On the Security and Vulnerability of PING*

Mohamed G. Gouda[1], Chin-Tser Huang[1], and Anish Arora[2]

[1] Department of Computer Sciences, The University of Texas at Austin,
Austin, TX 78712-1188, U.S.A.
{gouda, chuang}@cs.utexas.edu

[2] Department of Computer and Information Science, The Ohio State University,
Columbus, OH 43210-1277, U.S.A.
anish@cis.ohio-state.edu

Abstract. We present a formal specification of the PING protocol, and use three concepts of convergence theory, namely closure, convergence, and protection, to show that this protocol is secure against weak adversaries (and insecure against strong ones). We then argue that despite the security of PING against weak adversaries, the natural vulnerability of this protocol (or of any other protocol for that matter) can be exploited by a weak adversary to launch a denial of service attack against any computer that hosts the protocol. Finally, we discuss three mechanisms, namely ingress filtering, hop integrity, and soft firewalls that can be used to prevent denial of service attacks in the Internet.

1 Introduction

Recent intrusion attacks on the Internet, the so called denial of service attacks, have been through the well-known PING protocol [3]. These repeated attacks raise the following two important questions: Is the PING protocol secure? Can the PING protocol be made secure enough to prevent denial of service attacks? In this paper, we use several concepts of convergence theory to answer these two important questions. In particular, we use the three concepts of closure, convergence, and protection to show that the PING protocol is in fact secure. We also argue that the PING protocol cannot be secure enough to prevent denial of service attacks. An adversary can always exploit the natural vulnerability of any (possibly secure) protocol to launch a denial of service attack against any computer that hosts this protocol. We briefly discuss several techniques that can be used to safeguard the computers in any network against denial of service attacks on that network.

The PING protocol in this paper is specified using a version of the Abstract Protocol Notation presented in [6]. Using this notation, each process in a protocol is defined by a set of constants, a set of variables, a set of parameters, and a set of actions. For example, in a protocol consisting of two processes p and q and two channels (one from p to q and one from q to p), process p can be defined as follows.

* This work is supported in part by the grant OSU-RF-01-C-1901 from DARPA.

```
process p
const   <name of constant>  :   <type of constant>
        ...
        <name of constant>  :   <type of constant>
var     <name of variable>  :   <type of variable>
        ...
        <name of variable>  :   <type of variable>
par     <name of parameter> :   <type of parameter>
        ...
        <name of parameter> :   <type of parameter>
begin
        <action>
[]      <action>
        ...
[]      <action>
end
```

The constants of process p have fixed values. The variables of process p can be read and updated by the actions of process p. Comments can be added anywhere in a process definition; each comment is placed between the two brackets { and }.

Each **<action>** of process p is of the form:

```
<guard>  ->  <statement>
```

The guard of an action of p is one of the following three forms: a boolean expression over the constants and variables of p, a receive guard of the form **rcv <message> from q**, or a timeout guard that contains a boolean expression over the constants and variables of every process and the contents of the two channels in the protocol.

Each parameter declared in a process is used to write a finite set of actions as one action, with one action for each possible value of the parameter. For example, if process p has the following variable x and parameter i,

```
var     x :   0 .. n-1
par     i :   0 .. n-1
```

then the following action in process p

```
        x = i   ->   x := x + i
```

is a shorthand notation for the following n actions.

```
        x = 0   ->   x := x + 0
[]      ...
[]      x = n-1 ->   x := x + n-1
```

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.

The `<statement>` of an action of process p is a sequence of `<skip>`, `<assignment>`, `<send>`, `<selection>`, or `<iteration>` statements of the following forms:

```
<skip>         :    skip
<send>         :    send <message> to q
<assignment>   :    <variable in p>          :=    <expression>
<selection>    :    if <boolean expression>  ->    <statement>
                    ...
                    [] <boolean expression>  ->    <statement>
                    fi
<iteration>    :    do <boolean expression>  ->    <statement>
                    od
```

Executing an action of process p can cause a message to be sent to process q. There are two channels between the two processes: one is from p to q, and the other is from q to p. Each sent message from p to q remains in the channel from p to q until it is eventually received by process q or is lost. Messages that reside simultaneously in a channel form a set and so they are received or lost, one at a time, in any order and not necessarily in the same order in which they were sent.

