

# Hop Integrity in Computer Networks

Mohamed G. Gouda, *Member, IEEE*, E. N. (Mootaz) Elnozahy, *Member, IEEE*,  
Chin-Tser Huang, *Student Member, IEEE*, and Tommy M. McGuire

**Abstract**—A computer network is said to provide hop integrity iff when any router  $p$  in the network receives a message  $m$  supposedly from an adjacent router  $q$ , then  $p$  can check that  $m$  was indeed sent by  $q$ , was not modified after it was sent, and was not a replay of an old message sent from  $q$  to  $p$ . In this paper, we describe three protocols that can be added to the routers in a computer network so that the network can provide hop integrity, and thus overcome most denial-of-service attacks. These three protocols are a secret exchange protocol, a weak integrity protocol, and a strong integrity protocol. All three protocols are stateless, require small overhead, and do not constrain the network protocol in the routers in any way.

**Index Terms**—Authentication, denial-of-service attack, Internet, message modification, message replay, network protocol, router, security, smurf attack, SYN attack.

## I. INTRODUCTION

**M**OST COMPUTER networks 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, whose aim is to exhaust the communicating resources of a network or the computing resources of a host, such that normal services provided by the network or the host are reduced or completely denied. Two well-known examples of denial-of-service attacks in networks that support the Internet Protocol (IP) and the Transmission Control Protocol (TCP) are as follows.

### A. 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 (ICMP) [17]. 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.

### B. 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 [3], [23]. 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 [9]) types of attacks, an adversary inserts into the network messages with wrong original sources. These messages are accepted by unsuspecting routers and routed toward the computer under attack. To counter these attacks, each router  $p$  in the network should route a received  $m$  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. In

Manuscript received December 14, 2000; revised November 17, 2001; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Lee. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) Contract F33615-01-C-1901, and in part by two IBM Faculty Partnership Awards for the academic years of 2000–2001 and 2001–2002, and by the Texas Advanced Research Program, Texas Higher Education Coordinating Board, under Grant TARP 14-970823.

M. G. Gouda, C.-T. Huang, and T. M. McGuire are with the Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-1188 USA (e-mail: gouda@cs.utexas.edu; chuang@cs.utexas.edu; mcguire@cs.utexas.edu).

E. N. Elnozahy is with the System Software Department, IBM Austin Research Lab, Austin, TX 78758 USA (e-mail: mootaz@us.ibm.com).

Publisher Item Identifier S 1063-6692(02)05232-9.

this paper, we present a suite of protocols that provide hop integrity between adjacent routers: whenever a router  $p$  receives a message  $m$  from an adjacent router  $q$ ,  $p$  can detect whether  $m$  was indeed sent by  $q$  or it was modified or replayed by an adversary that operates between  $p$  and  $q$ .

It is instructive to compare hop integrity with secure routing [4], [14], [20], ingress filtering [6], traceback [2], [22], [21], and IPsec [10]. In secure routing, for example [4], [14], and [20], 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.

Using ingress filtering [6], each router on the network boundary checks whether the recorded source in each received message is consistent with the source from which the router received the message. If the message source is consistent, the router forwards the message as usual. Otherwise, the router discards the message. Thus, ingress filtering detects messages whose recorded sources are modified (to hide the true sources of these messages), provided that these modifications occur at the network boundary. Messages whose recorded sources are modified between adjacent routers in the middle of the network will not be detected by ingress filtering, but will be detected and discarded by hop integrity.

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: the message marking scheme [2], [22] and the hash-based scheme [21]. In the 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 the IP header of message  $m$  itself [22], or sent in a separate ICMP message [2]. Due to the overhead incurred by sending traceback information, both Bellovin [2] and Savage [22] employ probabilistic methods rather than applying their methods to every message. In the 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 in this paper and the IPsec protocol suite presented in [10]–[12], [15], and [16] are both intended to provide security at the IP 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.

In the next section, we describe the concept of hop integrity in some detail.

## II. HOP INTEGRITY PROTOCOL

A network consists of computers connected to subnetworks. (Examples of subnetworks are local area networks, telephone lines, and satellite links.) Two computers in a network are called adjacent iff both computers are connected to the same subnetwork. Two adjacent computers in a network can exchange messages over any common subnetwork to which they are both connected.

The computers in a network are classified into hosts and routers. For simplicity, we assume that each host in a network is connected to one subnetwork, and each router is connected to two or more subnetworks. A message  $m$  is transmitted from a computer  $s$  to a faraway computer  $d$  in the same network as follows. First, message  $m$  is transmitted in one hop from computer  $s$  to a router  $r.1$  adjacent to  $s$ . Second, message  $m$  is transmitted in one hop from router  $r.1$  to router  $r.2$  adjacent to  $r.1$ , and so on. Finally, message  $m$  is transmitted in one hop from a router  $r.n$  that is adjacent to computer  $d$  to computer  $d$ .

A network is said to provide hop integrity iff the following two conditions hold for every pair of adjacent routers  $p$  and  $q$  in the network.

