

# Emergence of Hop Integrity in Computer Networks with Algorithms and Description of Protocols

Ashis Saklani

Assistant Professor Dept. Of Computer Science H.N.B Garhwal Central University Srinagar Garhwal  
Uttarakhand

## 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. 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.

**KEYWORDS** : Security, Hop Integrity, Secret Exchange Protocol, Hop Integrity Protocol, Weak Integrity Protocol.

## I. INTRODUCTION

Most 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. Two well-known examples of such attacks in networks that support the Internet Protocol (or IP, for short) and the Transmission Control Protocol (or TCP, for short) are as follows.

## II. SMURF ATTACK:

In an IP network, any computer can send a "ping" message to any other computer which replies by sending back a "pong" message to the first computer as required by Internet Control Message Protocol (or ICMP, for short) [14]. 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 did not request.

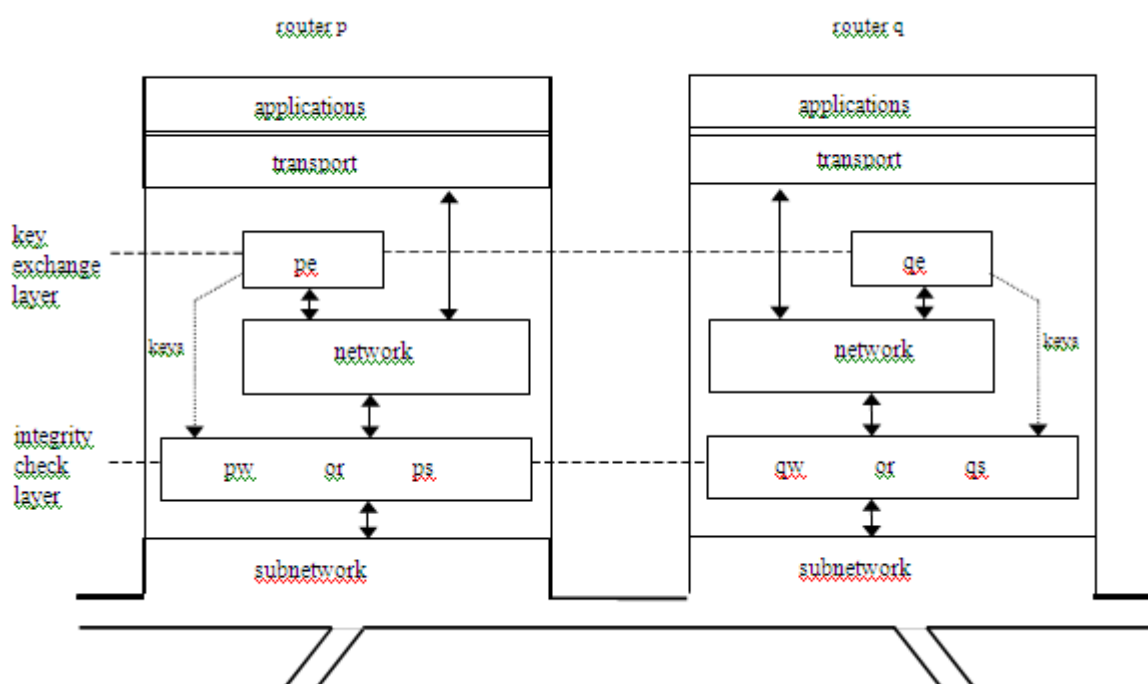
## III. SYN ATTACK:

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 [1, 18].

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 [7]) 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 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 [2, 11, 17], ingress filtering [4], and IPsec [8]. In secure routing, for example [2], [11], and [17], 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 [4], each router on the network boundary checks whether the recorded source in each received message is consistent with where the router received the message from. 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 hop integrity protocol suite in this paper and the IPsec protocol suite presented in [8], [9], [10], [12], and [13] 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. Next, we describe the concept of hop integrity in some detail. 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 Modification: 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  $p$ . ii. Detection of Message Replay: Whenever router  $p$  receives a message  $m$  over the subnetwork connecting routers  $p$  and  $q$ , and determines that message  $m$  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 [3] and [16], 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



