sequence number it expects to receive next, and accepts the next received sequence number without question.

There are two considerations behind this modification. First, it guarantees that process v never discards more than N data messages when the synchronization between u and v is lost due to some fault. Second, it ensures that the adversary cannot predict the instants when process v is willing to accept any received data message, and so cannot exploit any such predictions by sending replayed data messages at the predicted instants.

Formally, processes u and v in this protocol can be defined as follows.

```
process u
var   nxt      : integer    {sequence number of next sent message}
begin
    true →
        send data(nxt) to v;
        nxt := nxt + 1
end

process v
const N        : integer
var   s        : integer {sequence number of received message}
      exp      : integer {sequence number expected next}
      c, cmax  : 0 .. N-1
begin
    rcv data(s) from u →
        if s < exp ∧ c ≠ cmax →
            {reject message; report an adversary}
            c := (c + 1) mod N
        [] exp ≤ s ∨ c = cmax →
            {accept message}
            exp := s + 1;
            if c ≠ cmax →
                c := (c + 1) mod N
            [] c = cmax →
                c := 0;
                cmax := RANDOM(0, N-1)
```

```
            fi
        fi
end
```

2. STRONG INTEGRITY CHECK PROTOCOL

Processes u and v of the soft sequence number protocol presented in Section 7.1 can be combined with process pw of the weak integrity check protocol to construct process ps of the strong integrity check protocol. A main difference between processes pw and ps is that pw exchanges messages of the form data(t, d), whereas ps exchanges messages of the form data(s, t, d), where s is the message sequence number computed according to the soft sequence number protocol, t is the message text, and d is the message digest computed over the concatenation (s; t; scr) of s, t, and the shared secret scr. Process ps in the strong integrity check protocol can be defined as follows. (Process qs can be defined in the same way.)

```
process ps
const  sp      : integer
       sq      : array [0 .. 1] of integer
       N       : integer
var    s, t, d: integer
       exp, nxt: integer
       c, cmax : 0 .. N-1
begin
    rcv data(s, t, d) from qs →
        if MD(s; t; sq[0]) = d ∨ MD(s; t; sq[1]) = d →
            if  s < exp ∧ c ≠ cmax →
                    {reject message; report an adversary}
                    c := (c + 1) mod N
            [] exp ≤ s ∨ c = cmax →
                    {accept message}
                    exp := s + 1;
                    if c ≠ cmax →
                        c := (c + 1) mod N
                    [] c = cmax →
                        c := 0;
                        cmax := RANDOM(0, N-1)
                    fi
            fi
        [] MD(s; t; sq[0]) ≠ d ∧ MD(s; t; sq[1]) ≠ d →
```

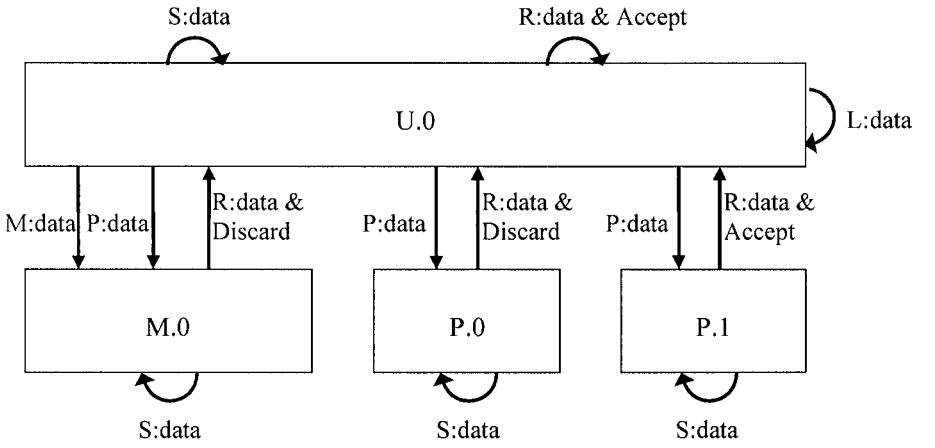
```
                {report an adversary} skip
            fi

[]  true →
            {p receives a data(s, t, d) from a router other than q and}
            {checks that its encryption is correct and}
            {its sequence number is within range}
            RTMSG

[]  true →
            {either p receives a data(t) from adjacent host or}
            {p generates the text t for the next data message}
            RTMSG
end
```

The first and second actions of process ps have a statement RTMSG that is defined as follows.

```
if NXT(t) = p →
        {accept message}
        skip
[] NXT(t) = q →
        d := MD(nxt; t; sp);
        send data(nxt, t, d) to qs;
        nxt := nxt + 1
[] NXT(t) ≠ p ∧ NXT(t) ≠ q →
        {compute next soft sequence number s}
        {for sending data to NXT(t); compute d}
        {as the message digest of the concatenation}
        {of s, t and the secret for sending data to}
        {NXT(t); forward data(s, t, d) to router NXT(t)}
        skip
fi
```

To verify the correctness of the strong integrity check protocol, we can use the state transition diagram of this protocol in Figure 7.1, which considers only the channel from process qs to process ps. (The channel from ps to qs, and the channels from ps to any other strong integrity check process in an adjacent router of p, can be verified in the same way.) This diagram has four nodes that represent all possible reachable states of the protocol. Every transition in the diagram stands for either a legitimate action (of process ps or process qs), or an illegitimate action of the adversary.

Note that because the strong integrity check protocol operates below the secret exchange protocol in the protocol stack, we can assert that (sq in qs = sq[0] in ps $\vee$ sq in qs = sq[1] in ps) is an invariant in every state of the strong integrity check protocol. We denote this invariant as I in the specification in Figure 7.1.



U.0 = I $\wedge$ ($\forall$ data(s, t, d) message in ch.qs.ps,
d = MD(s; t; sq) $\wedge$ (Head(data(s, t, d)) $\Rightarrow$ s $\geq$ exp in ps))

M.0 = I $\wedge$ ($\forall$ data(s, t, d) message in ch.qs.ps,
($\neg$Head(data(s, t, d)) $\Rightarrow$ d = MD(s; t; sq)) $\wedge$
(Head(data(s, t, d)) $\Rightarrow$ d $\neq$ MD(s; t; sq)))

P.0 = I $\wedge$ ($\forall$ data(s, t, d) message in ch.qs.ps,
d = MD(s; t; sq) $\wedge$
(Head(data(s, t, d)) $\Rightarrow$ s < exp in ps) $\wedge$ c $\neq$ cmax in ps)

P.1 = I $\wedge$ ($\forall$ data(s, t, d) message in ch.qs.ps,
d = MD(s; t; sq) $\wedge$
(Head(data(s, t, d)) $\Rightarrow$ s < exp in ps) $\wedge$ c = cmax in ps)

where

I = sq in qs = sq[0] in ps $\vee$ sq in qs = sq[1] in ps

Figure 7-1. State transition diagram of the strong integrity check protocol.

Initially, the protocol starts at state U.0. At state U.0, two legitimate actions, namely the send action in qs that sends a data message, and the receive action in ps that receives a data message, can be executed. Executing either one of the two actions at state U.0 keeps the protocol in state U.0.

State U.0 is the only good state in the strong integrity protocol. The set of transitions that leads the protocol from state U.0 to state U.0 constitutes the good cycle of the protocol. If only legitimate actions of processes ps and qs are executed, the protocol will stay in this good cycle indefinitely. Next, we discuss the bad effects caused by the actions of an adversary, and how the protocol can recover from these effects.

First, the adversary can execute a message loss action at states U.0. If the adversary executes a message loss action at state U.0, the predicate that for every data message data(s, t, d) in the channel from qs to ps, d = MD(s; t; sq), still holds. Therefore, the protocol stays at state U.0.

Second, the adversary can execute a message modification action at state U.0 causing the protocol to move to state M.0. The receive and discard action executed by ps at state M.0 leads the protocol back to state U.0.

Third, the adversary can execute a message replay action at state U.0. There are two cases to consider. First, if the replayed message data(s, t, d) is too old such that the secret used to compute the message digest is different from the current value of constant sq in process qs, then the protocol moves to state M.0, and later returns to state U.0 as discussed above. Second, if the replayed message data(s, t, d) is recent such that the secret used to compute the message digest is equal to the current value of constant sq in process qw, then the protocol moves either to state P.0 or to state P.1. With a high probability of (cmax – 1) / cmax, the protocol moves to state P.0, and the replayed message will be received and discarded by ps because the value of field s in the message indicates that the message is replayed. With a probability of 1 / cmax, the protocol moves to state P.1, and the replayed message will be received and accepted. In both cases the protocol returns to state U.0.

From the state transition diagram, it is clear that each illegitimate action by the adversary will eventually lead the protocol back to U.0, which is a good state. Once the protocol is in a good state, the protocol can progress in the good cycle. Moreover, if the adversary replays a recent data message, the replayed message will be detected and discarded with high probability (cmax – 1) / cmax. Hence the following two theorems about the strong integrity check protocol are proved.

Theorem 7.1 *In the absence of an adversary, a network that executes the strong integrity check protocol follows the good cycle, consisting of a single*

transition from state U.0 to state U.0, and will stay in this good cycle indefinitely.

