Hop Integrity in Computer Networks

M. G. Gouda M. El-Nozahy C.-T. Huang T. M. McGuire
Department of computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
{gouda, mootaz, chuang, mcguire} @cs.utexas.edu

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, and was not modified after it was sent and until it was received by p, and it 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. These three protocols are a frequent key exchange protocol, a weak integrity protocol, and a strong integrity check protocol. All three protocols are stateless, require small overhead, and do not constrain the network protocol in the routers in any way.

1. Introduction

Most computer networks suffers 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.

To counter this problem, we in this paper present protocols that can be used to provide hop integrity in any network. A network is said to provide hop integrity iff 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 inserted, modified, or replayed by an adversary that operates between p and q.

It is instructive to compare the hop integrity protocols in this paper with the IPsec protocols discussed in [KA98], [MSST98], and [Orm98]. On one hand, the IPsec protocols provide richer classes of security services than those provided by the hop integrity protocols. First, the IPsec protocols can provide privacy and authentication that cannot be provided by the hop integrity protocols. Second, the IPsec protocols can secure the communication between any set of (possibly faraway) computers, whether hosts or routers, whereas the hop integrity protocols secure the communication only between pairs of adjacent routers. Third, the IPsec protocols can provide different degrees of security to different flows on-demand, whereas the hop integrity protocols provide the same degree of security to every message flow. On the other hand, the hop integrity protocols are simpler and more efficient than the IPsec protocols.

2. Hop Integrity Protocols

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.

i. *Detection of Message Insertion and Modification*:
Whenever router p receives a message m over the subnetwork connecting routers p and q, p can correctly determine which of the following two assertions holds for message m.

1. *Message m is neither inserted nor modified*: Message m was sent by router q and was not modified by an adversary after it was sent by q and before it was received by p.

2. *Message m is either inserted or modified*: Message m was sent by an adversary or was modified by an adversary after it was sent by q and before it was received by p.

ii. *Detection of Message Replay*:
Whenever router p receives a message m over the subnetwork connecting routers p and q, and determines that message m is neither inserted nor modified, then p can correctly determine 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 [Com88] and [Ste94], 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 *key exchange layer* and the *integrity check layer*. The key 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 key exchange layer is to allow adjacent routers to periodically generate and exchange (and so share) new security keys. The exchanged keys are made available to the integrity check layer which uses them to compute and verify the integrity check for every data message transmitted between adjacent routers.

Figure 1 shows the protocol stacks in two adjacent routers p and q. The key 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 insertion and 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 message insertion, modification, and replay.

In the next three sections, we describe in some detail the three protocols in the key exchange layer and the two versions of the integrity check layer. The first protocol between processes pe and qe is discussed in Section 3. The second protocol between processes pw and qw is discussed in Section 4. The third protocol between processes ps and qs is discussed in Section 5.