2 The PING Protocol

The PING protocol (which stands for the Packet Internet Groper protocol) allows a computer in the Internet to test whether a specified computer in the Internet is up [9]. The test is carried out as follows. First, the testing computer p sends several echo request messages to the computer q[i] being tested. Second, the testing computer p waits to receive one or more echo reply messages from computer q[i]. Third, if the testing computer p receives one or more echo reply messages from computer q[i], p concludes that computer q[i] is up. On the other hand, if the testing computer p receives no echo reply message from computer q[i], p concludes that q[i] may not be up. The conclusion in this case is not certain because it is possible (though unlikely) that all the echo request messages sent from p to q[i] or the corresponding echo reply messages from q[i] to p are lost during transmission.

The testing computer p stores the test results in a local variable array named up that is declared as follows.

```
var     up :     array [0 .. n-1] of boolean
```

Note that n is the number of computers being tested. If at the end of a session of the PING protocol, up[i] = true in computer p, then computer q[i] was up sometime during that session. On the other hand, if at the end of a session, up[i] = false in computer p, then no firm conclusions can be reached (but it is likely that q[i] was down during that session).

For computer p to ensure that a received echo reply message from a computer q[i] corresponds to the echo request message that p has sent earlier to q[i], p adds a random identifier id[i] to all the echo request messages that p sends to q[i] in a session of the PING protocol. When a computer q[i] receives an erqst(id[i]) message from computer p, q[i] replies by sending an erply(id[i]) message to p. When computer p receives an erply(id[i]) message, p checks whether id[i] is the random identifier of the current protocol session with q[i]. If so, p accepts the message and assigns the corresponding up[i] element the value true. Otherwise, p discards the message.

The process of the testing computer p in the PING protocol is defined as follows.

```
process p
const   n, idmax, cmax
var     up :    array [0 .. n-1] of boolean
        wait :  array [0 .. n-1] of boolean
        id :    array [0 .. n-1] of 0 .. idmax
x :     0 .. idmax
        c :     0 .. cmax
par     i :     0 .. n-1
begin
        ~wait[i] ->     up[i] := false;
                        id[i] := random;
                        c := 0;
                        do (c < cmax) ->
                                send erqst(id[i]) to q[i];
                                c := c + 1;
                        od;
                        wait[i] := true

[]      rcv erply(x) from q[i] ->
                        if   wait[i]  ^  x = id[i]  ->
                                up[i] := true
                        [] ~ wait[i]  v  x <> id[i] ->
                                {discard erply} skip
                        fi

[]      timeout  ( wait[i] ^
          erqst(id[i])#ch.p.q[i] + erply(id[i])#ch.q[i].p = 0 ) ->
                        wait[i] := false
end
```

Process p has three actions. In the first action, p recognizes that it is no longer waiting for any erply messages from its last session of the protocol with q[i], and starts its next session with q[i]. Process p starts the next session with q[i] by selecting a new random identifier id[i] for the new session and sending cmax many erqst(id[i]) messages to process q[i]. In the second action, process p receives an erply(id[i]) message from any process q[i] and decides whether to accept the message and assign up[i] the value true, or discard the message. In the third action, process p recognizes that a long time has passed since p has sent the erqst(id[i]) messages in the current session, and so the number of erqst(id[i]) messages in the channel from p to q[i], denoted erqst(id[i])#ch.p.q[i], is zero and the number of erply(id[i]) messages in the channel from q[i] to p, denoted erply(id[i])#ch.q[i].p, is also zero. In this case, p terminates the current session with q[i] by assigning variable wait[i] the value false.

The process for any computer q[i] being tested is defined as follows.

```
process q[i: 0 .. n-1]
const   n, idmax
input   up :    boolean
var     x :     0 .. idmax
begin
        rcv erqst(x) from p ->
                        if   up ->    send erply(x) to p
                        [] ~ up ->    skip
                        fi

end
```

Process q[i] has a boolean input named up that describes the current state of q[i]. Clearly, the value of input up can change over time to reflect the change in the state of q[i]. Nevertheless, to keep our analysis of the PING protocol simple, we assume that the value of input up remains constant.

Process q[i] has only one action. In this action, q[i] receives an erqst(x) message from p and either sends an erply(x) message to p (if input up in q[i] is true), or discards the received erqst(x) message (if input up in q[i] is false).

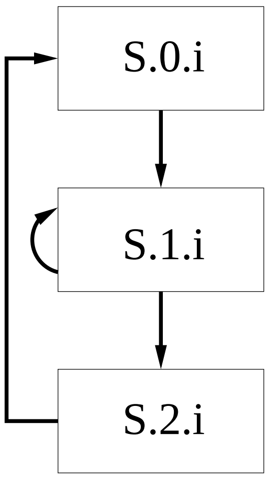

Fig. 1. State transition diagram of PING.