Theorem 7.2 *In the presence of an adversary, a network that executes the strong integrity check protocol will converge to the good cycle in a finite number of steps after the adversary finishes executing any number of message loss or message modification actions. This network will also converge to the good cycle in a finite number of steps with a high probability of (cmax – 1) / cmax after the adversary finishes executing any number of message replay actions.*

The protocols used by the weak hop integrity protocol and the strong hop integrity protocol have several novel features that make them correct and efficient. First, whenever the secret exchange protocol attempts to change a secret, it keeps both the old secret and the new secret until it is certain that the integrity check of any future message will not be computed using the old secret. Second, the integrity check protocol computes a digest at every router along the message route so that the location of any occurrence of message modification can be determined. Third, the soft sequence number protocol makes the strong hop integrity protocol tolerate any loss of synchronization between any two adjacent routers.

Chapter 8

STRONG HOP INTEGRITY USING HARD SEQUENCE NUMBERS

Recall that in the strong hop integrity protocol presented in Chapter 7, we use soft sequence numbers to achieve strong hop integrity. We call those sequence numbers "soft" because at some random instants the receiving process forgets the last kept sequence number and accepts the next sequence number received from the sending process. There are two considerations behind the designing of soft sequence numbers. First, we want to limit the number of discarded messages during the period when the sending process and the receiving process lose synchronization of their sequence numbers. Therefore, we make the receiving process accept the newest sequence number from the sending process once in a while, so that the two processes can regain their synchronization shortly after their synchronization was lost. Second, we do not want to make it easy for an adversary to guess the instants at which the receiving process will accept any sequence number received next. If the adversary can guess the instant of acceptance correctly, it can make the receiving process accept a replayed sequence number at this instant and replay more messages afterward. Therefore, we randomize the instant of acceptance.

Soft sequence numbers help achieve strong hop integrity almost flawlessly. The only flaw, however, is that there is still a slight chance that an adversary might correctly guess the instant of acceptance even though the instant is randomized. An alternative to avoid this slight possibility is to use hard sequence numbers to achieve strong hop integrity, such that the two processes stick to their sequence numbers and the adversary has no chance to try its luck. However, one problem with hard sequence numbers is that the sending process and the receiving process will lose synchronization of their sequence numbers when a reset occurs. In one case as stated in Section 7.1,

the receiving process may discard a lot of fresh data messages after the sending process wakes up from a reset. In another case as will be shown in Section 8.1, the receiving process may accept replayed data messages after the receiving process wakes up from a reset. We do not want any one of the two bad cases to occur, and therefore we need to first solve the problems due to resets before hard sequence numbers can be exploited.

In this chapter, we propose a reset-tolerant version of hard sequence numbers, such that hard sequence numbers can be used as a substitute of soft sequence numbers to achieve strong hop integrity. This chapter is organized as follows. In Section 8.1, we review the hard sequence number protocol, which was first presented in Section 7.1, and elaborate the problems with this protocol in presence of resets. Then in Section 8.2, we discuss how the two operations, "SAVE" and "FETCH", can be added to make the hard sequence number protocol tolerate resets, and formally specify the new protocol. We show in Section 8.3 that the new protocol can converge to the resynchronization of the two processes after a reset occurred, and we show in Section 8.4 how "SAVE" and "FETCH" can be applied in the strong hop integrity protocol as an alternative of soft sequence numbers. Finally, we discuss tradeoffs between soft sequence numbers and hard sequence numbers in Section 8.5.

1. HARD SEQUENCE NUMBER PROTOCOL

In Section 7.1, we presented a hard sequence number protocol. In that protocol which consists of a sending process u and a receiving process v, process u attaches an integer sequence number s to every data message sent to process v in order to overcome an adversary that replays old messages in the message stream from u to v. Process u maintains a variable nxt that stores the sequence number of the next data message to be sent, and process v maintains a variable exp that stores the sequence number of the next data message that v expects to receive. To send the next data(s) message, process u assigns s the current value of variable nxt, then increments nxt by one. When process v receives a data(s) message, v compares its variable exp with s. If exp $\leq$ s, then v accepts the received message and assigns exp the value s + 1; otherwise v discards the message. Processes u and v of this protocol are specified as follows.

```
process u
var nxt : integer    {sequence number of next sent message}
begin
    true →
```

```
            send data(nxt) to v;   nxt := nxt + 1
    end

    process v
    var s    :  integer {sequence number of received message}
        exp  :  integer {sequence number expected next}
    begin
        rcv data(s) from u →
            if s < exp →
                {reject message; report an adversary}
                skip
            [] exp ≤ s →
                {accept message}
                exp := s + 1
            fi
    end
```

The hard sequence number protocol presented above can be used to detect replayed messages as long as both u and v stay up and never get reset. If process v ever encounters a reset, then an unbounded number of replayed messages can be accepted by v after v wakes up from the reset. Moreover, if process u ever encounters a reset, then an unbounded number of fresh messages that are sent by u after u wakes up can be discarded by v. In the following three paragraphs, we explain how these two bad possibilities can occur.

First, consider the case where process v is reset and later wakes up. When v wakes up, v has lost the last value of its variable exp. Thus v resumes its operation with its variable exp set to 0, and any (positive) sequence number received next by v will be accepted by v. Suppose the last fresh sequence number received by v before the reset is x, which is unbounded. In this case, an adversary can replay in ascending order all the messages with sequence numbers in the range from 1 to x, and all these replayed messages will be unsuspectedly accepted by v.

Next, consider the case where process u is reset and later wakes up. When u wakes up, u has lost the last value of its variable nxt that it will use on the next message to be sent to v. Thus u resumes its operation with nxt set to 0, and the next fresh message u sends to v will be data(0), and the next fresh message u sends to v will be data(1), and so on. Suppose the current value of variable exp in v is y, which is unbounded. In this case, all fresh messages sent from u to v with sequence numbers less than y will be regarded as replayed messages and discarded by v.

Last, consider the case where both process u and process v are reset and later wake up. When u wakes up, u resumes its operation in the protocol with nxt set to 0. When v wakes up, v resumes its operation with exp set to 0. In this case, an adversary gets the chance to replay messages sent before u was reset. The adversary can disrupt the communication between u and v by replaying a message with sequence number z that is larger than the current value of variable nxt in u, so that v is forced to set its variable exp to z. As a result, all fresh messages sent from u to v with sequence numbers in the range between nxt and z will be regarded as replayed messages and will be discarded by v.

In the next section, we propose two operations, "SAVE" and "FETCH", that can be added to the hard sequence number protocol to help the two processes regain synchronization of their sequence numbers after a reset occurred to one or both of them.

2. A PROTOCOL WITH SAVE AND FETCH OPERATIONS

As we have shown, the hard sequence number protocol is susceptible to reset because computer u (or v) forgets the last sent (or received) sequence number after a reset occurs to it. Therefore, we propose two operations, "SAVE" and "FETCH", that can be used to somewhat "reserve" the sequence number and thus can protect the communication between u and v from the impact of resets. The functions of SAVE and FETCH are straightforward. When the SAVE operation is executed at a computer, the last sequence number in the memory of the computer is stored in the persistent memory (e.g. the hard disk) of the computer. It is realistic to assume that the content of the persistent memory of the computer will not be corrupted or erased by a reset of that computer. When the FETCH operation is executed at a computer, the last sequence number stored in the persistent memory is loaded from the persistent memory into the memory. (SAVE and FETCH can be implemented by write-to-file and read-from-file operations in an operating system.)

SAVE and FETCH can be used in designing a new hard sequence number protocol that can avoid the impact of resets. A computer that executes the hard sequence number protocol can regularly execute SAVE to store a copy of a recent sequence number in its persistent memory. If this computer is reset and wakes up shortly, then although the last sequence number kept in its memory has been forgotten, this computer can execute FETCH to reload the sequence number stored in its persistent memory into

its memory, such that this computer does not need to restart its sequence number from 0.

To make sure the new protocol is correct, however, two considerations need to be addressed before the reloaded sequence number can be used for the next sent (or received) message of the resumed traffic. Firstly, the execution of SAVE takes some time, during which the computer can still send (or receive) messages. Hence there can be a gap between the reloaded sequence number (which is the last stored sequence number) and the sequence number of the last message sent (or received) by this computer before the reset. If a computer that plays the sender uses the reloaded sequence number directly and the size of the gap between the reloaded sequence number and the last sent sequence number before the reset is n, then the first n sent messages will be regarded as replayed messages by the receiver and will be discarded. If a computer that plays the receiver uses the reloaded sequence number directly, then an adversary can replay old messages whose sequence numbers are in the gap between the reloaded sequence number and the last received sequence number. These replayed messages will be accepted by the receiver because their sequence numbers look fresh to the receiver. In order to avoid these bad possibilities, a leap number should be added to the reloaded sequence number to leap over the gap before it can be used. This leap number must be large enough to ensure that after adding it to the reloaded sequence number, the resulting new sequence number is larger than all previously used sequence numbers. We will discuss how large the leap number should be in the next section.

Secondly, another reset can occur to the same computer that just waked up and has not yet executed the first SAVE after the last reset. In this case, those sequence numbers that have been used before the second reset occurs will be reused (or can be replayed) after the computer wakes up from the second reset. To avoid this problem, the computer should first execute a SAVE after the leap number is added to the reloaded sequence number. If this computer plays the sender, it will wait for the SAVE to finish before it sends the next message. If this computer plays the receiver, it will temporarily keep the messages that are received before the SAVE finishes in a buffer. After the SAVE completes its execution, messages kept in the buffer will be either delivered or discarded based on their sequence numbers.