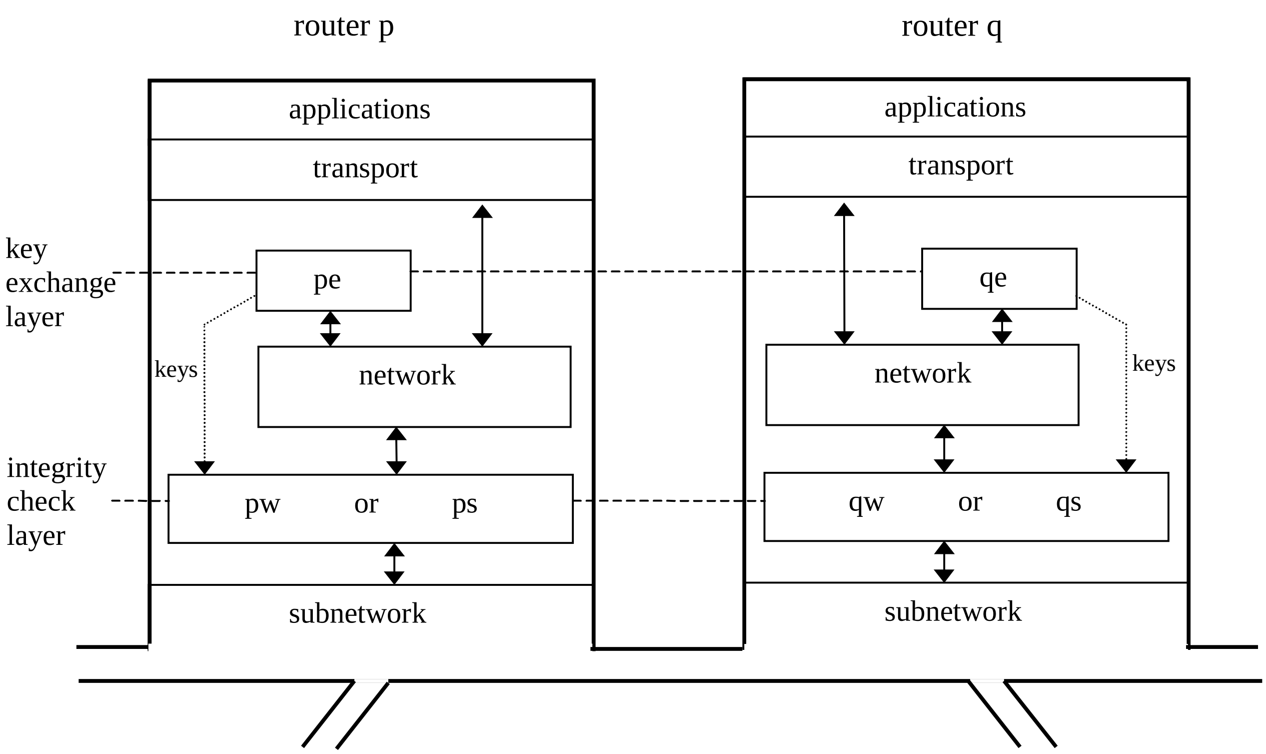

Figure 1. Protocol stack for achieving hop integrity.

We describe these three protocols using a variation of the Abstract Protocol Notation presented in [Gou98]. 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 the 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 py. 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> → <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 <selection> statement is of the form:

if <boolean expression> → <statement>
[] <boolean expression> → <statement>
…
[] <boolean expression> → <statement>
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, the actions in a protocol are executed one at a time. Second, an action is executed only when its guard is true. 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. Each sent message from px to qx remains in transit until it is eventually received by process qx or it is lost.

We assume that an adversary exists between processes px and qx. This adversary can modify the contents of messages while these messages are in transit (from px to qx or from qx to px). It can also insert new messages into transit. It can also insert old messages, which were sent and received some time ago, into transit; such messages are called *replayed messages*.

3. The Frequent Key Exchange Protocol

In the key exchange protocol, the two processes pe and qe maintain two shared keys kp and kq. Key kp is used by router p to compute the integrity check for each data message sent by p to router q. It is also used by router q to verify the integrity check for each data message received by q from router p. Similarly, key kq 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 keys kp and kq, processes pe and qe need to change these keys every te hours, for some chosen value te. Process pe is to initiate the change of key kq, and process qe is to initiate the change of key kp. Processes pe and qe have a shared key K that they use to encrypt and decrypt the messages that carry the new kp and kq between pe and qe.

For process pe to change key kq, the following four steps need to be performed. First, pe generates a new kq, and encrypts the concatenation of the old kq and the new kq using the shared key K, and sends the result in a rqst message to qe. Second, when qe receives the rqst message, it decrypts the message contents using the shared key K and obtains the old kq and the new kq. Then, qe checks that its current kq equals the old kq from the rqst message, and installs the new kq as its current kq, and sends a rply message containing the encryption of the new kq using the shared key K. Third, pe waits until it receives a rply message from qe containing the new kq encrypted using the shared key K. Receiving this rply message indicates that qe has received the rqst message and has accepted the new kq. Fourth, if pe sends the rqst message to qe but does not receive the rply message from qe for 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. Note that tr is an upper bound on the round trip time between pe and qe.

Note that the old key (along with the new key) is included in each rqst message and the new key is included in each rply message to ensure that if an adversary inserts, 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 kp and kq declared as follows.

```
var     kp :  integer
        kq :  array [0 .. 1] of integer
```

Similarly, process qe has an integer variable kq and an array variable kp.

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

kq[0] in pe = kq[1] in pe = kq in qe, and
kp[0] in qe = kp[1] in qe = kp in pe.