- 1) Whenever router  $p$  receives a message  $m$  over the subnetwork connecting routers  $p$  and  $q$ ,  $p$  can determine correctly whether message  $m$  was modified by an adversary after it was sent by  $q$  and before it was received by  $p$ .
- 2) Whenever router  $p$  receives a message  $m$  over the subnetwork connecting routers  $p$  and  $q$ , and determines that message  $m$  was not modified, then  $p$  can determine correctly whether message  $m$  is another copy of a message that is received earlier by  $p$ .

For a network to provide hop integrity, two “thin” protocol layers need to be added to the protocol stack in each router in the network. As discussed in [5] and [19], 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 two thin layers that need to be added to this protocol stack are the secret exchange layer and the integrity check layer. The secret exchange layer is added above the network layer (and below the transport layer), and the integrity check layer is placed below the network layer (and above the subnetwork layer).

The function of the secret exchange layer is to allow adjacent routers to periodically generate and exchange (and so share)

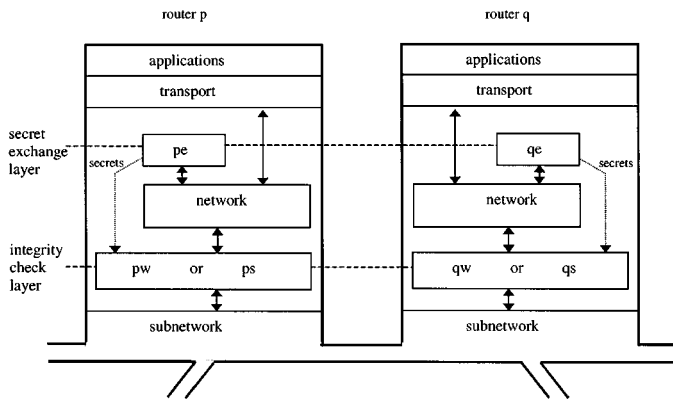


Fig. 1. Protocol stack for achieving hop integrity.

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.

Fig. 1 shows the protocol stacks in two adjacent routers  $p$  and  $q$ . The secret exchange layer consists of the two processes  $pe$  and  $qe$  in routers  $p$  and  $q$ , respectively. The integrity check layer has two versions, weak and strong. 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.

Next, we explain how hop integrity, along with ingress filtering, can be used to prevent smurf and SYN attacks (described in Section I). Recall that in smurf and SYN attacks, an adversary inserts into the network ping and SYN messages with wrong original sources. These forged messages can be inserted either through a boundary router or between two routers in the middle of the network. Ingress filtering (which is usually installed in boundary routers [6]) will detect the forged messages if they are inserted through a boundary router because the recorded sources in these messages would be inconsistent with the hosts from which these messages are received. However, ingress filtering may fail in detecting forged messages if these messages are inserted between two routers in the middle of the network. For example, an adversary can log into any host located between two routers  $p$  and  $q$ , and use this host to insert forged messages toward router  $p$ , pretending that these messages are sent by router  $q$ . The real source of these messages cannot be determined by router  $p$  because router  $p$  cannot decide whether these messages are sent by router  $q$  or by some host between  $p$  and  $q$ . However, if hop integrity is installed between the two routers  $p$  and  $q$ , then the (weak or strong) integrity check layer in router  $p$  concludes that the forged messages have been modified after being sent by router  $q$  (although they are actually inserted by the adversary and not sent by router  $q$ ), and so it discards them.

Smurf and SYN attacks can also be launched by replaying old messages. For example, the adversary can log into any host located between two routers  $p$  and  $q$ . When the adversary spots some passing legitimate ping or SYN message being sent from  $q$  to  $p$ , it keeps a copy of the passing message. At a later time,

the adversary can replay these copied messages over and over to launch a smurf or SYN attack. Hop integrity can defeat this attack as follows. If hop integrity is installed between the two routers  $p$  and  $q$ , then the strong integrity check layer in router  $p$  can detect the replayed messages and discard them.

In the next three sections, we describe in some detail the protocols in the secret exchange layer and in the two versions of the integrity check layer. The first protocol between processes  $pe$  and  $qe$  is discussed in Section III. The second protocol between processes  $pw$  and  $qw$  is discussed in Section IV. The third protocol between processes  $ps$  and  $qs$  is discussed in Section V.

These three protocols are described using a variation of the Abstract Protocol Notation presented in [7]. In this notation, each process in a protocol is defined by a set of inputs, a set of variables, and a set of actions. For example, in a protocol consisting of processes  $px$  and  $qx$ , process  $px$  can be defined as follows.

```

process  $px$ 
inp   (name of input): (type of input)
    ...
    (name of input): (type of input)
var   (name of variable): (type of variable)
    ...
    (name of variable): (type of variable)
begin
    (action)
  || (action)
  || ...
  || (action)
end

```

Comments can be added anywhere in a process definition; each comment is placed between two brackets, { and }.