Moreover, we have to decide how frequently the SAVE operation should be executed. On one hand, we do not want to execute SAVE too frequently because this can generate too much overhead. On the other hand, we do not want to execute SAVE too infrequently so that the saved sequence number is not recent enough. Our choice of the interval between two consecutive SAVEs is the maximum number of messages that can be sent (or received) during a time period that is equal to the execution time of SAVE. For

example, on a Pentium III 730-MHz machine running Linux 2.4.18, a write-to-file operation takes 100μs and sending a 1000-byte message takes 4μs on average. In this case, we can set the interval between two consecutive SAVEs to be at least 25.

Note that we measure the interval between two consecutive SAVEs in terms of the number of messages, rather than in terms of time, because the rate of message generation may change over time. At some time, the rate of message generation can be very low. In this case, measuring the interval in terms of time leads to wasteful SAVEs because when the interval to the next SAVE expires, the sequence number has not advanced much since the last SAVE was executed. Note also that the amount of time taken by every execution of SAVE can be different according to the current load of CPU. Therefore, we pick a reasonable upper bound on the execution time of SAVE, and determine the maximum number of messages that can be sent (or received) during this amount of time.

Next, we present the new hard sequence number protocol augmented with SAVE and FETCH. The new process u has two new inputs K_u and T_u, and has two new variables lst and wait. Input K_u is the interval between the sequence numbers stored by two consecutive SAVE operations in process u. Input T_u is the time needed to execute a SAVE operation at u. Variable lst is the last sequence number stored by a SAVE operation, and variable wait is a boolean that is set to true only when process u is reset. The new process u can be specified as follows.

```
process u
inp  Ku, Tu  :  integer          {Ku > 0}
var  nxt     :  integer          {next to be sent, init. 0}
     lst     :  integer          {last stored, init. 0}
     wait    :  boolean          {initially false}
begin
     ~ wait  →
          send data(nxt) to v;
          nxt := nxt + 1;
          if  nxt ≥ Ku + lst  →
                lst := nxt;
                &SAVE(lst)  {execute SAVE in background}
          []  nxt < Ku + lst  →
                skip
          fi

[]   (process u is reset)  →
          wait := true
```

```
[]  (process u wakes up after a reset) →
        FETCH(nxt);
        SAVE(nxt + 2K_u);        {execute SAVE in foreground}
        nxt := nxt + 2K_u;
        lst := nxt;
        wait := false
end
```

In the first action of process u, when variable wait is false, u sends the next message data(nxt) to process v and increment nxt by 1. Then, u checks whether nxt has become K_u greater than the last stored sequence number lst. If so, u executes SAVE to store nxt into persistent memory. (This SAVE should be executed in the background so that it does not block the normal communication between u and v.) In the second action, when u is reset, variable wait is set to true. In the third action, when u wakes up after a reset, u executes FETCH(nxt) to reload the last stored sequence number into variable nxt, executes SAVE(nxt + $2K_u$) to store the result of adding the leap number to the reloaded sequence number, and sets nxt and lst to their new values after the SAVE operation has finished. Then, variable wait is set to false, so that the first action is enabled again and u can send the next message data(nxt) to v.

The new process v, that is augmented with SAVE and FETCH, has two new inputs K_v and T_v, and two new variables lst and wait. Input K_v is the interval between the sequence numbers stored by two consecutive SAVE operations in process v. Input T_v is the time needed to execute a SAVE operation at v. Variable lst is the last sequence number stored by a SAVE operation, and variable wait is a boolean that is set to true only when process v is reset. The new process v can be specified as follows.

```
process v
inp  K_v, T_v  :  integer        {Kv > 0}
var  exp       :  integer        {expected to receive, init. 0}
     lst       :  integer        {last stored, init. 0}
     s         :  integer
     wait      :  boolean        {initially false}
begin
     rcv data(s) from u →
          if ~ wait →
               if s < exp →
                    {reject message; report an adversary}
                    skip
```

```
                [] exp ≤ s →
                      {accept message} exp := s + 1
                fi;
                if exp ≥ K_v + lst →
                      lst := exp;
                      &SAVE(exp)    {execute SAVE in background}
                [] exp < K_v + lst →
                      skip
                fi
          [] wait →
                {discard message} skip
          fi

[]  (process v is reset) →
          wait := true

[]  (process v wakes up after a reset) →
          FETCH(exp);
          SAVE(exp + 2K_v);    {execute SAVE in foreground}
          exp := exp + 2K_v;
          lst := exp;
          wait := false
end
```

Process v has three actions. In the first action, v receives data(s) from u and checks whether variable wait is true. If v is not waiting, then v decides whether to discard or deliver the message according to the value of s and the value of exp, and then checks whether exp has become at least K_v greater than the last stored sequence number lst. If so, v executes SAVE(exp) in the background to store exp into persistent memory. If v is waiting, then v discards the message. In the second action, when v is reset, variable wait is set to true. In the third action, when v wakes up after a reset, v executes FETCH(exp) to reload the last stored sequence number, executes SAVE(exp + $2K_v$) to store the result of adding the leap number to the reloaded sequence number, and sets exp and lst to their new values after the SAVE operation has finished. Finally, variable wait is set to false.

3. CONVERGENCE OF NEW HARD SEQUENCE NUMBER PROTOCOL

We are now ready to show why the sending process u and the receiving process v in the hard sequence number protocol are guaranteed to converge to a fresh sequence number after a reset. Our objective is to show that after adding a leap number to the reloaded sequence number, the resulting new sequence number is larger than the last sequence number used before the reset occurs, hence no old sequence number can be reused to send fresh message and no replayed message will be accepted by the receiver. We analyze the aforementioned two cases: a reset occurs at the sending process u, and a reset occurs at the receiving process v. (From the analysis of the two cases it is straightforward to verify the third case when both process u and process v are reset.) After showing that the new sequence number used after the reset is guaranteed to be fresh, we will show that the following two conditions hold under the new protocol. First, when process u is reset, a bounded number of sequence numbers will be lost but no fresh message will be discarded by process v if no message reorder occurs. Second, when process v is reset, the number of discarded fresh messages is bounded.

We start with the analysis of the case in which a reset occurs at process u. Assume that the reset occurs while process u is executing a SAVE to store the sequence number nxt into persistent memory. From Figure 8.1, there are two possible cases to consider: the reset occurs before the current SAVE finishes, or the reset occurs after the current SAVE finishes.

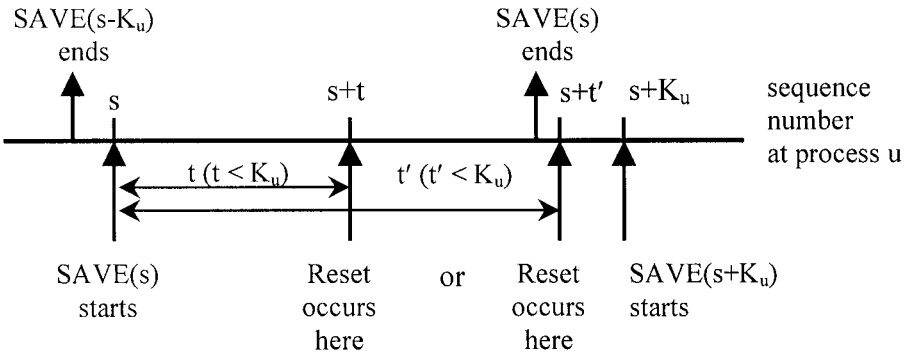


Figure 8-1. Analysis of reset occurring at process u.

To check the first case, suppose the reset occurs at sequence number s + t, where $t < K_u$ because the next sequence number to be stored will be $s + K_u$.

The sequence number fetched by u after it wakes up is $s - K_u$, as SAVE(s) has not completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(s + t) - (s - K_u) \leq (s + K_u) - (s - K_u) = 2K_u$$

To check the second case, suppose the reset occurs at $s + t'$, where $t' < K_u$. The sequence number fetched by u after it wakes up is s, as SAVE(s) has completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(s + t') - s \leq (s + K_u) - s = K_u$$

Therefore, if we add a leap number of $2K_u$ to the fetched sequence number, as specified in process u, the next sequence number used by u is guaranteed to be fresh.

Next, we analyze the case when a reset occurs at process v. Assume that a reset occurs while process v is executing a SAVE to store the sequence number r into persistent memory. From Figure 8.2, there are two possible cases to consider: the reset occurs before the current SAVE finishes, or the reset occurs after the current SAVE finishes.

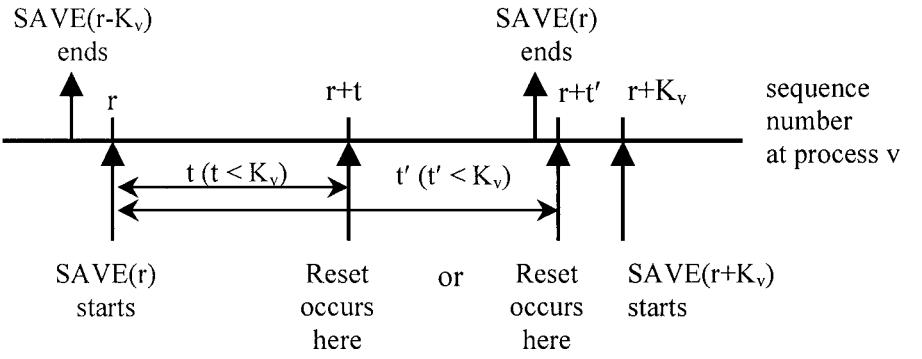


Figure 8-2. Analysis of reset occurring at process v.

