Approach to Mitigate DOS & DDOS Attacks in the Presence of Clock Drifts-Survey

Sachin K.R[1], Smt. Usha M.S[2]
[1] Student, [2] Associate professor, [1,2]Department of Computer Science and Engineering
[1,2]NIE Institute of Technology Mysore, Visvesvaraya Technological University, Karnataka, India.

***Abstract* – Common problems in network-based applications is that they open some known communication port(s), making themselves targets for DoS and DDoS attacks. To overcome these attacks, solutions developed are based on pseudo-random port-hopping. As this method requires communicating parties hop in a synchronized manner, these solutions suggest the presence of synchronized clocks, which can become targets to DoS or DDoS attack themselves and acknowledgment-based protocols between a client-server, where in, if acknowledgments are lost, can cause port to be open for longer time and thus leads to DoS attacks. In this paper we examine, communicating parties having clock rate drift. We present an adaptive algorithm, HOPERAA, for hopping in the presence of clock-drift. Here, each client interacts with the server independently of the other clients, without the need of time servers or the acknowledgements. We also propose BIGWHEEL algorithm, for server to communicate with multiple clients in a port-hopping manner. Here, server need not open fixed number of port in the beginning, nor does it require client to get a first-contact port from a third party.**

***Keywords* - Network Attacks, Port number, Contact initiation, Distributed Denial of Service Attack.**

I. INTRODUCTION

A Network attack is a threat, intrusion, and denial of service or other attack on a network infrastructure that will analyse your network and gain information to eventually cause your network to crash or to become corrupted.

There are at least seven types of network attacks. They are *mapping, Spoofing attack, Sniffing attack, Hijacking, Trojans, Social engineering, DoS and DDoS.* But this paper describes about Denial of Service (DoS) attacks and Distributed Denial of Service (DDoS) attacks.

Denial of Service (DoS) attack is an attempt to make machine or network resource unavailable to its intended users by disrupting it, crashing it, jamming it or flooding. The motivation for DoS attacks is not to break into a system. One can say that this will typically happen through following means:

1. Crashing the system. Deny communication between systems.
2. Bring the network or the system down or have it operate at a reduced speed which affects productivity.
3. Hang the system, which is more dangerous than crashing since there is no automatic reboot.

DoS attacks can also be major components of other type of attacks. There are many types of DoS attacks. Among them important attacks that exist are Teardrop attack, Bandwidth attack, Blind attack, SYN flood attack and Smurf attack.

1. **Teardrop attack** sends incorrect IP fragments to the target server. So the server gets crashed if it does not implement TCP/IP fragmentation reassembly code properly.
2. A **Bandwidth attack** is where an attacker tries to consume the available bandwidth of a network by sending a flood of packets. They attacks server fast in a kbps speed.
3. With a **Blind attack** the attacker uses one or more forged IP addresses, which is extremely difficult for the server to filter those packets.
4. A **SYN flood** is a form of denial of service attack in which an attacker sends a succession of SYN requests to a target's system in an attempt to consume enough server resources to make the system unresponsive to legitimate traffic.
5. In **Smurf attack**, the attacker sends an IP ping request to any website. The ping packet is broadcast to a number of hosts within that site's local network.

The request is from another site and it is the target site that receives the denial of service attack.

A Distributed Denial of Service (DDoS) attack is the combined effort of several machines to bring down victim. It occurs when multiple compromised systems or multiple attackers flood the bandwidth or resources of a targeted system with useless traffic.

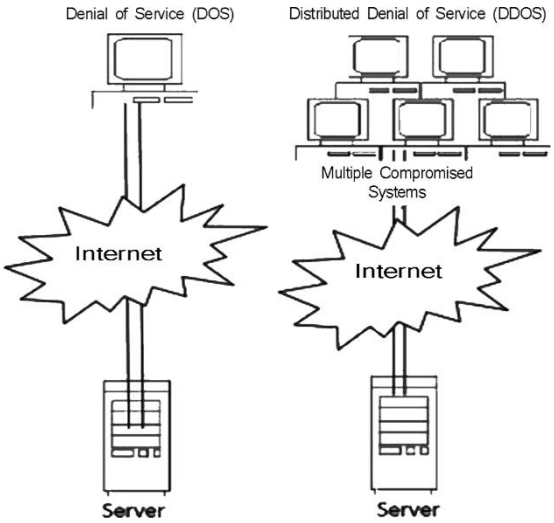

Fig 1: Denial of Service attack and Distributed Denial of Service attack

Most of the time, attackers collect many (millions) of *zombie machines or bots*. In many cases there is a master machine that launches the attack to zombie machines that are part of a bot network. Some bot networks contain many thousands of machines used to launch an attack