The state transition diagram for the PING protocol is shown in Figure 1. There are three nodes in this diagram. Each of the three nodes represents a set of states of the PING protocol. Each node v is labeled with a state predicate S.v.i, whose value is true at every state represented by node v. The three state predicates in the state transition diagram, namely S.0.i, S.1.i, and S.2.i, are defined as follows

$$\begin{array}{lllllllllll} \mathrm{S.0.i} = & \sim \mathrm{wait[i]} & \wedge & \mathrm{B.i} = 0 & \wedge & \mathrm{C.i} = 0 & & & & \\ \mathrm{S.1.i} = & \mathrm{wait[i]} & \wedge & \mathrm{B.i} > 0 & \wedge & \mathrm{C.i} = 0 & \wedge & \mathrm{X.i} & \wedge & \mathrm{Y.i} \\ \mathrm{S.2.i} = & \mathrm{wait[i]} & \wedge & \mathrm{B.i} = 0 & \wedge & \mathrm{C.i} = 0 & \wedge & & & \mathrm{Y.i} \end{array}$$

where

$\mathrm{B.i} = \mathrm{erqst(id[i])\#ch.p.q[i]} + \mathrm{erply(id[i])\#ch.q[i].p}$

B.i is the number of messages (in the two channels between p and q[i]) whose identifiers are equal to the value of variable id[i] in p.

$\mathrm{C.i} = \sum_{\mathrm{r} \neq \mathrm{id}} (\mathrm{erqst(r)\#ch.p.q[i]} + \mathrm{erply(r)\#ch.q[i].p})$

C.i is the number of messages (in the two channels between p and q[i]) whose identifiers are different from the value of variable id[i] in p.

$\mathrm{X.i} = (\ (\mathrm{erply(id[i])\#ch.q[i].p} > 0) \Rightarrow (\mathrm{up} = \mathrm{true\ in\ q[i]})\ )$

X.i states that if there is one or more erply(id[i]) message in the channel from a process q[i] to process p, then input up in q[i] has the value true.

$\mathrm{Y.i} = (\ (\mathrm{up[i]} = \mathrm{true\ in\ p}) \Rightarrow (\mathrm{up} = \mathrm{true\ in\ q[i]})\ )$

Y.i states that if an element up[i] in process p has the value true, then input up in process q[i] has the value true.

The directed edges in the state transition diagram in Figure 1 represent executions of actions in processes p and q[i]. The directed edge from node S.0.i to node S.1.i represents an execution of the first action in process p. The directed edge from node S.1.i to node S.2.i represents an execution of the second action in process p. The directed edge from node S.2.i to node S.0.i represents execution of the third action in process p. The self-loop at node S.1.i represents any of the following: an execution of the action in process q[i], an execution of the second action in process p, and a loss of one message from one of the two channels between p and q[i].

3 The PING Adversary

For two reasons, the PING protocol is designed to overcome the activities of a weak adversary, rather than a strong adversary. First, this assumption keeps the PING protocol, which performs a basic task (of allowing any computer to test whether another computer in the Internet is up) both simple and efficient. Second, by disrupting the PING protocol, a strong adversary achieves very little: merely convincing one computer that another computer in the Internet is up when in fact that other computer is not up. It is not clear what does a strong adversary gain by such disruption, and so it is doubtful that a strong adversary will attempt to use its strength to disrupt the PING protocol.

The weak adversary considered in designing the PING protocol is one that can insert a finite number of erqst(x) messages into the channel from process p to a process q[i], and can insert a finite number of erply(x) messages into the channel from a process q[i] to process p. Identifiers of the messages inserted by this adversary at any instant are different from the identifiers of the current session and any future session of the PING protocol. Thus, if the adversary inserts an erply(x) message in the channel from process q[i] to process p at some time instant, and if p receives this message at some future instant, then p can still detect that x is different from the current value of variable id and discard the message. For convenience, we refer to every message whose identifier is different from the identifiers of the current session and every future session as an adversary message.