To check the first case, suppose the reset occurs at sequence number r + t, where $t < K_v$ because the next sequence number to be stored will be $r + K_v$. The sequence number fetched by q after it wakes up is $r - K_v$, as SAVE(r) has not completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(r + t) - (r - K_v) \leq (r + K_v) - (r - K_v) = 2K_v$$

To check the second case, suppose the reset occurs at $r + t'$, where $t' < K_v$. The sequence number fetched by v after it wakes up is r, as SAVE(r) has completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(r + t') - r \leq (r + K_v) - r = K_v$$

Therefore, if we add a leap number of $2K_v$ to the fetched sequence number, as specified in process v, it is guaranteed that v will not accept any replayed message.

Next, we show that the hard sequence number protocol satisfies the following two properties.

I. *When the sender is reset, no more than $2K_u$ fresh sequence numbers are lost and no fresh messages are discarded by the receiver if no message reorder occurs.*

Note that process u may lose some fresh sequence numbers after a reset because u adds a leap number $2K_u$ to the reloaded sequence number. Suppose $s\text{-}K_u$ is the last stored sequence number when a reset occurs at u. Then when u wakes up, u resumes with sequence number $s+K_u$ because u first reloaded $s\text{-}K_u$ and added $2K_u$ to it. In this case, u loses no more than $2K_u$ fresh sequence numbers because u resumes with $s+K_u$ and all numbers between $s\text{-}K_u$ and $s+K_u$ become unusable. Therefore, the total number of lost sequence number is bounded by $2K_u$. Moreover, since $s+K_u$ is larger than all previously used sequence numbers, no fresh message will be discarded by the receiver unless any fresh message sent after the reset arrives earlier than any fresh message sent before the reset.

II. *When the receiver is reset, no more than $2K_v$ fresh messages are discarded by the receiver.*

Note that process v may discard some fresh messages after a reset because q adds a leap number $2K_v$ to the reloaded sequence number. Suppose $r\text{-}K_v$ is the last stored sequence number when a reset occurs at v. Then when v wakes up, v resumes with sequence number $r+K_v$ because v first reloaded $r\text{-}K_v$ and added $2K_v$ to it. The worst case that can occur is that $r\text{-}K_v+1$ has not been received by v when a reset occurs. In this case, v may discard at most $2K_v$ fresh messages if no message loss occurs, because v resumes with $r+K_v$, and all fresh messages with sequence numbers between $r\text{-}K_v$ and $r+K_v$ will be regarded as replayed messages by v. Therefore, the total number of discarded fresh messages is bounded by $2K_v$.

4. APPLICATION OF SAVE AND FETCH IN STRONG HOP INTEGRITY PROTOCOL

We have shown that the sending process u and the receiving process v in the hard sequence number protocol, with the help of SAVE and FETCH, are guaranteed to converge to a fresh sequence number after a reset. Next, we discuss how SAVE and FETCH can be integrated with the strong integrity check protocol, such that the protocol can recover from resets.

To integrate SAVE and FETCH with the strong integrity check protocol, the following four steps can be followed. First, in the first action of process ps in the protocol, we need to add statements to periodically execute SAVE, and add statements to put incoming messages in a buffer when ps is waiting for a SAVE that executes after a FETCH to finish. Second, in the RTMSG statement, we also need to add a statement to periodically execute SAVE. Third, we need to add an action to process ps to execute FETCH and SAVE when process ps wakes up from a reset. Fourth, we need to add a timeout action to set up the sequence number properly after a post-reset SAVE finishes its execution.

Similarly, the SAVE and FETCH operations can be used in the anti-replay window protocol in IPsec to make the protocol reset-tolerant [22], such that security associations (SA) that are affected by resets do not need to be deleted and reestablished as proposed in previous works [19, 30].

5. TRADEOFFS BETWEEN SOFT SEQUENCE NUMBERS AND HARD SEQUENCE NUMBERS

Although both soft sequence numbers and hard sequence number can be used to achieve strong hop integrity, the two approaches are different and each of them has its own advantages and disadvantages. In this section, we discuss the tradeoffs between soft sequence numbers and hard sequence numbers.

First, we discuss the implementation complexity of the two approaches. Soft sequence numbers are easier to implement because they do not require SAVE and FETCH operations and do not require persistent memory. By contrast, implementation of hard sequence numbers requires write and read operations, namely SAVE and FETCH, and a real-time timeout for executing SAVE after FETCH. Moreover, a good upper bound of the number of sent message and a good upper bound of the number of received message during the execution delay of SAVE are also needed by hard sequence numbers.

Second, we discuss the degrees of security of the two approaches. Soft sequence numbers can only provide high, but not complete, protection against message replay attacks. This is because there is a small chance that an adversary may correctly guess the point that the receiving process accepts next received sequence number anyway. By contrast, hard sequence numbers can provide complete protection against message replay attacks, because both the sending process and the receiving process stick to the sequence number they keep, and an adversary has no chance to try its luck.

Chapter 9

IMPLEMENTATION CONSIDERATIONS

We have introduced in Chapters 5 to 8 the three components of hop integrity protocol suite, namely the secure address resolution protocol, the weak hop integrity protocol, and two versions of the strong hop integrity protocol that use soft sequence numbers and hard sequence numbers respectively. We discussed their functions, specified each of the protocols in a formal fashion using a variation of Abstract Protocol Notation, and verified the correctness of each protocol using state transition diagrams. All the protocols are stateless, require small overhead, and do not constrain the network protocol in the routers in any way. Thus, we believe they are compatible with IP in the Internet.

In this chapter, we discuss implementation considerations of hop integrity protocols and acceptable values for the inputs of each of these protocols. In Section 9.1, we discuss several issues concerning the implementation of keys and secrets. In Section 9.2, we discuss acceptable lengths of timeout periods used in the secret exchange protocol. In Section 9.3, we discuss considerations about sequence numbers used in the strong integrity check protocol. Finally in Section 9.4 we discuss message overhead of the strong integrity check protocol.

1. KEYS AND SECRETS

In the secret exchange protocol presented in Chapter 6, we define keys with two inputs R_p and B_q. Input R_p is a private key for router p, and input B_q is a public key for router q. These are long-term keys that remain fixed for long periods of time (say one to three months), and can be changed only off-line and only by the system administrators of the two routers. Thus, these

keys should consist of a relatively large number of bytes, say 128 bytes (1024 bits) each. There are no special requirements for the encryption and decryption functions that use these keys in the secret exchange protocol.

In the integrity check protocols in Chapters 6 and 7, we define secrets with two inputs sp and sq and define the integrity check computation function as function MD. Inputs sp and sq are short-lived secrets that are updated every 4 hours. Thus, this key should consist of a relatively small number of bytes, say 8 bytes. Function MD is used to compute the digest of a data message. Function MD can compute in two steps as follows. First, the standard function MD5 [44] is used to compute a 16-byte digest of the data message. Second, the first 4 bytes from this digest constitute our computed message digest. (Computing a message digest over a 1024-byte message using MD5 is timed at just 0.037 ms on a Pentium III 730MHz machine running Linux. It is not a significant overhead to a router.)

2. TIMEOUTS

In the secret exchange protocol, we define two needed timeout values with two inputs te and tr. Input te is the time period between two successive secret exchanges between pe and qe. On one hand, this time period should be small so that an adversary does not have enough time to deduce the secrets sp and sq used in computing the integrity checks of data messages. On the other hand, it should be large so that the overhead that results from the secret exchanges is reduced. An acceptable value for te is around 4 hours.

Input tr is the timeout period for resending a rqst message when the last rqst message or the corresponding rply message was lost. The value of tr should be an upper bound on the round-trip delay between the two adjacent routers. If the two routers are connected by a high speed Ethernet, then an acceptable value of tr is around 4 seconds.

3. SEQUENCE NUMBERS

The sequence numbers in the strong integrity check protocol presented in Chapters 7 and 8 can be made recyclable. Note that the sequence numbers used in the strong integrity check protocol are specified as unbounded, for the simplicity of our presentation but without loss of generality. In practice, there is an upper bound on the sequence number, because we need to determine how many bits should be allocated to the sequence number field. However, two problems arise when bounded sequence numbers are used.

First, the sequence number wraps around when it reaches the upper bound. At the instant that the sequence number wraps around, a smaller sequence number looks fresher than a greater sequence number. Thus, if the shared secret sq between process ps and process qs remains the same during the whole round of the sequence number, then when ps receives from qs a message whose sequence number just wraps around, it cannot distinguish whether the message is fresh or replayed from the last round. Second, the received sequence numbers may not be consecutive because messages may get lost in their transit. Thus, when process ps receives from process qs a message whose sequence number is smaller than the value of exp, there are two possibilities that process ps has to distinguish: either the message is a fresh message whose sequence number just wraps around (all the messages between the last one and this one are lost in transit), or the message is indeed a replayed message.

The above two problems can be overcome if the following two requirements are satisfied. First, the shared secret is updated at least once during every round of the sequence number, such that an adversary cannot take any message in the last round and replay it in the current round. Second, the upper bound of the sequence number is chosen large enough, such that no loss of consecutive messages can confuse process ps on its judgment of a message's freshness. Assume that we choose S as the upper bound of sequence number, and te as the time period between two successive secret exchanges. Also assume that the maximum rate that messages can be transmitted in the network is R, and the maximum number of consecutive messages that can get lost in their transit over the network is L. Then the above two requirements can be translated into the following two formulas:

$$S \geq te * R \qquad (1), \text{ and}$$

$$S \geq 2L \qquad (2).$$

In a usual Ethernet, at most 800 messages can be sent in a second. Therefore, in the period of 4 hours, which is the value we choose for timeout period te, at most 11,520,000 messages can be sent. Using 4 bytes to store the sequence numbers is a proper choice with considerations of covering the maximum number of consumed sequence numbers in timeout period te and aligning with the original IP header.