The inputs of process  $px$  can be read but not updated by the actions of process  $px$ . Thus, the value of each input of  $px$  is either fixed or is updated by another process outside the protocol consisting of  $px$  and  $qx$ . The variables of process  $px$  can be read and updated by the actions of process  $px$ . Each (action) of process  $px$  is of the form

(guard)  $\rightarrow$  (statement).

The (guard) of an action of  $px$  is either a (boolean expression) or a (receive) statement of the form

**rcv** (message) **from**  $qx$ .

The (statement) of an action of  $px$  is a sequence of skip, (assignment), (send), or (selection) statements. An (assignment) statement is of the form

(variable of  $px$ ) := (expression).

A (send) statement is of the form

**send** (message) **to**  $qx$ .

A  $\langle$ selection $\rangle$  statement is of the form

```

if  $\langle$ boolean expression $\rangle \rightarrow \langle$ statement $\rangle$ 
...
 $\|$  $\langle$ boolean expression $\rangle \rightarrow \langle$ statement $\rangle$ 
fi

```

Executing an action consists of executing the statement of this action. Executing the actions (of different processes) in a protocol proceeds according to the following three rules. First, an action is executed only when its guard is true. Second, the actions in a protocol are executed one at a time. Third, an action whose guard is continuously true is eventually executed.

Executing an action of process  $px$  can cause a message to be sent to process  $qx$ . There are two channels between the two processes: one is from  $px$  to  $qx$ , and the other is from  $qx$  to  $px$ . Each sent message from  $px$  to  $qx$  remains in the channel from  $px$  to  $qx$  until it is eventually received by process  $qx$  or is lost. Messages that reside simultaneously in a channel form a sequence  $\langle m.1, m.2, \dots, m.n \rangle$  in accordance with the order in which they have been sent. The head message in the sequence,  $m.1$ , is the earliest sent, and the tail message in the sequence,  $m.n$ , is the latest sent. The messages are to be received in the same order in which they were sent.

We assume that an adversary exists between processes  $px$  and  $qx$ , and that this adversary can perform the following three types of actions to disrupt the communications between  $px$  and  $qx$ . First, the adversary can perform a message loss action where it discards the head message from one of the two channels between  $px$  and  $qx$ . Second, the adversary can perform a message modification action where it arbitrarily modifies the contents of the head message in one of the two channels between  $px$  and  $qx$ . Third, the adversary can perform a message replay action where it replaces the head message in one of the two channels by a message that was sent previously. For simplicity, we assume that each head message in one of the two channels between  $px$  and  $qx$  is affected by at most one adversary action.

### III. 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 have a public key and a private key that they use to encrypt and decrypt the messages that carry the new secret  $s$  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 request (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 reply (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  $sq[0] \neq sq[1]$  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

$$sq[0] \text{ in } pe = sq[1] \text{ in } pe = sq \text{ in } qe$$

and

$$sp[0] \text{ in } qe = sp[1] \text{ in } qe = sp \text{ in } pe.$$

Process  $pe$  can be defined as follows.

```

process  $pe$ 
inp    $R_p$ : integer {private key of  $pe$ }
         $B_q$ : 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  $sq[0] = sq[1] \wedge$ 

```

```

    (te hours passed since rqst message sent
last) →
    sq[1] := NEWSQR;
    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 reply(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

```

Process  $q_e$  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  $qe$ .

The four actions of process  $pe$  use three functions NEWSQR, NCR, and DCR, defined as follows. Function NEWSQR 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

$$e := \text{NCR}(B_q, (sq[0]; sq[1]))$$

causes the concatenation of  $sq[0]$  and  $sq[1]$  to be encrypted using the public key  $B_q$ , 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

$$d := \text{DCR}(R_p, e)$$

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

$$(d, e) := \text{DCR}(R_p, e).$$

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  $R_p$ ,

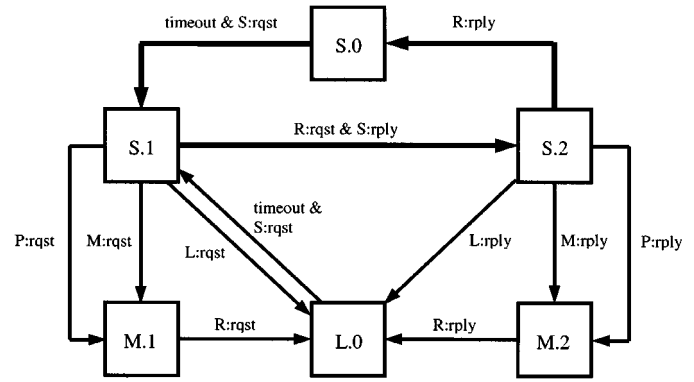


Fig. 2. State transition diagram of the secret exchange protocol.

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, refer to the state transition diagram of this protocol in Fig. 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. For convenience, each transition is labeled by the message event that is executed during that transition. In particular, each transition has a label of the form

$$\langle \text{event type} \rangle : \langle \text{message type} \rangle$$

where  $\langle \text{event type} \rangle$  is one of the following.

$S$  stands for sending a message of the specified type.

$R$  stands for receiving 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.