To avoid all those attacks we use port –hopping technique in the presence of *clock drifts*, which *implies* clock values, can vary arbitrarily much with time and in multiparty applications. The application parties communicate via ports that change periodically over time using the pseudorandom function. This method was inspired from the well-known frequency hopping paradigm used in signal communication protocols. The focus in that area is to find hopping sequences with the optimal Hamming Correlation Properties. But in the earlier solutions, port-hopping support between pairs of processes which are synchronous or exchange acknowledgements. Acknowledgment, if lost, can cause a port to be open for longer time .This become targets to DoS attack themselves.

We propose two algorithms in proposed to avoid attacks. One is HOPERAA (Hopping-Period-Align-and-Adjust) algorithm, executed by each client to adjust its period length and align its hopping time with the server. Second is BIGWHEEL algorithm enable multiparty communication with port hopping, this algorithm for a server to support hopping with many clients.

The basic idea in both algorithms is that each client interacts independently with the server and considers the server's clock as the point of reference clock. In this algorithm, there is no need for group synchronization which would raise scalability issues. The HOPERAA and BIGWHEEL algorithm's detailed explanations are in the following Section.

II. PROBLEM ANALYSIS

Problem that an adversary wants to subvert the communication of client-server application is by attacking the communication channels. At each time point, some ports must be open at the server side to receive the messages sent from legitimate clients The N which denotes the size of port number space, meaning that there are N ports that the server can use for communication at the server side.

The server and the legitimate clients share a pseudorandom function. It generate the port numbers which will be used in the communication. We assume there exists a preceding authentication procedure. It enables the server to distinguish the messages from the genuine clients. We assume that every client is honest which means any execution of the client is based on the protocol and clients will not reveal the random function to the adversary. The attacker is modeled as an adaptive adversary who can eavesdrop and attack a bounded number of ports simultaneously.

III. SYSTEM ARCHITECTURE

Clock drift is used to maintain clock rates between client and server. Clock rates between the client and server has been adjusted and aligned to same. Based on the application the client use, Pseudo random function generates pseudo random seed in the server and it assigns ports to each client.

Contact messages: (1) Client to send message to a port that is already closed or is not opened yet and then client align the hopping time period at adversary chosen time intervals to control the align.(2) HOPERRA Executed by each client to adjust its hopping period length and align its hopping period with the server.

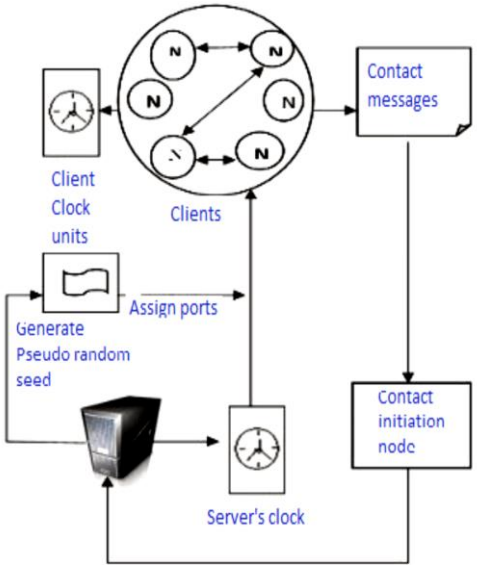

Fig 2: System Architecture

IV. RELATED WORK

There are many network-based solutions against DDoS attacks. These solutions usually use **routers or overlay networks** to filter malicious traffic. A good survey about network-based defence mechanisms against DDoS attacks is presented by Peng et al. [5]. Here, we focus on application-based mitigation.

The most closely related results are the port-hopping protocols presented by Badishi et al [4]. The **ack-based protocol** in that paper is focused on the communication only between two parties, modeled as sender and receiver. The receiver sends back an acknowledgment for every message received from the sender, and the sender uses these acknowledgments as the signals to change the destination port numbers of its messages. Since this protocol is ack-based, time synchronization is not necessary. But note that the acknowledgments can be lost in the network, and this may keep the two parties using a certain port for a longer time. If the attacker gets the port number during this time, then a *directed attack* will be launched under which the communication can hardly survive.