**Figure 1. Protocol stack for achieving hop integrity.**

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) new secrets. The exchanged secrets are made available to the integrity check layer which uses them to compute and verify the integrity check for every data message transmitted between the adjacent routers. Figure 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 (which are described in the Introduction). 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 [4]) 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 can not 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 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. These three protocols are described using a variation of the Abstract Protocol Notation presented in [5]. 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 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>[] [] [] [] [] <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:

```

<boolean expression>   → <statement>
...
if <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, 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 <m.1; m.2; [] ; m.n> 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  $p_x$  and  $q_x$  is affected by at most one adversary action. 3. The Secret Exchange Protocol In the secret exchange protocol, the two processes  $p_e$  and  $q_e$  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  $p_e$  and  $q_e$  need to change these secrets periodically, say every  $t_e$  hours, for some chosen value  $t_e$ . Process  $p_e$  is to initiate the change of secret  $sq$ , and process  $q_e$  is to initiate the change of secret  $sp$ . Processes  $p_e$  and  $q_e$  each has a public key and a private key that they use to encrypt and decrypt the messages that carry the new secrets between  $p_e$  and  $q_e$ . 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

```

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

keys of process  $p_e$  are named  $B_p$  and  $R_p$  respectively; similarly the public and private keys of process  $q_e$  are named  $B_q$  and  $R_q$  respectively. For process  $p_e$  to change secret  $sq$ , the following four steps need to be performed. First,  $p_e$  generates a new  $sq$ , and encrypts the concatenation of the old  $sq$  and the new  $sq$  using  $q_e$ 's public key  $B_q$ , and sends the result in a  $rqst$  message to  $q_e$ . Second, when  $q_e$  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,  $q_e$  checks that its current  $sq$  equals the old  $sq$  from the  $rqst$  message, and installs the new  $sq$  as its current  $sq$ , and sends a  $rply$  message containing the encryption of the new  $sq$  using  $p_e$ 's public key  $B_p$ . Third,  $p_e$  waits until it receives a  $rply$  message from  $q_e$  containing the new  $sq$  encrypted using  $B_p$ . Receiving this  $rply$  message indicates that  $q_e$  has received the  $rqst$  message and has accepted the new  $sq$ . Fourth, if  $p_e$  sends the  $rqst$  message to  $q_e$  but does not receive the  $rply$  message from  $q_e$  for some  $t_r$  seconds, indicating that either the  $rqst$  message or the  $rply$  message was lost before it was received, then  $p_e$  resends the  $rqst$  message to  $q_e$ . Thus  $t_r$  is an upper bound on the round trip time between  $p_e$  and  $q_e$ . 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  $p_e$  or  $q_e$ ). Process  $p_e$  has two variables  $sp$  and  $sq$  declared as follows.

Similarly, process  $q_e$  has an integer variable  $sq$  and an array variable  $sp$ . In process  $p_e$ , 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  $p_e$  has generated and sent the new secret  $sq$ , and that  $q_e$  may not have received it yet. The assertion  $sq[0] = sq[1]$  indicates that  $q_e$  has already received and accepted the new secret  $sq$ . Initially,  $sq[0]$  in  $p_e = sq[1]$  in  $q_e$ , and  $sp[0]$  in  $q_e = sp[1]$  in  $p_e$ . Process  $p_e$  can be defined as follows. (Process  $q_e$  can be defined in the same way except that each occurrence of  $R_p$  in  $p_e$  is replaced by an occurrence of  $R_q$  in  $q_e$ , each occurrence of  $B_q$  in  $p_e$  is replaced by an occurrence of  $B_p$  in  $q_e$ , each occurrence of  $sp$  in  $p_e$  is replaced by an occurrence of  $sq$  in  $q_e$ , and each occurrence of  $sq[0]$  or  $sq[1]$  in  $p_e$  is replaced by an occurrence of  $sp[0]$  or  $sp[1]$ , respectively, in  $q_e$ .) process  $p_e$

```

if  $sq[1] = d \rightarrow sq[0] := sq[1]$ 
  []  $sq[1] = d \rightarrow \{detect\ adversary\}$  skip
fi
timeout
   $sq[0] \neq sq[1] \wedge$ 
  ( $t_r$  seconds passed since  $rqst$  message sent last)  $\rightarrow$ 
     $e := NCR(B_q, (sq[0]; sq[1]));$ 
    send  $rqst(e)$  to  $q_e$ 
    -----
    
```

end The four actions of process  $p_e$  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  $e := NCR(B_q, (sq[0]; sq[1]))$  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  $d := \text{DCR}(R p, e)$

```
inp Rp : integer {private key of p}
```

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  $R p$ , 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  $e$ . A proof of the correctness of the secret exchange protocol is presented in the full version of the paper [6].