Process pe can be defined as follows. (Process qe can be defined in exactly the same way except that each occurrence of kp in pe is replaced by an occurrence of kq in qe, and each occurrence of kq[0] or kq[1] in pe is replaced by an occurrence of kp[0] or kp[1], respectively, in qe.)

```
process pe
inp     K     : integer                          {shared key between pe and qe}
        te    : integer                          {time between key exchanges}
        tr    : integer                          {upper bound on round trip time}
var     kp    : integer
        kq    : array [0 .. 1] of integer        {initially kq[0] = kq[1] = kq in qe}
        d, e  : integer
begin
        timeout   kq[0] = kq[1]  ∧  (te hours passed since rqst message sent last) →
              kq[1] := NEWKEY;
              e := NCR(K, (kq[0]; kq[1]));
              send rqst(e) to qe

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

[]      rcv rply(e) from qe →
```

```
                d := DCR(K, e);
                if kq[1] = d →  kq[0] := kq[1]
                [] kq[1] ≠ d →  {detect adversary} skip
                fi

[]      timeout   kq[0] ≠ kq[1]  ∧  (tr seconds passed since rqst message sent last) →
                e := NCR(K, (kq[0]; kq[1]));
                send rqst(e) to qe
end
```

The four actions of process pe use three functions NEWKEY, NCR, and DCR defined as follows. Function NEWKEY takes no arguments, and when invoked, it returns a fresh key that is different from any key 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(K, (kq[0]; kq[1]))

causes the concatenation of kq[0] and kq[1] to be encrypted using key K, 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(K, e)

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

(d, e) := DCR(K, e)

This statement indicates that the value of e is the encryption of the concatenation of two values (v_0; v_1) using key K. Thus, executing this statement causes e to be decrypted using key K, 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.

4. The 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 host s then transmitted through a sequence of adjacent routers r.1, r.2, …, r.n to a destination host d. When data(t) reaches the first router r.1, r.1 computes from the message text t a message digest d by executing the statement d := MD(t). Then r.1 encrypts d using the appropriate key provided by the key exchange process in r.1, and adds both d and its encryption e to the message before transmitting the resulting data(t, d, e) message to router r.2.

When the second router r.2 receives the data(t, d, e) message, r.2 encrypts d using the appropriate key provided by the key exchange process in r.2, and checks whether the result equals e. If they are equal, then r.2 concludes that the received message has been neither inserted nor 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 encrypting d using the appropriate key provided by the key exchange process in r.2 and storing the result in field e of the data(t, d, e) message.) Otherwise, r.2 concludes that the received message has been either inserted or modified and discards it.

When the last router r.n receives the data(t, d, e)message, first checks that MD(t) equals d. Then r.n encrypts d using the appropriate key provided by the key exchange process in r.n, and checks that the result equals e.

Note that the digest of a message is computed only once (when the message reaches the first router in its route) and is checked only once (when the message reaches the last router in its route). On the other hand, encryption of the message digest is computed and checked in every hop along the route. This is beneficial because computing and checking message digests are usually more expensive operations than computing and checking the encryptions of these digests (especially if a message digest consists of small number of bytes).

Process pw in the weak integrity check protocol has two inputs kp and kq 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, kp, and kq is replaced by an occurrence of q, p, qw, pw, kq, and kp, respectively.)

```
process pw
inp     kp          :  integer
        kq          :  array [0 .. 1] of integer
var     t, d, e     :  integer
begin
        rcv data(t, d, e) from qw →
            if NCR(kq[0], d) = e  ∨  NCR(kq[1], d) = e →    RTMSG
            [] NCR(kq[0], d) ≠ e  ∧  NCR(kq[1], d) ≠ e →
                                                  {detect adversary} skip
            fi

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

[]      true →
            {either p receives data(t) from an adjacent host or}
            {p generates the text t for the next data message}
            if NXT(t) = p →       {arrived} skip
            [] NXT(t) ≠ p →       d := MD(t);
                                  if NXT(t) = q →
                                              e := NCR(kp, d);
                                              send data(t, d, e) to qw
                                  [] NXT(t) ≠ q →
                                              {compute e as the encryption of d using the}
                                              {key for sending data to NXT(t); forward}
                                              {data(t, d, e) to router NXT(t)} skip
                                  fi
            fi
end
```