Figure 2 shows the state transition diagram that describes the activities of both the PING protocol and its (weak) adversary. Note that this diagram has three additional nodes over the diagram in Figure 1. These three nodes are labeled with the state predicates U.0.i, U.1.i, and U.2.i defined as follows.

$$\begin{array}{llllllllll} \text{U.0.i} = & \sim \text{wait[i]} & \wedge & \text{B.i} = 0 & \wedge & \text{C.i} > 0 & & & & \\ \text{U.1.i} = & \text{wait[i]} & \wedge & \text{B.i} > 0 & \wedge & \text{C.i} > 0 & \wedge & \text{X.i} & \wedge & \text{Y.i} \\ \text{U.2.i} = & \text{wait[i]} & \wedge & \text{B.i} = 0 & \wedge & \text{C.i} > 0 & \wedge & & & \text{Y.i} \end{array}$$

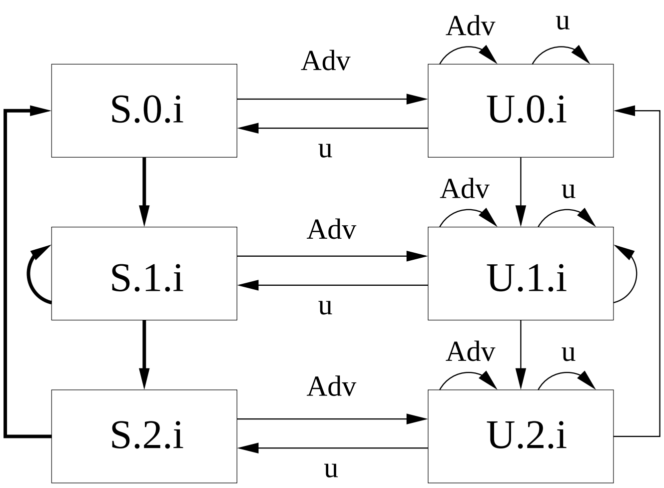

Fig. 2. State transition diagram of PING and adversary.

Note that B.i, C.i, X.i, and Y.i are defined above in Section 2. Note also that each predicate U.v.i is the same as the corresponding predicate S.v.i except that the conjunct C.i $= 0$ in S.v.i is replaced by the conjunct C.i > 0 in U.v.i. Thus, each U.v.i state is the same as a corresponding S.v.i state except that some adversary messages are inserted into some channels in the protocol.

In the state transition diagram in Figure 2, each edge labeled "Adv" represents an adversary action where one or more adversary messages are inserted into some channels in the protocol. Each edge or self-loop labeled "u" represents an execution of some protocol action where an adversary message is either received by a process q[i] (and another adversary message is sent from q[i] to p) or received by process p (and discarded).

Note that, despite the adversary involvement, the state predicate Y.i holds at every S.2.i state and every U.2.i state. Thus, Y.i holds at the end of every session between process p and process q[i] of the PING protocol.

4 Security of PING

In this section, we use three concepts of the theory of convergence [5], namely closure, convergence, and protection, to show that the PING protocol (presented in Section 2) is secure against the weak adversary (presented in Section 3). In general, to show that a protocol P is secure against an adversary D, one needs to partition the reachable states of P into safe states and unsafe states, then identify the critical variables of P (those that need to be protected from the actions of D), and show that the following three conditions hold ([1] and [7]).

i. **Closure**:
The set of safe states is closed under any execution of a P action, and the set of reachable states (i.e. the union of the safe state set and the unsafe state set) is closed under any execution of a P action or a D action.

ii. **Convergence**:
Starting from any unsafe state, any infinite execution of the P actions leads P to safe states.

iii. **Protection**:
If an execution of a P action starting at an unsafe state s changes the values of the critical variables of P from V to V', then there is a safe state s' such that the values of the critical variables in s equals to V, and execution of the same action starting at s changes the values of the critical variables of P from V to V'. (Note that this condition is a generalization of the corresponding condition in [7] which states that each execution of a P action starting at an unsafe state cannot change the values of the critical variables of P.)

Following this definition, the security of the PING protocol can be established by identifying the safe, unsafe, and reachable states of the protocol, then identifying its critical variables, and finally showing that the PING protocol satisfies the three conditions of closure, convergence, and protection.

The safe states of the PING protocol are specified by the state predicate S.i, where

S.i = S.0.i $\vee$ S.1.i $\vee$ S.2.i

The unsafe states of PING are specified by the state predicate U.i, where

U.i = U.0.i $\vee$ U.1.i $\vee$ U.2.i
Thus, the reachable states of the protocol are specified by the state predicate S.i $\vee$ U.i.