4. The 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:  $d := \text{MD}(t; \text{scr})$  where  $\text{MD}$  is the message digest function,  $(t; \text{scr})$  is the concatenation of the message text  $t$  and the shared secret  $\text{scr}$  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  $\text{pw}$  in the weak integrity protocol has two inputs  $\text{sp}$  and  $\text{sq}$  that  $\text{pw}$  reads but never updates. These two inputs in process  $\text{pw}$  are also variables in process  $\text{pe}$ , and  $\text{pe}$  updates them periodically, as discussed in the previous section. Process  $\text{pw}$  can be defined as follows. (Process  $\text{qw}$  is defined in the same way except that each occurrence of  $p, q, \text{pw}, \text{qw}, \text{sp}$ , and  $\text{sq}$  is replaced by an occurrence of  $q, p, \text{qw}, \text{pw}, \text{sq}$ , and  $\text{sp}$ , respectively.)

```
process pw
```

```
inp sp      : integer
   sq      : array [0 .. 1] of integer
var t, d    : integer
begin
  rcv data(t, d) from qw →
    if MD(t; sq[0]) = d ∨ MD(t; sq[1]) = d →
      {defined later} RTMSG
    [] MD(t; sq[0]) ≠ d ∧ MD(t; sq[1]) ≠ d →
      {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
```

end In the first action of process  $\text{pw}$ , if  $\text{pw}$  receives a  $\text{data}(t, d)$  message from  $\text{qw}$  while  $\text{sq}[0] \neq \text{sq}[1]$ , then  $\text{pw}$  cannot determine beforehand whether  $\text{qw}$  computed  $d$  using  $\text{sq}[0]$  or using  $\text{sq}[1]$ . In this case,  $\text{pw}$  needs to compute two message digests using both  $\text{sq}[0]$  and  $\text{sq}[1]$  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
    
```

A proof of the correctness of the weak integrity protocol is presented in the full version of the paper [6].

**5. The Strong Integrity Protocol** The weak integrity protocol in the previous section can detect message modification but not message replay. In this section, we discuss how to strengthen this protocol to make it detect message replay as well. 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 (in the previous section) 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. When process  $v$  receives a data(s) message,  $v$  compares its variable  $exp$  with  $s$ . If  $exp \leq s$ , 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  $exp \leq nxt$  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 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  $cmax$ , whose values are in the range  $0..N-1$ . When process  $v$  receives a data(s) message,  $v$  compares the values of  $c$  and  $cmax$ . If  $c \leq cmax$ , then process  $v$  increments  $c$  by one (mod  $N$ ) and proceeds as before (namely either accepts the data(s) message if  $exp \leq s$ , or discards the message if  $exp > s$ ). Otherwise,  $v$  accepts the message, assigns  $c$  the value 0, and assigns  $cmax$  a random integer in the range  $0..N-1$ .

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.

```

var nxt : integer      {sequence number of
                       {next sent message}

begin
  true → send data(nxt) to v;  nxt := nxt + 1
end

process v
inp  N  : integer
var  s  : integer      {sequence number of
                       {received message}
    exp : integer      {sequence number of
                       {next expected message}
    c, cmax : 0..N-1

begin
  rcv data(s) from u →
    if s < exp ∧ c = cmax →
      {reject message; report an adversary}
      c := (c + 1) mod N
    [] s ≥ exp ∨ c = cmax →
      {accept message}
      exp := s + 1;
      if c = cmax → c := (c + 1) mod N
      [] c = cmax →
        c := 0;
        cmax := RANDOM(0, N - 1)
      fi
    fi
end

```

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  sp      : integer
    sq      : array [0 .. 1] of integer
    N       : integer
var  s, t, d : integer
    exp, nxt : integer
    c, cmax  : 0..N-1

begin
  rcv data(s, t, d) from qs →
    if MD(s; t; sq[0]) = d ∨ MD(s; t; sq[1]) = d →
      if s < exp ∧ c = cmax →
        {reject message and}
        {report an adversary}
        c := (c + 1) mod N
      [] s ≥ exp ∨ c = cmax →
        {accept message}
        exp := s + 1;
        if c = cmax → c := (c + 1) mod N
        [] c = cmax →
          c := 0;
          cmax := RANDOM(0, N - 1)
        fi
      fi
    [] MD(s; t; sq[0]) ≠ d ∧ MD(s; t; sq[1]) ≠ d →
      {report an adversary} skip
    fi
  [] true →

```



```

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

```

A proof of the correctness of the strong integrity protocol is presented in the full version of the paper [6].