Hari and Dohi et al. [6] presented a **sensitivity analysis** of this protocol to various attack. To cope with that, a solution that **reinitializes the protocol** is presented in [4]. The latter solution depends on that the clocks have the same rate; it allows for bounded drift in the clock phases (resulting in bounded differences of clock values) but not their rates (which would imply arbitrary differences of clock values). In [4] the authors also present a rigorous model and analysis of the problem of possible DoS to applications (ports) by an adaptive adversary, i.e. one that can eavesdrop, as in our case, too. The analysis, besides the parts that involve the port-hopping protocols proposed in that paper, also includes a part on the effect of the adversary when it launches blind attacks. As that part of the analysis holds regardless of the application's defense mechanism, it carries over any setting. Hence, we do not elaborate on that part.

Another port-hopping scheme for the client-server mode was proposed by Lee and Thing et al. [7]. There, **time is divided into discrete time slots**. The clients and the server share a pseudo-random function to compute which port should be used in a certain time slot. This scheme bounds the time offset and the message delay by a constant value l, so there is no time synchronization mechanism. Instead, the valid time of the communication port for a time slot is prolonged both backward and forward by $\frac{1}{2}\, l$. This scheme shows the basic idea of the time-based port hopping, but it only works when no clock drift exists, which limits its adaptability.

There is a **client-transparent approach** proposed by Srivatsa et al. [8] which is quite similar to port hopping. This approach uses JavaScript to embed authentication code into the TCP/IP layer of the networking stack, so the messages with invalid authentication code would be filtered by the server's firewall. In order to defend the DoS attacks, the authentication code changes periodically. There is a challenge server in charge of issuing keys, controlling the number of clients connected with the server and synchronizing the clients with the server as well. Since this approach relies on the challenge server, the protection of the challenge server is quite important. The paper mentions that a cryptographic based mechanism can be used to protect the challenge server, but this was not discussed in detail.

V. SOLUTION TO THE PROBLEM

The solution was designed to overcome a very common vulnerability, present at the application layer, in which certain programs may open "well-known ports" in order to perform whatever action they're meant to do thus, an attacker, can eavesdrop some packages, discover which port is being used and launch a directed attack over such port; or, even if it can't discover which port is being used, he can still perform blind attacks (sending packets to a largely random set of ports, to then target any of the ports who responded) and eventually accomplish the same objective. Taking this scenario in consideration, the solution studied is based on the idea that the parties involved are capable of communicating with each other "hopping" in between different available ports over time; an attacker is not able to perform a attack over a particular/vulnerable port (used for communication) since the latter is always changing in a synchronized manner. In order to achieve such behaviour, and overcome the need of a global

synchronization mechanism in the system, two algorithms were proposed; (1) BIG WHEEL, which is used for servers to support communication with multiple clients, in a port-hopping manner and; (2) HOPERAA, used for synchronous port hopping in the presence of clock-drifts. In the following sections, we will discuss in detail how these algorithms work.

A. *Hopping Period, Align and Adjust Algorithm :*

The Hopping Period, Align and Adjust algorithm (Referred as to HOPERAA) is an adaptive algorithm, which is executed by each client when its hopping period length and alignment drift apart from the server's; the latter, ensure synchronization among the parties, without having to rely on a "common synchronization" server.

Within a network, is very common for the clients to have local clocks which differs from the one of the server, sometimes it could be slower and sometimes faster; since the ports being used for communication become available and unavailable over time, the periods of the Client and Server may start drifting apart from each other after some time, causing messages loss due to the fact that the Client may send messages to some of the Server's ports that has been closed or not yet open due to asynchronous clocks. The HOPERAA algorithm fulfil its objective by dealing with problems as the previous scenario and, avoiding messages losses due to unsynchronized port hops.

Consider the following ground rules and assumptions:

- Each communication party has its own clock and the clock rate of each local clock is constant.
- The server's clock is used as reference for the whole operation hence each client's clock drift is defined as the ratio between its own clock and the server's clock rate.
- The client and server share a "common secret", which is a pseudo-random function *"fψ"* used to generate the port number for transmission
- "μ" is the maximum round-trip delivery latency for the messages.
- This solution mitigates attacks based on the application layer; therefore it's assumed that network is always available and attacks depleting the bandwidth of the server's network are not considered.
- The server has a set of N ports (Port Number Space) available for communicating with legitimate clients.
- Ports opened at the server's side can be of two types, based on their function. (1) **Worker ports**, used for receiving data messages from the client or (2) **Guard Ports**, used for receiving coordination messages from the client. Guard Ports can become worker ports after some time.
- Worker ports are opened every L time units and, they remain open for L+μ time units. (fig 3)