The notation  $ch_{pe,qe}$  is used to denote the content of the channel from process  $pe$  to process  $qe$ .

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:

$$S.0 = ch_{pe,qe} = \langle \rangle \wedge ch_{qe,pe} = \langle \rangle \wedge sq[0] \text{ in } pe = sq[1] \text{ in } pe = sq \text{ in } qe.$$

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:

$$S.1 = ch_{pe,qe} = \langle \text{rqst}(e) \rangle \wedge ch_{qe,pe} = \langle \rangle \wedge e = \text{NCR}(B_q, (sq[0]; sq[1])) \wedge sq[0] \text{ in } pe \neq sq[1] \text{ in } pe \wedge sq[0] \text{ in } pe = sq \text{ in } qe.$$

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:

$$S.2 = ch_{pe,qe} = \langle \rangle \wedge ch_{qe,pe} = \langle \text{rply}(e) \rangle \wedge$$

$$e = \text{NCR}(B_p, sq) \wedge \\ sq[0] \text{ in } pe \neq sq[1] \text{ in } pe \wedge sq[1] \text{ in } pe = sq \text{ in } qe.$$

At state  $S.2$ , exactly one legitimate action, namely, the receive action (that receives a reply 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:

$$L.0 = ch.pe.qe = \langle \rangle \wedge ch.qe.pe = \langle \rangle \wedge \\ sq[0] \text{ in } pe \neq sq[1] \text{ in } pe \wedge \\ (sq[0] \text{ in } pe = sq \text{ in } qe \vee sq[1] \text{ in } pe = sq \text{ in } qe).$$

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:

$$M.1 = ch.pe.qe = \langle \text{rqst}(e) \rangle \wedge ch.qe.pe = \langle \rangle \wedge \\ e \neq \text{NCR}(B_q, (sq[0]; sq[1])) \wedge \\ sq[0] \text{ in } pe \neq sq[1] \text{ in } pe \wedge \\ (sq[0] \text{ in } pe = sq \text{ in } qe \vee sq[1] \text{ in } pe = sq \text{ in } qe).$$

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

$$M.2 = ch.pe.qe = \langle \rangle \wedge ch.qe.pe = \langle \text{rply}(e) \rangle \wedge \\ e \neq \text{NCR}(B_p, sq) \wedge \\ sq[0] \text{ in } pe \neq sq[1] \text{ in } pe \wedge \\ (sq[0] \text{ in } pe = sq \text{ in } qe \vee sq[1] \text{ in } pe = sq \text{ in } qe).$$

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 Fig. 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 correctness of the secret exchange protocol is verified.

#### IV. WEAK INTEGRITY PROTOCOL

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

$$d := \text{MD}(t; scr)$$

where  $\text{MD}$  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  $\text{data}(t, d)$  message to router  $r.2$ .

When the second router  $r.2$  receives the  $\text{data}(t, d)$  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  $\text{data}(t, d)$  message.

When the last router  $r.n$  receives the  $\text{data}(t, d)$  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  $\text{data}(t)$  message to its destination  $\text{dst}$ .

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 protocol has two inputs  $sp$  and  $sq$  that  $pw$  reads but never updates. These two inputs 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
inp   sp: integer
       sq: array [0 .. 1] of integer
var
       t, d: integer
begin
  rcv data(t, d) from qw  $\rightarrow$ 
  if MD(t; sq[0]) = d  $\vee$  MD(t; sq[1]) = d  $\rightarrow$ 
    {defined later} RTMSG
  |MD(t; sq[0])  $\neq$  d  $\wedge$  MD(t; sq[1])  $\neq$  d  $\rightarrow$ 
    {report 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  $sq[0] \neq sq[1]$ , 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.