As discussed in Chapter 7, input N, which is the upper bound for random integer cmax, needs to be much larger than 1. For example, if we choose N to be 200, then the maximum number of messages that can be discarded wrongly whenever synchronization between two adjacent routers is lost is

200, and the probability that an adversary who replays an old message will be detected is 99 percent.

4. MESSAGE OVERHEAD

The message overhead of the strong integrity check protocol is about 8 bytes per data message: 4 bytes for storing the message digest, and 4 bytes for storing the sequence number of the message.

We propose to add the message digest and sequence number used by the strong integrity check protocol to the IP options in the IP header of each message. IP options are auxiliary fields used mainly for network control or testing purposes. They are added at the tail of the standard 20-byte IP header, and the total length of all IP options in a message can be as much as 40 bytes because the maximum length of IP header is 60 bytes. Special options for the message digest and sequence number can be defined and inserted into the IP options field of each message.

Chapter 10

OTHER USES OF HOP INTEGRITY

The three protocols in the hop integrity protocol suite, namely secure address resolution protocol, weak hop integrity protocol, and strong hop integrity protocol, can be used to counter most denial-of-service attacks, because they satisfy the three conditions of hop integrity and therefore can discard most denial-of-service attack messages at their first hops. What is even better, though, is that according to our investigation, we discovered that hop integrity can be used to solve other network security problems besides preventing denial-of-service attacks.

In this chapter, we present four other applications of hop integrity. In Section 10.1, we present an application of hop integrity in the mechanism of mobile IP. In Section 10.2, we present an application of hop integrity in the mechanism of multicast. In Section 10.3, we discuss how to make routing protocols more secure. Finally in Section 10.4, we discuss an application of hop integrity to provide security for ad hoc networks and sensor networks. In each section, we first give an overview of the mechanism itself. Then, we discuss a network security problem that can disrupt this mechanism. Finally, we show how hop integrity can be used to solve the problem.

1. MOBILE IP

Mobile IP [41] is a mechanism designed to accommodate the communication need of mobile computers. According to IP version 4, the IP address of a computer uniquely identifies the computer's point of attachment to the Internet. Therefore, when a mobile computer visits a foreign subnetwork, this computer must change its IP address such that messages destined to this computer can be delivered to it. However, changing IP

address along with the change of location makes a computer lose its current transport and application layer connections, because higher-layer connections are dependent on the computer's IP address. Mobile IP provides the feature that a mobile computer can keep its IP address by registering a foreign agent in the foreign subnetwork with the home agent in its home subnetwork. The foreign agent registered by the mobile computer then works with the home agent to accomplish the delivery of IP messages destined to the mobile computer.

According to mobile IP, while a mobile computer c is visiting a foreign subnetwork F, IP messages destined to mobile computer c are routed indirectly, and IP messages generated at mobile computer c are routed directly. The indirect routing of an IP message destined to c proceeds in three steps. First, the IP message is routed toward the home subnetwork H of c and is intercepted by the home agent ha of c in its home subnetwork H. Second, ha forwards the message in a tunnel to the foreign agent fa of c in the foreign subnetwork F. Third, fa forwards the IP message over F to mobile computer c. This procedure of indirect routing is illustrated in Figure 10.1.

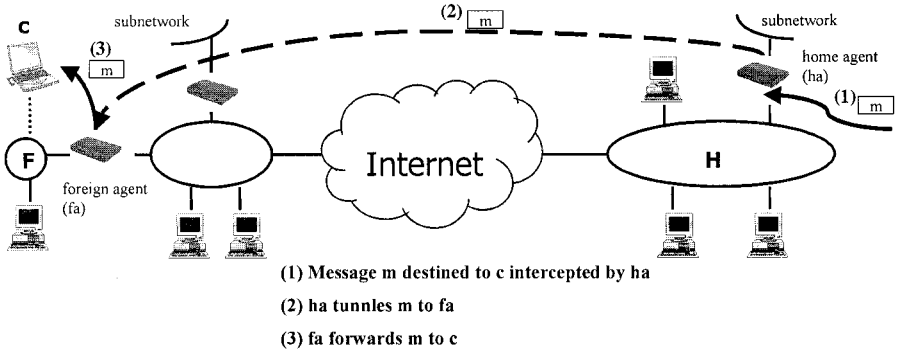


Figure 10-1. Indirect routing in mobile IP.

The direct routing of an IP message generated at the mobile computer proceeds in two steps. First, mobile computer c forwards the IP message over the foreign subnetwork F to the foreign agent fa. Second, fa and all subsequent routers forward the IP message towards its intended ultimate destination.

However, the aforementioned direct routing causes the following serious problem. When foreign agent fa forwards for mobile computer c a message m to next router q, q applies ingress filtering to m and discovers that the original source of m (namely mobile computer c) is not consistent with

where m came from (namely foreign agent fa). Thus, q ends up discarding m although m is a legitimate message. This problem is illustrated in Figure 10.2.

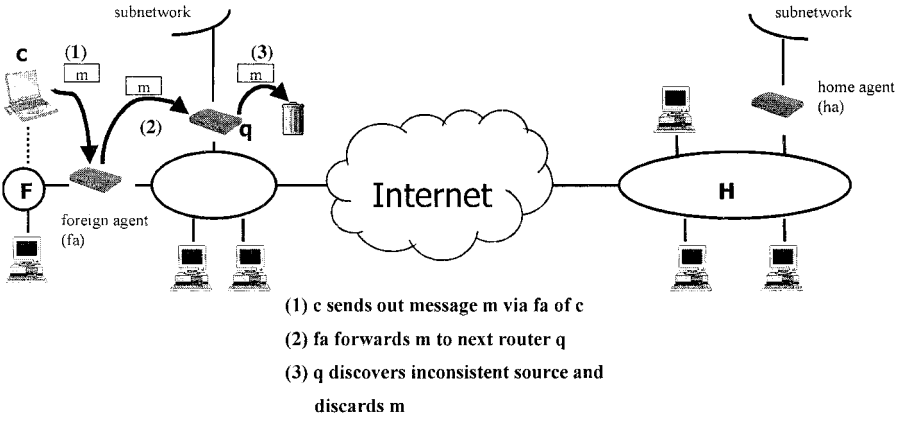


Figure 10-2. Problem with direct routing in mobile IP.

In RFC 3024 [35], reverse tunneling for mobile IP was proposed to solve this problem. This scheme, designed to be symmetric to indirect routing on purpose, requires that every IP message generated at mobile computer c is first forwarded in a tunnel to the home agent ha of the mobile computer, and then routed toward its ultimate destination by home agent ha. When ha forwards a message m generated at the mobile computer to next router q′, q′ applies ingress filtering to m and discovers that the original source of m (namely mobile computer c) is consistent with where m came from (namely home agent ha). Thus, q′ forwards m toward its ultimate destination as usual. However, one problem with reverse tunneling is that the cost of reverse tunneling is expensive. Every IP message generated at mobile computer c needs to unnecessarily travel all the way to home agent ha before it can be routed toward its ultimate destination, no matter where the ultimate destination is. Reverse tunneling can be illustrated as in Figure 10.3.

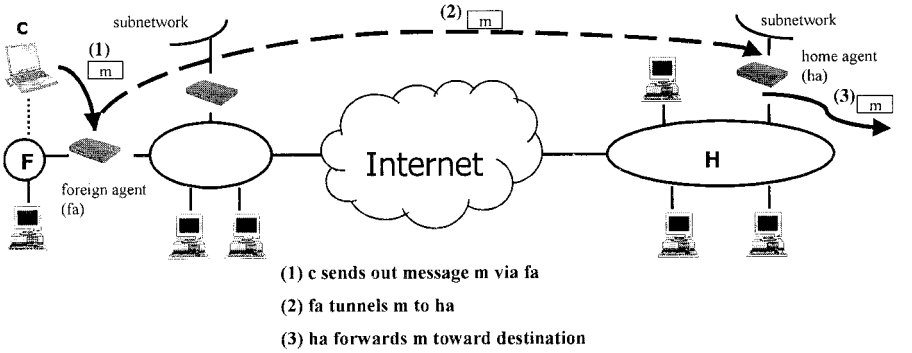


Figure 10-3. Reverse tunneling in mobile IP.

We find that if hop integrity is deployed in all the routers of the network, then we can still use direct routing to route IP messages generated at mobile computer c toward their ultimate destinations, thereby avoiding the expensive cost of reverse tunneling. Recall that the problem with direct routing is that when next router q receives a message m generated at mobile computer c from foreign agent fa, q cannot determine from source address of m whether m is forwarded by fa, or m carries forged source address. However, if hop integrity is deployed in all the routers of the network, then foreign agent fa will add an integrity check d to message m before it forwards m to next router q. When next router q receives message m from fa, q can correctly determine from the integrity check d contained in m whether m was indeed forwarded by fa. If d is consistent with m, q accepts m, computes a new integrity check d for the next router, and proceeds to forward it toward its ultimate destination. Otherwise, if d is not consistent with m, then q discards m. Thus the problem with direct routing is solved. This procedure is illustrated in Figure 10.4.