The PING protocol has only one critical variable, namely array up in process p. It remains now to show that the protocol satisfies the above three conditions of closure, convergence, and protection.

Satisfying the Closure Condition: From the state transition diagram in Figure 1, the set of safe states is closed under any execution of an action of the PING protocol. From the state transition diagram in Figure 2, the set of reachable states is closed under any execution of an action of the PING protocol or an action of the weak adversary.

Satisfying the Convergence Condition: Along any infinite execution of the actions of the PING protocol, the following three conditions hold.

i. No adversary message is added to the channel from process p to a process q[i].
ii. Each adversary message in a channel from process p to a process q[i] is eventually discarded (if up = false in q[i]), or replaced by another adversary message in the channel from q[i] to p (if up = true in q[i]).
iii. Each adversary message in a channel from a process q[i] to process p is eventually discarded.

Thus, starting from an unsafe U.i state, any infinite execution of the actions of the PING protocol leads the protocol to a safe S.i state where no channel has adversary messages.

Satisfying the Protection Condition: Assume that an execution of an action of the PING protocol starting at an unsafe state s changes the value of array up in process p. Then, the executed action is one where process p receives an erply(x) message from a process q[i], where x = id. Thus, the received erply(x) message is not an adversary message, and receiving this message causes the value of element up[i] in p to change from false to true. Let s' be the state that results from removing all the adversary messages that exist in state s. From the state transition diagram in Figure 2, state s' is a safe state. At state s', message erply(x) is still in the channel from process q[i] to process p. Thus, executing the action where process p receives the erply(x) message, starting at state s', causes the value of element up[i] in p to change from false to true. This completes our proof of the security of the PING protocol against the weak adversary.

It is also straightforward to show that the PING protocol is not secure against a strong adversary that can insert messages whose identifiers are equal to the identifier of the current session. Consider an unsafe protocol state s where the adversary has inserted an erply(x) message, where x = id, at the channel from a process q[i], whose input up is false, to process p. Executing the action where process p receives the inserted erply(x) message, starting at the unsafe state s, changes the value of element up[i] in p from false to true (even though the value of input up in q[i] is false). Because no action execution, starting at any safe state, changes the value of element up[i] in p from false to true (given that the value of input up in q[i] is false), then the protection condition does not hold. This argument shows that PING is not secure against a strong adversary.

5 Vulnerability of PING

Security of the PING protocol against the weak adversary is established in the last section by showing that the protocol satisfies the three conditions of closure, convergence, and protection. The closure condition states that the unsafe states (specified by U.i) are the furthest that the adversary can lead the protocol away from its safe states (specified by S.i). The convergence condition states that, when the adversary stops inserting adversary messages into the protocol channels, the PING protocol eventually converges from its current unsafe state to the safe states. The protection condition states that while the protocol is in its unsafe states (due to the influence of the adversary), the critical array "up" in process p is updated as if the protocol is in a safe state.

Despite the security of PING against its weak adversary, the weak adversary can exploit the natural vulnerability of the PING protocol to attack any computer that hosts PING as follows. The adversary inserts a very large number of adversary messages into the protocol channels. The protocol processes p and q[i: 0..n-1] become very busy processing and eventually discarding these messages. Thus, the computers that host these processes become very busy and unable to perform any other service. Such an attack is usually referred to as a denial of service attack.

It follows that denial of service attacks by weak adversaries can succeed by exploiting the natural vulnerability of any (even the most secure) protocols. The only way to prevent denial of service attacks is to prevent the adversary messages from reaching the protocol processes. In other words, the adversary messages need to be detected as such and discarded before they reach their destination processes. The question now is how to detect the adversary messages?

The answer to this question, in the case of the PING protocol, is straightforward. In the known denial of service attacks that exploit the PING protocol, the adversary inserts messages whose source addresses are wrong. Because the source of the inserted messages is the adversary itself, the source address in each of these messages should have been the address of the adversary. However, the source address in each inserted erqst(x) message is recorded to be the address of p so that the reply to the message is sent to p. Also, the source address in each inserted erply(x) message is recorded to be the address of some q[i] in order to hide the identity of the adversary.

6 Preventing Denial of Service Attacks

To prevent denial of service attacks, the routers in the Internet need to be modified to perform the following task: detect and discard any message whose source address is wrong. Note that this task can prevent any adversary from exploiting the natural vulnerability of any protocol, not only PING, to launch a denial of service attack against any host of that protocol. This task can be achieved using two complementary mechanisms named "Ingress Filtering" [4] and "Hop Integrity" [8]. These two complementary mechanisms can be described as follows:

i. **Ingress Filtering**:
A router that receives a message, supposedly from an adjacent host H, forwards the message only if the source address recorded in the message is that of H.

ii. **Hop Integrity**:
A router that receives a message, supposedly from an adjacent router R, forwards the message only after it checks that the message was indeed sent by R.