In the first action of process pw, if pw receives a data(t, d, e) message from qw while kq[0] ≠ kq[1], then pw cannot determine beforehand whether qw computed e from d using kq[0] or using kq[1]. In this case, pw needs to encrypt d using both kq[0] and kq[1], and compare the results of the two encryptions with e. If either encryption equals e, 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 text of a message, and computes a digest for that message. 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 →      if MD(t) = d → {accept message} skip
                     [] MD(t) ≠ d → {detect adversary} skip
                     fi
[] NXT(t) = q →      e := NCR(kp, d);
                     send data(t, d, e) to qw
[] NXT(t) ≠ p  ∧  NXT(t) ≠ q →
                     {compute e as the encryption of d using the}
                     {key for sending data to NXT(t); forward}
                     {data(t, d, e) to router NXT(t)} skip
fi
```

5. The Strong Integrity Check Protocol

The weak integrity check protocol in the previous section can detect message insertion and modification but not message replay. In this section, we discuss how to strengthen this protocol to make it detect message replay as well. A simple protocol for detecting message replay is to add consecutive sequence numbers to all sent messages, and to make the receiving process keep track of the expected sequence number of the next message to be received. If the receiving process receives a message whose number is not expected, the process declares that a message replay has been detected.

Unfortunately, the expected sequence number that the receiving process maintains in this protocol makes the resulting protocol “stateful”. To keep the resulting protocol “somewhat stateless”, the expected sequence number is maintained for at most T seconds in the receiving process then it becomes invalid. Then, the receiving process accepts the sequence number of the next received message as the new expected sequence number and maintains it for at most T seconds, and so on.

Next, we describe this soft sequence number protocol in some detail. Then, we discuss how to combine this protocol with the weak integrity check protocol in the previous section to obtain a strong integrity check protocol that can detect message insertion, modification, and replay.

The soft sequence number protocol consists of two processes named u and v. Process u sends data messages to process v at a rate of at most R messages per second. Each data message has a sequence number s in the range 0 .. 2N – 1. Process u can be defined as follows.

```
process u
inp      N, R :  integer                              {N in u = N in v}
var      s     :  0 .. 2N-1
begin
         timeout    (at least 1/R seconds passed since this action executed last) →
              send data(s) to v;
              s := (s + 1) mod 2N
end
```

Process v has three variables slast, s, and valid. Variable slast is used to store the sequence number of the last data message received by v. Variable s is used to store the sequence number of the data message currently being received by v. Variable valid is boolean, and its value is true iff the value of variable slast is valid. The initial value of valid is false. Process v can be defined as follows.

```
process v
inp     N, T      :  integer                          {N in u = N in v}
var     slast, s  :  0 .. 2N-1
        valid     :  boolean                          {initially false}
begin
        rcv data(s) from u →
            if ~  valid  ∨    within(s, slast, N) → {accept data} valid, slast := true, s
            []    valid  ∧  ~ within(s, slast, N) → {detect replay} skip
            fi

[]      timeout   (at most T seconds passed since this action executed last) →
            valid := false