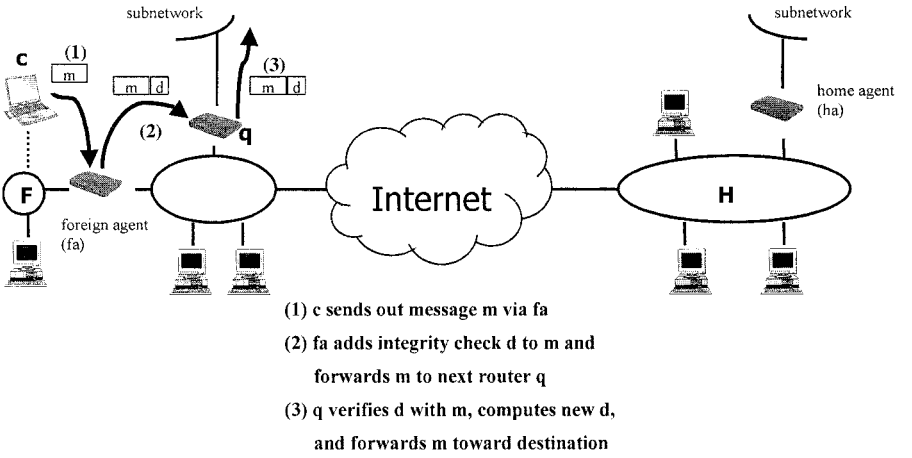


Figure 10-4. Direct routing in mobile IP with hop integrity.

2. SECURE MULTICAST

Multicast IP [10] is a mechanism designed to transmit an IP message to a set of zero or more hosts identified by a single IP destination address. Many multicast protocols have been proposed and widely deployed in the Internet to achieve multicast IP, for example the Distance Vector Multicast Routing Protocol (or DVMRP, for short) [53] and the Protocol Independent Multicast (PIM-DM for Dense Mode and PIM-SM for Sparse Mode) [11]. Multicast protocols are based on organizing the routers between the multicast source and the multicast destinations into a rooted spanning tree. When a router in the spanning tree receives a multicast IP message, it forwards a copy of the message to every multicast destination that is adjacent to it and to every router that is its "child" in the spanning tree. Figure 10.5 illustrated an example of a multicast spanning tree. In this example, router r.0 forwards a copy of message m to its two children in the spanning tree, namely r.1 and r.2, and router r.1 forwards a copy of m to its child r.3 in the spanning tree and to a multicast destination that is adjacent to it. Other routers in the spanning tree proceed in a similar way to forward a copy of m to every multicast destination in the spanning tree.

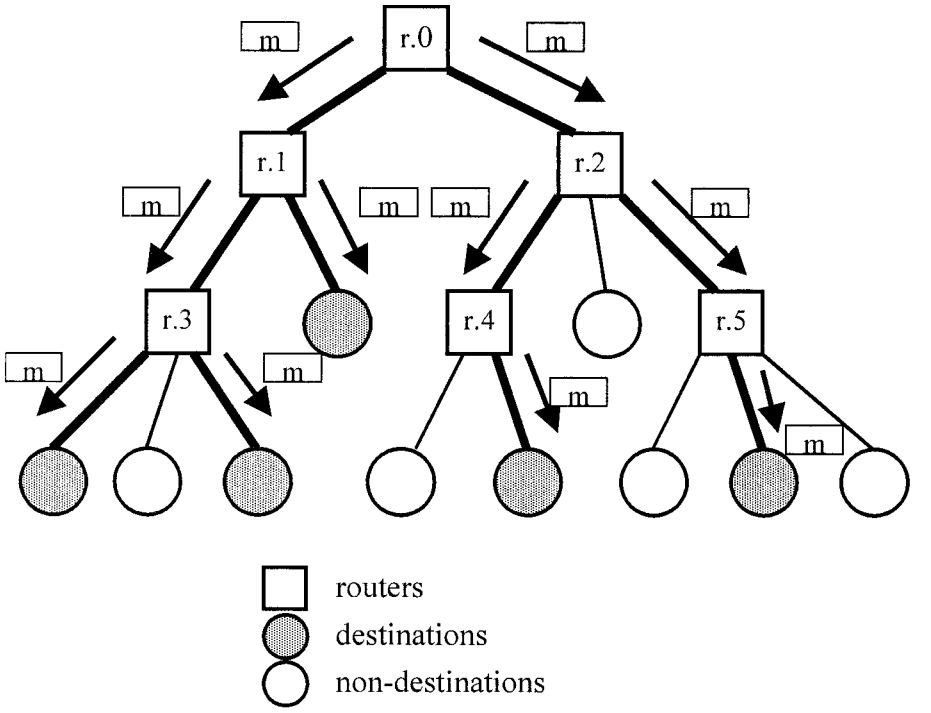


Figure 10-5. A multicast spanning tree.

Because IP messages can be lost while in transit, the multicast IP protocols do not guarantee that every multicast message generated at the multicast source is eventually received at every multicast destination. Instead, the multicast IP protocols guarantee the following weaker correctness criterion: if a multicast destination receives a multicast IP message, then each multicast destination receives the same message with high probability.

However, this weak correctness criterion can still be violated by a simple adversary as follows. If the adversary inserts a new multicast IP message between two routers in the middle of the spanning tree, or modifies a message while the message is being transmitted between two routers in the middle of the spanning tree, then only a small fraction of the multicast destinations eventually receive the inserted or modified message. In an example illustrated in Figure 10.6, an adversary sitting between router r.1 and router r.3 intercepts a message m forwarded by r.1 toward r.3, modifies m to become m′, and forwards m′ to r.3. Router r.3 accepts the modified message m′ unsuspectingly, and forwards a copy of m′ to the two multicast destinations that are adjacent to it. As a result, the above weak correctness

criterion is violated because only the two multicast destinations that are adjacent to r.3 eventually receives the modified message m′.

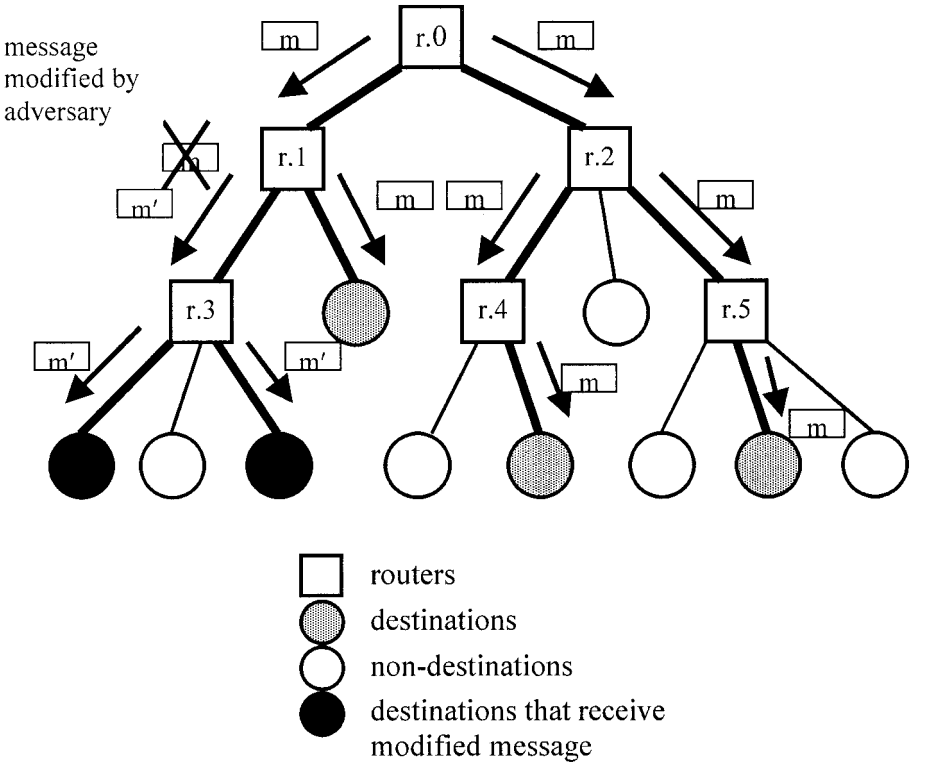


Figure 10-6. Correctness criterion of multicast is violated by an adversary.

We discover that hop integrity can be used to keep the above weak correctness criterion of multicast IP as follows. If hop integrity is deployed between each pair of adjacent routers in the spanning tree, then each pair of adjacent routers in the spanning tree share two unique secrets (one for each direction) that can be used to compute an integrity check for every message exchanged between this pair of routers. Therefore, every multicast IP message exchanged between any pair of adjacent routers in the spanning tree is protected by an integrity check added by the sending router. Because an adversary does not know the secret shared between two adjacent routers, this adversary cannot compute a correct integrity check for any multicast IP message it inserts or modifies no matter where it is located. As a result, the inserted or modified multicast IP messages will be detected and discarded by the first router that receives them. For example, as shown in Figure 10.7,

when router r.3 receives the modified message m′, r.3 detects that m′ is modified because its integrity check does not match m′. Thus r.3 discards m′ and will not forward m′ to any adjacent multicast destination.