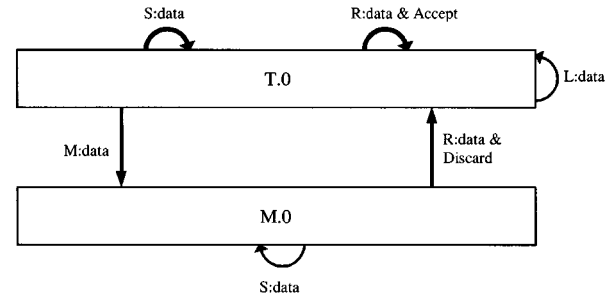
```

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, refer to the state transition diagram of this protocol in Fig. 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.

Note that because the weak integrity protocol operates below the secret exchange protocol in the protocol stack, we can assert that  $(sq \text{ in } qw = sq[0] \text{ in } pw \vee sq \text{ in } qw = sq[1] \text{ in } pw)$  is an invariant in every state of the weak integrity protocol. We denote this invariant as  $I$  in the specification in Fig. 3. Also, note that the notation  $\text{Head}(\text{data}(t, d))$  in the specification in Fig. 3 is a predicate whose value is true iff  $\text{data}(t, d)$  is the head message of the specified channel.



$T.0 = I \wedge (\forall \text{data}(t, d) \text{ message in ch.qw.pw, } d = \text{MD}(t; sq))$ $M.0 = I \wedge (\forall \text{data}(t, d) \text{ message in ch.qw.pw,}$ $(\neg \text{Head}(\text{data}(t, d)) \Rightarrow d = \text{MD}(t; sq)) \wedge$ $(\text{Head}(\text{data}(t, d)) \Rightarrow d \neq \text{MD}(t; sq)))$ where $I = sq \text{ in } qw = sq[0] \text{ in } pw \vee sq \text{ in } qw = sq[1] \text{ in } pw$
--

Fig. 3. State transition diagram of the weak integrity protocol.

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  $\text{data}(t, d)$  in the channel from  $qw$  to  $pw$ ,  $d = \text{MD}(t; sq)$  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. However, the weak integrity protocol, while being able to detect and discard all modified messages, cannot detect some replayed messages. In the next section, we introduce the strong integrity protocol that is capable of detecting and discarding all modified and replayed messages.

## V. STRONG INTEGRITY PROTOCOL

In this section, we discuss how to strengthen the weak integrity protocol to make it detect message replay. We present the strong 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 combine

this protocol with the weak integrity protocol to form the strong integrity protocol.

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 next data message to be received by  $v$ .

To send the next `data(s)` message, process  $u$  assigns  $s$  the current value of variable `nxt`, then increments `nxt` by one. Assume that no more than  $L$  consecutive messages can get lost in transit. When process  $v$  receives a `data(s)` message,  $v$  compares its variable `exp` with  $s$ . If  $\text{exp} \leq s \leq \text{exp} + L$ , then  $v$  accepts the received `data(s)` message and assigns `exp` the value  $s + 1$ ; otherwise,  $v$  discards the `data(s)` message.

Correctness of this protocol is based on the observation that the predicate  $\text{exp} \leq \text{nxt}$  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 much larger than the value of `nxt`, then all the data messages that  $u$  sends from this point on will be wrongly discarded by  $v$  until `nxt` becomes equal to `exp`. Next, we describe how to modify this protocol such that the number of `data(s)` 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 much larger than one.

The modification consists of adding to process  $v$  two variables  $c$  and  $c \text{ max}$ , whose values are in the range  $0 \dots N - 1$ . When process  $v$  receives a `data(s)` message,  $v$  compares the values of  $c$  and  $c \text{ max}$ . If  $c \neq c \text{ max}$ , then process  $v$  increments  $c$  by one (mod  $N$ ) and proceeds as before [namely, either accepts the `data(s)` message if  $\text{exp} \leq s \leq \text{exp} + L$ , or discards the message if  $\text{exp} > s$  or  $\text{exp} + L < s$ ]. Otherwise,  $v$  accepts the message, assigns  $c$  the value 0, and assigns  $c \text{ max}$  a random integer in the range  $0 \dots N - 1$ .

This modification achieves two objectives. 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 such predictions by sending replayed data messages at those instants.

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

```

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

```

```

process  $v$ 
inp    $N$ : integer
         $L$ : integer
var    $s$ : integer
        {sequence number of received message}
        exp: integer
            {sequence number of next}
            {expected message}
         $c, c \text{ max}$ :  $0 \dots N - 1$ 
begin
    rcv data(s) from  $u \rightarrow$ 
        if ( $s < \text{exp} \vee s > \text{exp} + L$ )  $\wedge c \neq c \text{ max} \rightarrow$ 
            {reject message; report an
            adversary}
             $c := (c + 1) \bmod N$ 
         $\llbracket (\text{exp} \leq s \leq \text{exp} + L) \vee c = c \text{ max} \rightarrow$ 
            {accept message}
            exp :=  $s + 1$ ;
            if  $c \neq c \text{ max} \rightarrow c := (c + 1) \bmod N$ 
             $\llbracket c = c \text{ max} \rightarrow$ 
                 $c := 0$ ;
                 $c \text{ max} := \text{RANDOM}(0, N - 1)$ 
            fi
        fi
end.

```

Processes  $u$  and  $v$  of the soft sequence number protocol can be combined with process  $pw$  of the weak integrity protocol to construct process  $ps$  of the strong integrity 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 protocol can be defined as follows. (Process  $qs$  can be defined in the same way.)

```

process  $ps$ 
inp    $sq$ : integer
         $sq$ : array [0 ... 1] of integer
         $N$ : integer
         $L$ : integer
var    $s, t, d$ : integer
        exp, nxt: integer
         $c, c \text{ max}$ :  $0 \dots N - 1$ 
begin
    rcv data(s, t, d) from  $qs \rightarrow$ 
        if  $\text{MD}(s; t; sq[0]) = d \vee \text{MD}(s; t; sq[1]) = d \rightarrow$ 
            if ( $s < \text{exp} \vee s > \text{exp} + L$ )  $\wedge c \neq c \text{ max} \rightarrow$ 
                {reject message; report an
                adversary}
                 $c := (c + 1) \bmod N$ 
             $\llbracket (\text{exp} \leq s \leq \text{exp} + L) \vee c = c \text{ max} \rightarrow$ 
                {accept message}
                exp :=  $s + 1$ ;
                if  $c \neq c \text{ max} \rightarrow$ 
                     $c := (c + 1) \bmod N$ 
                fi
        fi
end.

```