#### IV. 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 four inputs in the secret exchange protocol in Section 3. They are  $R_p$ ,  $B_q$ ,  $t_e$  and  $t_r$ . 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 1024 bytes each. There are no special requirements for the encryption and decryption functions that use these keys in the secret exchange protocol. Input  $t_e$  is the time period between two successive secret exchanges between  $p_e$  and  $q_e$ . This time period should be small so that an adversary does not have enough time to deduce the secrets  $s_p$  and  $s_q$  used in computing the integrity checks of data messages. It should also be large so that the overhead that results from the secret exchanges is reduced. An acceptable value for  $t_e$  is around 4 hours. Input  $t_r$  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  $t_r$  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  $t_r$  is around 4 seconds. Next, we consider the two inputs  $s_p$  and  $s_q$  and function MD used in the integrity protocols in Sections 4 and 5. Inputs  $s_p$  and  $s_q$  are short-lived secrets that are updated every 4 hours. Thus, this key should consist of a relatively small number of bytes, say 8 bytes. Function MD is used to compute the digest of a data message. Function MD is computed in two steps as follows. First, the standard function MD5 [15] 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. As discussed in Section 5, input  $N$  needs to be much larger than 1. For example,  $N$  can be chosen 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 that an adversary who replays an old message will be detected is 99 percent. The message overhead of the strong integrity protocol is about 8 bytes per data message: 4 bytes for storing the message digest, and 4 bytes for storing the soft sequence number of the message.

## V. CONCLUDING REMARKS IN THIS PAPER,

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

## REFERENCES

- [1] TCP SYN Flooding and IP Spoofing Attacks", CERT Advisory CA-96.21, available at <http://www.cert.org/>.
- [2] S. Cheung, "An Efficient Message Authentication Scheme for Link State Routing", Proceedings of the 13th Annual Computer Security Applications Conference, San Diego, California, December 1997, pp. 90-98.
- [3] D. E. Comer, Internetworking with TCP/IP: Vol. I: Principles, Protocols, and Architecture, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [4] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", RFC 2267, January 1998. [5] M. G. Gouda, Elements of Network Protocol Design, John Wiley & Sons, New York, NY, 1998. [6] M. G. Gouda, E. N. Elnozahy, C.-T. Huang, T. M. McGuire, "Hop Integrity in Computer Networks", Technical Report TR-00-19, Department of Computer Sciences, The University of Texas at Austin, August 2000.
- [5] L. Joncheray, "A Simple Active Attack Against TCP", Proceedings of the 5th USENIX UNIX Security Symposium, 1995, pp. 7-19.
- [6] [8] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.
- [7] S. Kent and R. Atkinson, "IP Authentication Header", RFC 2402, November 1998.
- [8] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)", RFC 2406, November 1998. [11] S. Murphy and M. Badger, "Digital Signature Protection of the OSPF Routing Protocol", Proceedings of the 1996 Internet Society Symposium on Network and Distributed Systems Security, San Diego, California, February 1996. [12] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)", RFC 2408, November 1998.
- [9] H. Orman, "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.
- [10] J. Postel, "Internet Control Message Protocol", RFC 792, September 1981. [15] R. L. Rivest, "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.
- [11] [W. R. Stevens, TCP/IP Illustrated, Vol. I: The Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [12] B. Smith, S. Murthy, and J. J. Garcia-Luna-Aceves, "Securing Distance Vector Routing Protocols", Proceedings of the 1997 Internet Society Symposium on Network and Distributed Systems Security, San Diego, California, February 1997.
- [13] M. de Vivo, G. de Vivo, and G. Isern, "Internet Security Attacks at the Basic Levels", Operating Systems Review, Vol. 32, No. 2, SIGOPS, ACM, April 1998.

## BIOGRAPHY

**Ashis Saklani** :- Received M.tech in Computer Science from Karnataka University, and MCA from Graphic Era Institute of Technology (Now Graphic Era University) Dehradun. He is a CISCO And Microsoft Certified Professional. He worked as a VPN Support Engineer at HCL Infinet, Noida. Presently he is working as Assistant Professor, at H.N.B Garhwal Central University, Srinagar, Uttarakhand, India.