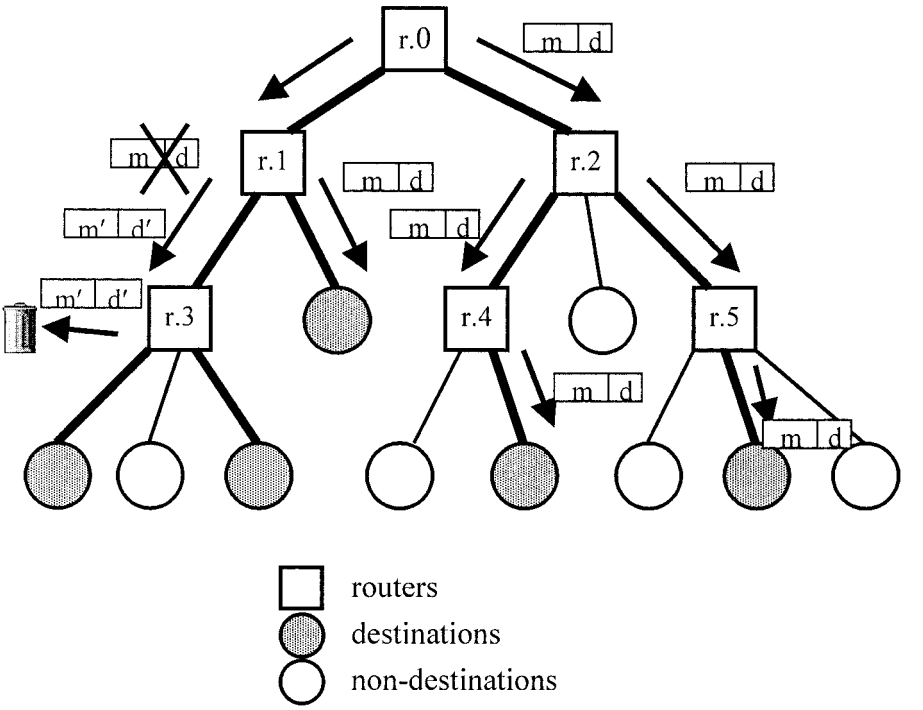


Figure 10-7. Hop integrity keeps correctness criterion of multicast.

Therefore, if hop integrity is deployed between each pair of adjacent routers in the spanning tree, then no multicast destination will receive the inserted or modified multicast IP messages, and the weak correctness criterion is maintained.

3. SECURITY OF ROUTING PROTOCOLS

Routers use routing protocols to compute the entire path or the next hop for forwarding a message toward its ultimate destination. Most widely used routing protocols, for example Routing Information Protocol (RIP for short) [18, 33], Open Shortest Path First (OSPF for short) [36], and ReSerVation Protocol (RSVP for short) [4], define their own routing messages that routers can use to exchange routing information with other routers on the network.

The protection of routing messages is important because routing messages carry routing information that is vital to the correctness of routing protocols. If routing messages are messed up by an adversary, the operation of routing protocols will be disrupted and normal data messages will not be correctly routed toward their ultimate destination.

There have been a number of works on how to extend specific routing protocols to make them more secure. However, if strong hop integrity protocol is deployed in each pair of adjacent routers in a network, then without any other security mechanism added, any routing protocol that is used in the network gets secured. In the following three subsections, we discuss how strong hop integrity can be applied to enhance the security of RIP, OSPF, and RSVP.

3.1 Security of RIP

RIP [18], which is shorthand for Routing Information Protocol, is a widely used routing protocol for IP-based networks. RIP allows a router to exchange routing information with its adjacent routers. It is a distance-vector protocol, which means that the routing information a router receives from an adjacent router is a vector of distances (measured in the number of hops) from the adjacent router to all possible destinations in the network. Each router then independently uses the routing information it receives from its adjacent routers to compute its best routes to all possible destinations in the network. (At the beginning of the execution of RIP, the routes computed by one router may not conform to those computed by another router, because initially a router does not have much routing information about the network. However, with the periodical update, routing information of each router will spread over the network and eventually the routes computed by different routers will converge to be consistent with each other.)

There are two types of messages used in RIP, namely request and response messages. A router can send a request message to its adjacent routers to ask these routers to send back their current routing tables. A router that receives a request message is required to return a response message that contains its own routing table. Moreover, a router sends a response message to all its adjacent routers every 30 seconds.

There is a security need for protecting the response messages that contain routing information in RIP. In the absence of any protection for response messages, an adversary sitting between two routers in the network can disrupt the network in several ways. First, the adversary can either insert a fake response message with incorrect routing information that it fabricates. Second, the adversary can modify a correct response message and make its routing information incorrect. Third, the adversary can also replay a previous

response message whose routing information is no longer correct. When a router receives a response message with incorrect routing information (from the adversary), it will unsuspectingly accept the message and use the incorrect routing information contained in the message to update its own routing table. Even worse, the router will send its routing table (incorrect now) to all adjacent routers. Consequently, the router may compute bad routes for destinations because of the false routing information it receives, and routing loops may be formed because of the spread of false routing information.

The original RIP does not have any mechanism for authenticating the response messages. In RIP version 2 [33], a simple authentication mechanism is added to every response message in the protocol: a 16-byte clear text password is inserted in every response message. This authentication mechanism is easy, but cannot provide enough protection because the adversary can easily copy the password and use it in the fake response messages it inserts, or copy a response message and replay it later.

By contrast, strong hop integrity can protect a network against the three attacks on RIP mentioned above. If strong hop integrity protocol is implemented in each pair of adjacent routers in a network, then each pair of adjacent routers in this network share two unique secrets (one for each direction) that can be used to compute an integrity check for every message exchanged between this pair of routers. Therefore, every RIP response message exchanged between any pair of adjacent routers in the network is protected by an integrity check and a (soft or hard) sequence number added by the sending router. If an adversary launches against the network the first attack or the second attack, namely inserting a fake RIP response message or modifying a correct RIP response message between any pair of adjacent routers, then the inserted or modified response message will be detected and discarded by the router that receives this message because the integrity check contained in this message is not correct. If an adversary launches against the network the third attack, namely replaying a previous RIP response message between any pair of adjacent routers, then the inserted or modified response message will be detected and discarded by the router that receives this message because the sequence number contained in this message is not correct. Therefore, strong hop integrity can secure RIP response messages, and can prevent an adversary from spreading false routing information over a network.

3.2 Security of OSPF

OSPF [36], which is shorthand for Open Shortest Path First, is another widely used routing protocol in the Internet. Unlike RIP, OSPF is a link-state

protocol, which means that each router gathers information on the state of its links to all adjacent routers and sends the link state information to all other routers in the network. The process that a router forwards to its adjacent routers every link state message it receives without change is called flooding. By periodical flooding, OSPF routers in the same network share a synchronized database that is consisted of link state records. These records represent the current topology of the network, and are used by OSPF routers to compute their best routes to destinations.

OSPF protocol consists of three sub-protocols: Hello, Exchange, and Flooding protocols. The Hello protocol is used to check whether an adjacent router and the link connecting to that router are up or not. A link between two routers is considered up if messages can go in both directions. After establishing their two-way connectivity, two routers can use the Exchange protocol to achieve the initial synchronization of their link state database by exchanging database description messages. A link might change its state as time goes by. Therefore, the router that is responsible for a link whose state has changed needs to advertise the new state of the link to all other routers in the network. This is done by using Flooding protocol to send a link state update message to all other routers, and other routers who receive this message should send back an acknowledgment message so as to keep every router's link state database synchronized.

The possible security threats faced by OSPF can be listed as follows. First, an adversary may insert a fake message that incorrectly advertises some link as the best route to other networks, so as to congest that link with high-volume misled traffic. Second, an adversary may modify a message that contains the state information of an important link, so that an area in the network might become unreachable. Third, an adversary may impersonate some router in a network and may insert a fake update message that requests all other routers to purge all link state records of the impersonated router. By repeating this trick, the adversary can slash the link state database in every router.

We discover that hop integrity can be used to counter the three attacks mentioned above. If hop integrity is implemented in each pair of adjacent routers in a network, then each pair of adjacent routers in this network share two unique secrets (one for each direction) that can be used to compute an integrity check for every message exchanged between this pair of routers. Therefore, every hello message, every database description message, every update message, and every acknowledgment message exchanged between any pair of adjacent routers in the network is protected by an integrity check added by the sending router. If an adversary inserts or modifies any OSPF message between any pair of adjacent routers, then the inserted or modified OSPF message will be detected and discarded by the first router that receives

this message because the integrity check contained in this message is not correct. Therefore, OSPF messages are secured by hop integrity and an adversary cannot use inserted or modified OSPF messages to mess up link state databases maintained by routers.

3.3 Security of RSVP

RSVP [4], which is shorthand of ReSerVation Protocol, is a resource reservation protocol designed for providing integrated services in the Internet. RSVP allows a host that wants to receive particular application data flows from a sending host to request from the network a specific degree of services in advance (although there is no guarantee that the requested service is available in the network). RSVP also allows a router to exchange service requests with other routers to establish and maintain state of the service it provides. Once the requested service is established, the host that requested the service is guaranteed that each router along the data path (between this host and the sending host) has reserved needed resources for the service the router promised to provide, and that the provided service will last till the end of the transmission of the data flow.

There are two main types of messages used in RSVP, namely Resv and Path messages. Each sending host periodically sends a Path message to all receiving hosts for a data flow that this sending host generates. The Path message is designed to mark the path that is traveled by data messages. Each router along the data path maintains a state that remembers the previous router corresponding to this particular data flow. With the path information marked by Path messages, each receiving host is able to send Resv messages, which contain the reservation requests, toward the sending host. When receiving a Resv message, each router on the path determines how many resources it can grant to this reservation request, and relays the Resv message toward the sending host.

The security issues concerned with RSVP are the integrity and authentication of service request messages. If an adversary spoofs the source address of a service request message, and the service request message is accepted by unsuspecting routers along the data path, then the adversary can steal the established service. If an adversary modifies the parameter of service specified in a service request message, or replays several service request messages and inserts them into the network, the normal service provided by the network may be severely reduced or totally denied.