These two mechanisms can be used together to detect and discard any message, whose source address is wrong, that is inserted by a weak or strong adversary into the Internet. Thus, a large percentage of denial of service attacks can be prevented.

Another mechanism for detecting and discarding adversary messages is called soft firewalls. A soft firewall for a process p is another process fp that satisfies the following three conditions:

i. **Output Observation**:
Each message that process p intends for another process q is first sent to the firewall process fp before it is forwarded to process q.

ii. **Input Observation**:
Each message from another process q intended for process p is first sent to the firewall process fp before it is forwarded to process p.

iii. **Input Filtering**:
The firewall process fp maintains a coarse image of the local state of process p, and uses this image to detect and discard any inappropriate message intended for p from any other process or from any adversary.

(Note that the soft firewall processes described here are similar to stateless firewall processes described in [2], with one exception. A stateless firewall does not maintain any image of the local state of the process behind the firewall, whereas a soft firewall process maintains a soft state image of the local state of the process behind the firewall.)

A possible soft firewall for process p in the PING protocol is a process fp that maintains one bit "w" as a coarse state for array "wait" in process p. Whenever fp receives an erqst(x) message from process p intended for process q[i], fp assigns its bit w the value 1. Process fp keeps the value of bit w "1" for one minute, since fp received the last erqst(x) message from p, then fp assigns bit w the value "0". (The one minute is an ample time for the sent erqst(x) message to reach the intended q[i] and for the resulting erply(x) to return from q[i] to p.) Whenever the firewall process fp receives an erply(x) message intended for p, fp checks the current value of bit w and forwards the received erply(x) message to process p only if the value of bit w is "1". Thus, all erply(x) messages, that are generated by the weak adversary, are discarded by fp before they reach process p (as long as p itself does not send erqst(x) messages to any q[i]).

7 Concluding Remarks

Our objective in this paper is three-fold. First, we want to demonstrate the utility of a new definition of system security [7] that is based on the three concepts of closure, convergence, and protection of convergence theory. Our demonstration show that the concept of protection as presented in [7] is too strong, and suggest a sensible weakening of this concept, discussed in Section 4. Second, we want to formally show that the PING protocol is secure (against a weak adversary), despite the fact that this protocol has been used repeatedly to launch denial of service attacks against several computers in the Internet. Our proof, based on the two state transition diagrams in Figures 1 and 2, is both simple and straightforward. Third, we want to make the point that every protocol (whether secure or unsecure) has a natural vulnerability that can be exploited by (possibly weak) adversaries to attack any computer that hosts this protocol. Such attacks can be foiled, not by making protocols more secure which is impossible, but by detecting the attacking messages early on and discarding them promptly. In this regard, the ideas of ingress filtering, hop integrity, and soft firewalls offer much hope.

References

1. Arora, A., Gouda, M.G.: Closure and convergence: A foundation for fault-tolerant computing. IEEE Transactions on Software Engineering, Vol. 19, No. 3 (1993) 1015–1027
2. Cheswick, W.R., Bellovin, S.M.: Firewalls and Internet Security. 1st edn. Addison-Wesley Publishing Co., Reading, Massachusetts (1994)
3. CERT Advisory: Smurf IP Denial-of-Service Attacks. CERT Advisory CA-1998-01, `http://www.cert.org/` (1998)
4. Ferguson, P., Senie, D.: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267 (1998)
5. Dolev, S.: Self-Stabilization. 1st edn. MIT Press, Cambridge Massachusetts (2000)
6. Gouda, M.G.: Elements of Network Protocol Design. 1st edn. John Wiley & Sons, New York, New York (1998)

7. Gouda, M.G.: Elements of security: Closure, convergence, and protection. Information Processing Letters, Vol. 77, Nos. 2–4 (2001) 109–114
8. Gouda, M.G., Elnozahy, E.N., Huang, C.-T., McGuire, T.M.: Hop Integrity in Computer Networks. Proceedings of the 8th IEEE International Conference on Network Protocols (2000) 3–11
9. Postel, J.: Internet Control Message Protocol. RFC 792 (1981)