```

    || c = c max →
      c := 0;
      c max := 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;}
  {compute d as the message digest of
  the}
  {concatenation of snxt, 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 protocol, refer to the state transition diagram of this protocol in Fig. 4, which considers the channel from process  $qs$  to process  $ps$ . (The channel from  $ps$  to  $qs$ , and the channels from  $ps$  to any other strong integrity 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 protocol operates below the secret exchange protocol in the protocol stack, we can assert that  $(sq \text{ in } qs = sq[0] \text{ in } ps \vee sq \text{ in } qs = sq[1] \text{ in } ps)$  is an in-

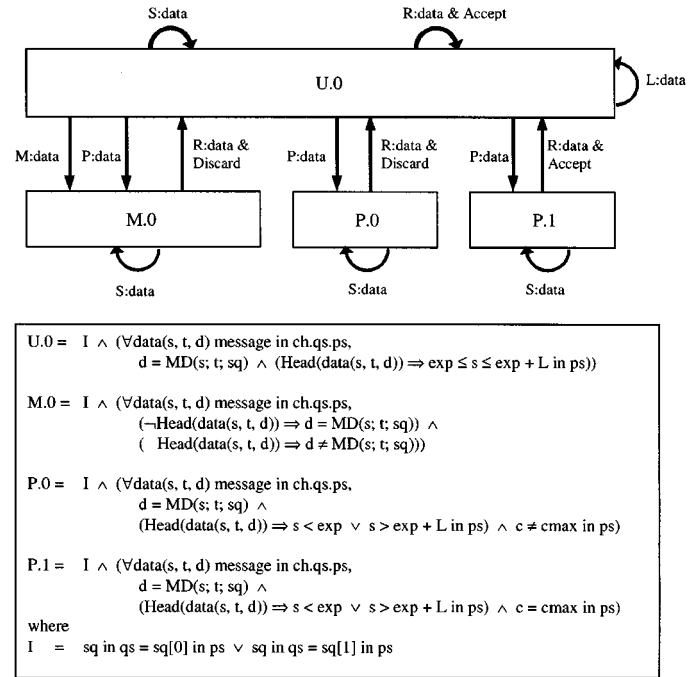


Fig. 4. State transition diagram of the strong integrity protocol.

variant in every state of the strong integrity protocol. We denote this invariant as  $I$  in the specification in Fig. 4.

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 sequence of the transitions 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 state  $U.0$ . If the adversary executes a message loss action at state  $U.0$ , the predicate that for every data message  $\text{data}(s, t, d)$  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  $\text{data}(s, t, d)$  is too old such that the secret used to compute the message digest is different from the current value of input  $sq$  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  $\text{data}(s, t, d)$  is recent such that the secret used to compute the message digest is equal to the current value of input  $sq$  in process  $qs$ , then the protocol moves either to state  $P.0$  or to state  $P.1$ . With a high probability of  $(c \text{max} - 1)/c \text{max}$ , the protocol moves to state  $P.0$ , and the re-

played message will be received and discarded by  $ps$  because the value of field  $s$  in the message tells that the message is replayed. With a probability of  $1/c_{\max}$ , 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 the high probability  $(c_{\max} - 1)/c_{\max}$ .

## VI. IMPLEMENTATION CONSIDERATIONS

In this section, we discuss several issues concerning the implementation of the hop integrity protocols presented in the previous three sections. In particular, we discuss acceptable values for the inputs of each of these protocols.

There are four inputs in the secret exchange protocol in Section III. They are  $R_p$ ,  $B_q$ ,  $te$ , and  $tr$ . 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 B (1024 bits) each. There are no special requirements for the encryption and decryption functions that use these keys in the secret exchange protocol.

Input  $te$  is the time period between two successive secret exchanges between  $pe$  and  $qe$ . This time period should be small enough that an adversary does not have enough time to deduce the secrets  $sp$  and  $sq$  used in computing the integrity checks of data messages. It should also be large enough that the overhead that results from the secret exchanges is reduced. An acceptable value for  $te$  is around 4 hours.

Input  $tr$  is the time-out 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 s.

Next, we consider the two inputs  $sp$  and  $sq$  and function MD used in the integrity protocols in Sections IV and V. 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 B. Function MD is used to compute the digest of a data message. Function MD is computed in two steps as follows. First, the standard function MD5 [18] is used to compute a 16-byte digest of the data message. Second, the first 4 B 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 730-MHz machine running Linux. It is not a significant overhead to a router.)

The soft sequence numbers in Section V can be recycled provided that not each of the sequence numbers has been used at least once in time period  $te$ . In a usual Ethernet, at most 800 messages can be sent in a second, thus at most 11 520 000 mes-

sages can be sent in a period of 4 hours. Using 4 B to store the soft sequence numbers is a proper choice with considerations of covering the maximum number of consumed sequence numbers in time period  $te$  and aligning with the original IP header.