end
```

Process v has a predicate within(s, slast, N) whose value is true iff ($1 \leq$ (s – slast) **mod** 2N $\leq$ N).

As discussed below, the correctness of this protocol is based on the assumption that the three integer inputs of the protocol, namely N, R, and T, satisfies the following condition:

$$N > R * T$$

Henceforth, we refer to this condition as the *correctness criteria* of the protocol.

To verify this protocol, we need to argue that the protocol satisfies the following two properties:

i. *Restraint*:
If process v receives two successive messages and neither message was a replay, and if v concludes that the first message is not a replay, then v also concludes that the second message is not a replay.

ii. *Detection*:
If process v receives two successive messages and the first message was a replay but the second message was not a replay, and if variable valid is true when process v receives the two messages, then v detects a replay on receiving the first message or on receiving the second message.

Note that according to the detection property, the protocol accurately detects the occurrence of replays but it cannot determine which of the received messages is a replay.

A proof that the protocol satisfies the restraint property is as follows. Assume that process v receives two successive messages data(s_0) and data(s_1) at times t_0 and t_1, respectively, and that neither data(s_0) nor data(s_1) was a replay. Assume also that process v concludes at t_0 that data(s_0) is not a replay. We need to prove that v concludes at t_1 that data(s_1) is not a replay.

There are two cases to consider, and we show that in each of these two cases, process v concludes at t_1 that data(s_1) is not a replay.

case 0: $(t_1 – t_0) > T$
In this case, valid = false at t_1 and process v concludes at t_1 that data(s_1) is not a replay.

case 1: $(t_1 – t_0) \leq T$
In this case, we have

$N > R * T$	{the correctness criteria}
$R * T \geq R * (t_1 – t_0)$	{because $T \geq t_1 – t_0$}
$R * (t_1 – t_0) \geq (s_1 – s_0)$ **mod** 2N	{because neither message is a replay}
$(s_1 – s_0)$ **mod** 2N > 0	{because $t_1 – t_0 \leq T$ and}
	{neither message is a replay}

From these facts, we have $N \geq (s_1 – s_0)$ **mod** $2N > 0$. In other words, within(s_1, s_0, N) is true at t_1. Moreover, slast = s_0 at t_1 because process v concludes at t_0 that data(s_0) is not a replay. Thus, process v concludes at t_1 that data(s_1) is not a replay.

A proof that the protocol satisfies the detection property is as follows. Assume that process v receives two successive messages data(s_0) and data(s_1) at times t_0 and t_1, respectively, and that data(s_0) was a replay and data(s_1) was not a replay. Assume also that valid = true at t_1. We need to prove that v will detect a replay at t_0 or t_1. Because data(s_0) was a replay and data(s_1) was not a replay, we have slast = $(s_1 – 1)$ **mod** 2N immediately before t_0. If process v does not detect a replay at t_0, then because valid = true at t_0, we conclude that within(s_0, $(s_1 – 1)$ **mod** 2N, N) and slast = s_0 at t_0 and during the period from t_0 till immediately before t_1. Therefore, ~ within(s_1, slast, N) at t_1, and because valid = true at t_1, process v detects a replay at t_1.

Processes u and v of the soft sequence number protocol 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 while pw exchanges with qw messages of the form data(t, d, e), ps exchanges with qs messages of the form data(t, d, e, s), where s is the message sequence number computed according to the soft sequence number protocol. Another difference between pw and ps is that ps has a buffer, named the send-q buffer, for storing all the messages that ps needs to send to qs. Process ps sends the messages from this buffer to process qs at a rate of no more than R messages per second, as dictated by the soft sequence number protocol. Process ps in the strong integrity check protocol can be defined as follows.

```
process ps
inp     kp              :  integer
        kq              :  array [0 .. 1] of integer
        N, R, T         :  integer
var     t, d, e         :  integer
        slast, s, snxt  :  0 .. 2N –1
        valid           :  boolean            {initially false}
        buff            :  integer            {# messages in send-q buffer; initially 0}

begin
        rcv data(t, d, e, s) from qs →
           if NCR(kq[0], d) = e ∨ NCR(kq[1], d) = e →
                  if ~  valid ∨     within(s, slast, N) →  {accept message}
                                    valid, slast := true, s;  RTMSG
                  []    valid ∧  ~ within(s, slast, N) →
                                   {detect replay} skip
                  fi
           [] NCR(kq[0], d) ≠ e  ∧  NCR(kq[1], d) ≠ e →
                  {detect adversary} skip
```

```
          fi

[]      true →
          {p receives a data(t, d, e, s) 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}
          if NXT(t) = p →     {arrived} skip
          []NXT(t) ≠ p →      d := MD(t);
                              if NXT(t) = q →     e := NCR(kp, d);
                                                  {store data(t, d, e) in send-q buffer}
                                                  buff := buff + 1
                              [] NXT(t) ≠ q →
                                  {compute e as the encryption of d using}
                                  {key for sending data to router NXT(t);}
                                  {compute next soft sequence number s;}
                                  {forward data(t, d, e, s) to router NXT(t)}
                                  skip
                              fi
          fi

[]      timeout   buff > 0  ∧
                  (at least 1/R seconds passed since this action executed last) →
          {get the next data(t, d, e) from send-q buffer}
          send data(t, d, e, snxt) to qs;
          snxt := (snxt + 1) mod 2N;
          buff := buff – 1

[]      timeout   (at most T seconds passed since this action executed last) →
          valid := false
end
```

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