An extension to RSVP [2] provides a mechanism to protect RSVP messages against message modification, message spoofing, and message replay. The proposed scheme uses a secret shared between a pair of adjacent RSVP routers to compute a keyed cryptographic digest of a RSVP message,

and includes the digest as part of the RSVP message. However, a working key management protocol is missing in that proposal and manual key management may be necessary at its current stage.

By contrast, if strong hop integrity along with ingress filtering is deployed in the network, then not only RSVP messages, but all other types of messages will also be protected against message modification, message spoofing, and message replay. Moreover, hop integrity is easier to manage because it updates shared secrets in a distributed way (by each pair of adjacent routers themselves).

4. SECURITY IN AD HOC NETWORKS AND SENSOR NETWORKS

Ad hoc networks and sensor networks are two new types of networks that have found many applications in today's world. Because of the ease and flexibility in their deployment, these types of networks are widely used in situations in which information exchange and/or aggregation is needed but communication infrastructure is unavailable, for example battlegrounds, disaster rescue sites, and construction sites.

There are two common characteristics of ad hoc networks and sensor networks. First, the nodes in these networks communicate with each other through wireless media. Therefore, no infrastructure is needed for the deployment of these networks. Second, there is no dedicated router in these networks. Instead, each node in these networks also plays the role of a router, in that each node, when receiving a message not destined for it, will forward the message to a neighboring node that is closer to the ultimate destination.

However, the ad hoc nature of these types of networks also makes them vulnerable to message insertion and message modification attacks as follows. If an adversary inserts a new message between two neighboring nodes, or arbitrarily modifies a message in transit, then the inserted or modified message will be forwarded by unwitting nodes that receive this message towards the ultimate destination.

To counter the aforementioned attacks, a variation of hop integrity can be implemented in these types of networks as follows. Before the deployment, each node in an ad hoc network or a sensor network is loaded with a group secret that is only known to the nodes belonging to the network. If every transmitted message is appended with a piece of integrity check information computed using the group secret, then any inserted or modified message will be detected and discarded by a node that receives the message next. The

TinySec project conducted by Karlof et al. [31] also proposes a scheme that is similar to hop integrity.

References

[1] Atkins, D., et al., Internet Security, 2nd edition, New Riders, 1997.

[2] Baker, F., B. Lindell, M. Talwar, "RSVP Cryptographic Authentication", RFC 2747, January 2000.

[3] Bellovin, S., M. Leech, T. Taylor, "ICMP Traceback Messages", Internet Draft, work in progress, October 2001.

[4] Braden, R., L. Zhang, S. Berson, S. Herzog, S. Jamin, "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification", RFC 2747, January 2000.

[5] "TCP SYN Flooding and IP Spoofing Attacks", CERT Advisory CA-96.21, available at http://www.cert.org/.

[6] Cheung, S., "An Efficient Message Authentication Scheme for Link State Routing", Proceedings of the 13th Annual Computer Security Applications Conference, San Diego, California, December 1997, pp. 90-98.

[7] Comer, D. E., Internetworking with TCP/IP: Vol. I: Principles, Protocols, and Architecture, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[8] Carl-Mitchell, S., J. S. Quarterman, "Using ARP to Implement Transparent Subnet Gateways", RFC 1027, October 1987.

[9] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, March 1997.

[10] Deering, S., "Host Extensions for IP Multicasting", RFC 1054, May 1988.

[11] Estrin, D., D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, L. Wei, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification", RFC 2362, June 1998.

[12] Ferguson, P., D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", RFC 2827, May 2000.

[13] Gouda, M. G., Elements of Network Protocol Design, John Wiley & Sons, New York, NY, 1998.

[14] Gouda, M. G., E. N. Elnozahy, C.-T. Huang, E. Jung, "An Overview of Hop Integrity", Proceeding of 2001 IBM Austin Center for Advanced Studies Conference, 2001.

[15] Gouda, M. G., E. N. Elnozahy, C.-T. Huang, T. M. McGuire, "Hop Integrity in Computer Networks", IEEE/ACM Transactions on Networking, Vol. 10, No. 3, June 2002, pp. 308-319.

[16] Gouda, M. G., C.-T. Huang, "A Secure Address Resolution Protocol", Computer Networks, Vol. 41, No. 1, January 2003, pp. 57-71.
[17] Gouda, M. G., C.-T. Huang, E. Li, "Anti-Replay Window Protocols for Secure IP", Proceedings of 9th International Conference on Computer Communications and Networks, Las Vegas, October 2000.
[18] Hedrick, C., "Routing Information Protocol", RFC 1058, June 1988.
[19] Huang, C.-T., "Hop Integrity: A Defense against Denial-of-Service Attacks", Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, August 2003.
[20] Huang, G., S. Beaulieu, D. Rochefort, "A Traffic-Based Method of Detecting Dead IKE Peers", Internet Draft, draft-ietf-ipsec-dpd-01.txt, August 2001.
[21] Huang, C.-T., E. N. Elnozahy, M. G. Gouda, "Hop Integrity and the Security of Routing Protocols", Proceedings of 2002 IBM Austin Center for Advanced Studies Conference, 2002.
[22] Huang, C.-T., M. G. Gouda, E. N. Elnozahy, "Convergence of IPsec in Presence of Resets", Proceedings of 2nd International Workshop on Assurance in Distributed Systems and Networks, Providence, May 2003.
[23] Hussain, A., J. Heidemann, C. Papadopoulos, "A Framework for Classifying Denial of Service Attacks", Proceedings of SIGCOMM'03, Karlsruhe, Germany, August 2003.
[24] Joncheray, L., "A Simple Active Attack Against TCP", Proceedings of the 5th USENIX UNIX Security Symposium, 1995, pp. 7-19.
[25] Ji, P., Z. Ge, J. Kurose, D. Towsley, "A Comparison of Hard-state and Soft-state Signaling Protocols", Proceedings of SIGCOMM'03, Karlsruhe, Germany, August 2003.
[26] Kent, S., and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.
[27] Kent, S., and R. Atkinson, "IP Authentication Header", RFC 2402, November 1998.
[28] Kent, S., and R. Atkinson, "IP Encapsulating Security Payload (ESP)", RFC 2406, November 1998.
[29] Krawczyk, H., M. Bellare, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
[30] Krywaniuk, A., T. Kivinen, "Using Isakmp Heartbeats for Dead Peer Detection", Internet Draft, draft-ietf-ipsec-heartbeats-01.txt, July 2000.
[31] Karlof, C., N. Sastry, D. Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks", Proceedings of SenSys'04, Baltimore, November 2004.
[32] Network Research Group, Lawrence Berkeley National Laboratory, ARPWATCH 2.0, available at: ftp://ftp.ee.lbl.gov/arpwatch.tar.Z.
[33] Malkin, G., "RIP Version 2: Carrying Additional Information", RFC 1723, November 1994.
[34] Montenegro, G., "Reverse Tunneling for Mobile IP", RFC 2344, May 1998.
[35] Montenegro, G., "Reverse Tunneling for Mobile IP, revised", RFC 3024, January 2001.
[36] Moy, J., "OSPF Version 2", RFC 1583, March 1994.
[37] Murphy, S., and M. Badger, "Digital Signature Protection of the OSPF Routing Protocol", Proceedings of the 1996 Internet Society Symposium on Network and Distributed Systems Security, San Diego, California, February 1996.
[38] Maughan, D., M. Schertler, M. Schneider, and J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)", RFC 2408, November 1998.
[39] NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.
[40] Orman, H., "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.

[41] Perkins, C., Ed., "IP Mobility Support for Ipv4", RFC 3220, January 2002.
[42] Plummer, D. C., "An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware", RFC 826, November 1982.
[43] Postel, J., "Internet Control Message Protocol", RFC 792, September 1981.
[44] Rivest, R. L., "The MD5 Message-Digest Algorithm", RFC 1321, 1992.
[45] Skoudis, E., Counter Hack: A Step-by-Step Guide to Computer Attacks and Efficient Defenses, Prentice Hall PTR, 2001.
[46] Stevens, W. R., TCP/IP Illustrated, Vol. I: The Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[47] Smith, B., S. Murthy, and J. J. Garcia-Luna-Aceves, "Securing Distance Vector Routing Protocols", Proceedings of the 1997 Internet Society Symposium on Network and Distributed Systems Security, San Diego, California, February 1997.
[48] Snoeren, A., C. Patridge, L. Sanchez, C. Jones, F. Tchakounito, S. Kent, W. Strayer, "Hash-Based IP Traceback", Proceedings of ACM SIGCOMM'01, San Diego, California, August 2001.
[49] Savage, S., D. Wetherall, A. Karlin, and T. Anderson, "Network Support for IP Traceback", IEEE/ACM Transactions on Networking, Vol. 9, No. 3, June 2001.
[50] The User-Mode Linux Kernel Home Page, http://user-mode-linux.sourceforge.net.
[51] De Vivo, M., G. de Vivo, and G. Isern, "Internet Security Attacks at the Basic Levels", Operating Systems Review, Vol. 32, No. 2, SIGOPS, ACM, April 1998.
[52] Whalen, S., An Introduction to ARP Spoofing, April 2001, http://chocobospore.org/arpspoof.
[53] Waitzman, D., C. Partridge, S. Deering, "Distance Vector Multicast Routing Protocol", RFC 1075, November 1988.
[54] 2004 CSI/FBI Computer Crime and Security Survey, available at http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2004.pdf.

Index