As discussed in Section V, input  $N$  needs to be much larger than 1. For example,  $N$  can be chosen to be 200. In this case, the maximum number of messages that can be discarded wrongly whenever synchronization between two adjacent routers is lost is 200, and the probability of detecting an adversary which replays an old message is 99%.

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

## VII. CONCLUDING REMARKS

In this paper, we introduced the concept of hop integrity in computer networks. A network is said to provide hop integrity iff whenever a router  $p$  receives a message supposedly from an adjacent router  $q$ , router  $p$  can check whether the received message was indeed sent by  $q$  or was modified or replayed by an adversary that operates between  $p$  and  $q$ .

The effectiveness of hop integrity is apparent in those situations where ingress filtering is not effective. For example, ingress filtering can detect and discard messages with wrongly recorded sources at the network boundary, but cannot do so between adjacent routers in the middle of the network. By contrast, hop integrity can detect and discard messages with wrongly recorded source between adjacent routers in the middle of the network.

Moreover, ingress filtering is not compatible with mobile IP. A message sent by a mobile node and forwarded by the foreign agent (of this mobile node) will be filtered out by the next router because the recorded source of the message seems wrong to the router. By contrast, hop integrity can guarantee that every message forwarded by the foreign agent will be accepted by the router. (Reverse tunneling [13] was proposed to remedy this problem, but the cost of using reverse tunneling is high because every message that is sent by a mobile node has to be tunneled back to the home agent of the mobile node before the message can be forwarded.)

We presented three protocols that can be used to make any computer network provide hop integrity. These three protocols are a secret exchange protocol, a weak integrity protocol, and a strong integrity protocol. These three protocols 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 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 strong integrity protocol uses soft sequence numbers to make the protocol tolerate any loss of synchronization.

It is possible to reduce the overhead induced by the three protocols as follows. If in a network some hops are assured to be adversary-proof (for example, the hops between core routers),

then hop integrity does not need to be implemented over these hops. When a router needs to forward a message over an adversary-proof hop, it just follows the same procedure as in normal IP.

All three protocols are stateless, require small overhead at each hop, and do not constrain the network protocol in any way. Thus, we believe that they are compatible with IP in the Internet, and it remains to estimate or measure the performance of IP when augmented with these protocols.

Recently, hop integrity has been proposed to protect the messages of particular protocols, for example, RSVP [1]. By contrast, our proposed protocol suite is more general because it protects every message transmitted through the network, and is easier to manage because it updates shared secrets in a distributed way.

#### ACKNOWLEDGMENT

The authors would like to thank Dr. D. Lee and the anonymous referees for their help in improving the paper's presentation.

#### REFERENCES

- [1] F. Baker, B. Lindell, and M. Talwar, "RSVP cryptographic authentication," RFC 2747, Jan. 2000.
- [2] S. Bellovin, M. Leech, and T. Taylor. (2001, Oct.) ICMP traceback messages. Internet Draft, work in progress. [Online]. Available: <http://www.research.att.com/~smb/papers/draft-bellovin-itrace-00.txt>.
- [3] TCP SYN flooding and IP spoofing attacks. CERT Advisory CA-96.21. [Online]. Available: <http://www.cert.org/advisories/CA-1996-21.html>.
- [4] S. Cheung, "An efficient message authentication scheme for link state routing," in *Proc. 13th Annu. Computer Security Applications Conf.*, San Diego, CA, Dec. 1997, pp. 90–98.
- [5] D. E. Comer, *Internetworking with TCP/IP: Vol. 1—Principles, Protocols, and Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [6] P. Ferguson and D. Senie, "Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing," RFC 2827, May 2000.
- [7] M. G. Gouda, *Elements of Network Protocol Design*, 1st ed. New York: Wiley, 1998.
- [8] M. G. Gouda, E. N. Elnozahy, C.-T. Huang, and T. M. McGuire, "Hop integrity in computer networks," in *Proc. IEEE Int. Conf. Network Protocols*, Osaka, Japan, Nov. 2000, pp. 3–11.
- [9] L. Joncheray, "A simple active attack against TCP," in *Proc. 5th USENIX UNIX Security Symp.*, 1995, pp. 7–19.
- [10] S. Kent and R. Atkinson, "Security architecture for the Internet protocol," RFC 2401, Nov. 1998.
- [11] —, "IP authentication header," RFC 2402, Nov. 1998.
- [12] —, "IP encapsulating security payload (ESP)," RFC 2406, Nov. 1998.
- [13] G. Montenegro, "Reverse tunneling for mobile IP," RFC 2344, May 1998.
- [14] S. Murphy and M. Badger, "Digital signature protection of the OSPF routing protocol," in *Proc. 1996 Internet Society Symp. Network and Distributed Systems Security*, San Diego, CA, Feb. 1996, pp. 93–102.
- [15] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet security association and key management protocol (ISAKMP)," RFC 2408, Nov. 1998.
- [16] H. Orman, "The OAKLEY key determination protocol," RFC 2412, Nov. 1998.
- [17] J. Postel, "Internet control message protocol," RFC 792, Sept. 1981.
- [18] R. L. Rivest, "The MD5 message-digest algorithm," RFC 1321, Apr. 1992.
- [19] W. R. Stevens, *TCP/IP Illustrated, Vol. 1—The Protocols*. Englewood Cliffs, NJ: Prentice Hall, 1994.
- [20] B. Smith, S. Murthy, and J. J. Garcia-Luna-Aceves, "Securing distance vector routing protocols," in *Proc. 1997 Internet Society Symp. Network and Distributed Systems Security*, San Diego, CA, Feb. 1997, pp. 85–92.
- [21] A. Snoeren, C. Patridge, L. Sanchez, C. Jones, F. Tchakounito, S. Kent, and W. Strayer, "Hash-based IP traceback," in *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001, pp. 3–14.
- [22] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Network support for IP traceback," *IEEE/ACM Trans. Networking*, vol. 9, pp. 226–237, June 2001.
- [23] M. de Vivo, G. de Vivo, and G. Isern, "Internet security attacks at the basic levels," *Oper. Syst. Rev.*, vol. 32, no. 2, pp. 4–15, Apr. 1998.