```
if NXT(t) = p →     if MD(t) = d → {accept message} skip
                    [] MD(t) ≠ d → {detect adversary} skip
                    fi
[] NXT(t) = q →     e := NCR(kp, d);
                    {store data(t, d, e) in send-q buffer}
                    buff := buff + 1
[] NXT(t) ≠ p  ∧  NXT(t) ≠ q →
                    {compute e as the encryption of d using}
                    {key for sending data to router NXT(t);}
                    {compute next soft sequence number s;}
                    {forward data(t, d, e, s) to router NXT(t)}
                    skip
fi
```

6. Implementation Considerations

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

There are three inputs in the frequent key exchange protocol in Section 3. They are K, te, and tr. Input K is a shared key between two adjacent routers. This is a long-term key that remains fixed for long periods (say one to three months), and can be changed only off-line and only by the system administrators of the two routers. Thus, this key should consist of a relatively large number of bytes, say 128 bytes. There are no special requirements for the encryption and decryption functions that use this key in the key exchange protocol.

Input te is the time period between two successive key exchanges between pe and qe. This time period should be small so that an adversary does not have enough time to deduce the keys kp and kq used in computing the integrity checks of data messages. It should also be large so that the overhead that results from key exchanges is reduced. An acceptable value of 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 seconds.

Next, we consider input kp and the two functions NCR and MD used in the integrity check protocols in Sections 4 and 5. Input kp is a short-term key that is updated every 4 hours. Thus, this key should consist of a relatively small number of bytes, say 8 bytes. The encryption function NCR that use this key in the integrity check protocols needs to be fast because it is computed twice in each hop of every data message. A good candidate for this function is RC4 described in [Sch94]. We estimate that applying this function to encrypt a 4-byte message digest using an 8-byte key takes about 1.2 microseconds. 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 [Riv92] 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.

Consider the three inputs R, T, and N of the strong integrity check protocol in Section 5. Input R is the maximum rate at which one router sends data messages to an adjacent router. If two adjacent routers are connected by a high speed Ethernet, then R is about 8000 messages per second. Input T is the maximum period for keeping variable valid true. We choose T to be 4 seconds. From the correctness criteria of the protocol, we have N > 32,000. Thus, the soft sequence numbers of data messages should be in the range 0..64,000, and each soft sequence number occupies 2 bytes.

In summary, the message overhead of the strong integrity check protocol is about 10 bytes per data message: 4 bytes for storing the message digest, 4 bytes for storing the encrypted message digest, and 2 bytes for storing the soft sequence number of the message.

7. 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 whenever the received message was indeed sent by q or was inserted, modified, or replayed by an adversary that operates between p and q.

We also presented three protocols that can be used to make any computer network provide hop integrity. These three protocols are a frequent key exchange protocol (in Section 3), a weak integrity check protocol (in Section 4), and a strong integrity check protocol (in Section 5).

These three protocols have several novel features that make them correct and efficient. First, whenever the key exchange protocol attempts to change a key, it maintains both the old key and the new key until it is certain that the integrity check of any future message will not be computed using the old key. Second, the integrity check protocol computes the digest only when the message reaches its first router and checks this digest only when the message reaches its last router. Third, the strong integrity check protocol uses soft sequence numbers to keep the protocol stateless.

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

References

[Com88] Comer, D. E., *Internetworking with TCP/IP:* Vol. I: *Principles, Protocols, and Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Gou98] Gouda, M., *Elements of Network Protocol Design*, Wiley, 1998.

[KA98] Kent, S., Atkinson, R., "*Security Architecture for the Internet Protocol*", RFC 2401, November 1998.

[MSST98] Maughan, D., Schertler, M., Schneider, M., and J. Turner, "*Internet Security Association and Key Management Protocol (ISAKMP)*", RFC 2408, November 1998.

[Orm98] Orman, H., "*The OAKLEY Key Determination Protocol*", RFC 2412, November 1998.

[Riv92] Rivest, R. L., "*The MD5 Message Digest Algorithm*", RFC 1320, 1992.

[Sch94] Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Wiley, New York, 1994.

[Ste94] Stevens, W. R., *TCP/IP Illustrated*, Vol. I: *The Protocols*, Prentice-Hall, Englewood Cliffs, NJ, 1994.