**Mohamed G. Gouda** (M'93) was born in Egypt. He received B.Sc. degrees in engineering and mathematics, both from Cairo University, Cairo, Egypt. He received the M.A. degree in mathematics from York University, Toronto, ON, Canada, and the M.S. and Ph.D. degrees in computer science from the University of Waterloo, Waterloo, ON, Canada.

He was with the Honeywell Corporate Technology Center, Minneapolis, MN, during 1977–1980. In 1980, he joined the University of Texas at Austin where he currently holds the Mike A. Myers Centennial Professorship in Computer Sciences. He spent one summer at Bell Labs, Murray Hill, NJ, one summer at MCC, Austin, and one winter at the Eindhoven Technical University, Eindhoven, The Netherlands. His research areas are distributed and concurrent computing and network protocols. In these areas, he has been working on abstraction, formality, correctness, nondeterminism, atomicity, reliability, security, convergence, and stabilization. He has published over 50 journal papers, and over 170 conference papers. He has supervised over 17 Ph.D. dissertations. He was the founding Editor-in-Chief of the Springer-Verlag journal *Distributed Computing* (1985–1989). He served on the editorial board of *Information Sciences* during 1996–1999, and he is currently on the editorial boards of *Distributed Computing* and the *Journal of High Speed Networks*. He is the author of the textbook *Elements of Network Protocol Design* (New York: Wiley, 1998). This is the first textbook where network protocols were presented in abstract and formal setting. Currently, he is writing the textbook *Elements of Secure Network Protocols*. He is an original member of the Austin Tuesday Afternoon Club.

Dr. Gouda is a member of the Association for Computing Machinery (1993). He is the 1993 winner of the Kuwait Award in Basic Sciences. He was the recipient of an IBM Faculty Partnership Award for the academic year 2000–2001 and again for the academic year 2001–2002. He won the 2001 IEEE Communication Society William R. Bennet Best Paper Award for his paper "Secure Group Communications Using Key Graphs," coauthored with C. K. Wong and S. S. Lam and published in the February 2000 issue of the IEEE/ACM TRANSACTIONS ON NETWORKING. He was the program committee chairman of the ACM SIGCOMM Symposium in 1989. He was the first program committee chairman of the IEEE International Conference on Network Protocols in 1993. He was the first program committee chairman of the IEEE Symposium on Advances in Computers and Communications, which was held in Egypt in 1995. He was the program committee chairman of the IEEE International Conference on Distributed Computing Systems in 1999. He is on the steering committee of the IEEE International Conference on Network Protocols.



**E. N. (Mootaz) Elnozahy** (M'87) received the B.S. degree in electrical engineering from Cairo University, Cairo, Egypt, in 1984, and the Ph.D. degree in computer science from Rice University, Houston, TX, in 1994.

He is currently a Research Staff Member with the IBM Austin Research Lab, Austin, TX, where he leads the System Software Department. He is also an Adjunct Professor at the University of Texas at Austin. His research interests include operating systems, fault tolerance, distributed systems, security, and low-power systems. He has published over 25 refereed articles in various aspects of distributed systems and has served on many conference committees.

Dr. Elnozahy is a member of the Association for Computing Machinery (1987). He serves as an associate editor for the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS.



**Chin-Tser Huang** (S'97) received the B.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1993, and the M.S. degree in computer sciences from the University of Texas at Austin in 1998, where he is currently working toward the Ph.D. degree in computer sciences.

He is currently a Research Assistant with the Network Protocol Design Lab, University of Texas at Austin. His research interests include network security, network protocol design and verification,

and distributed systems.



**Tommy M. McGuire** received the B.A. and M.A. degrees in computer sciences from the University of Texas at Austin in 1990 and 1994, respectively.

He has worked as a System Administrator and Programmer for IBM and the Department of Computer Sciences at the University of Texas at Austin. His research interests are in network protocol design and implementation.