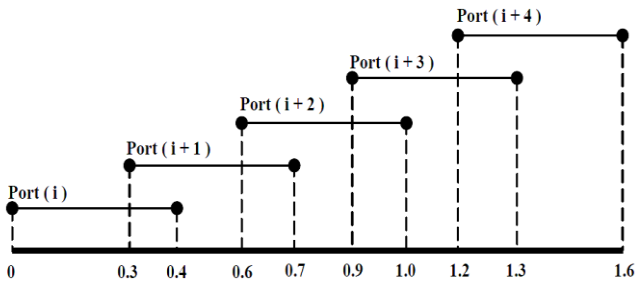

Fig 3: Assuming that L=0.3 and μ = 0.1, the worker ports at the server side will open and close as above presented

Phase 1: Contact-Initialization

During this phase, the Client contacts the Server without any "well-known" port being opened at neither the server's side nor, Client having to rely on a third-party to get the port information.

In order to achieve this, the server must do the following:

1. Divide the range of port numbers into "k" intervals evenly.

Open "k" different guard ports at the same time, one of them per one interval, and changes them every "τ" time units.

After the server has executed the previous actions, the algorithm then performs like this:

1. Client tries to contact the Server by sending "contact-initiation messages" to all the ports in a randomly-chosen interval. In this message the Client includes a timestamp "time" with the time at which the message was sent.
2. When the Server receives the "contact-initiation message", it waits until the next worker port opens, open a session for the Client who contacted it and replies with the following information:
   a. **"σ"**, seed used to compute the next worker port.
   b. **"time"**, Timestamp at which the reply was sent.
   c. **"t1"**, time at which the Server received the contact-initiation message from the Client.
   d. **"h1"**, timestamp at which the Client sent the contact-initiation message
3. If the Server doesn't receive any message from the Client by the next worker port, the session opened in the step 2 will be closed; on the other hand, if the Client doesn't receive any reply from the Server, after 2μ+L time units it will send "contact-initiation messages" to another randomly chosen interval.

The actions described above, from both Client and Server, are presented as an algorithm below.

(1) Algorithm for the Client in the Contact-Initiation Phase
(2) Algorithm for the Server in the Contact-Initiation Phase

Tc = *undef*;
Reply = *false*;

- Sending contact-initiation messages:

```
while (Reply == false) do
    I = SELECT (Ii | i ε {1, 2, ..., k})
    for (all ports in 'I') do
        SEND (init, time, p)
    end for
    WAIT (2μ +L)
end while
```

- After receiving ***(init, time, p)***:

```
t1 = time(now);
If (session == undef) then
    OPEN (session, C);
    h1 = time;
end if
WAIT (Next worker port opens)
SEND (reply, σ, timestamp, h1, t1)
```

Phase 2: HOPERAA Execution

After the contact-initiation phase, the application data from C to S is sent out through ports of S that change with period L time units of S's clock, corresponding to 'Pc' time units in C's clock (initially Pc = L) however, before actually start sending data to the Server, the Client has to perform the actions described in this section. As previously mentioned, this phase is reached after the Client has received the reply from the Server and, before it starts sending Data Packets; in this phase, the HOPERAA algorithm uses the clock's information, from the exchanged messages, to determine whether the Client's clock is slower or faster than the Server's and, based on such, it takes the proper actions to ensure successful data transmission. The reply sent by the Server to the Client is structured as follows:

Pkt (*reply, h1, t1, timestamp, seed***)**

In order to keep synchronous communication in between the parties, the following actions are performed:

1. The "HOPERAA execution interval" is initiated to 0.
2. The Client initializes the following variables:
   a. **Hc (t4)** = Time at which the Client received the reply from the Server.
   b. **Hc (t1)** = h1
   c. **t2** = t1
   d. **t3** = timestamp
3. The Client, bounds its clock drift using the following:

$$\frac{hc(t4) - hc(t1)}{t3 - t2 + 2\mu} \leq \boldsymbol{\rho} \leq \frac{hc(t4) - hc(t1)}{t3 - t2}$$

4. The HOPERAA execution interval (HEI) is calculated based on the following conditions:

   a. *If (ρl < ρu < 1) then* HEI: *(ρl.Δ)/(1-ρl)*
   b. *If (1 < ρl < ρu) then* HEI: *(ρu.Δ)/(ρu-1)*
   c. *If (ρl < 1) and (1 < ρu) then*
      HEI: *min {(ρl.Δ)/(1-ρl), (ρu.Δ)/(ρu-1)}*

5. The "HOPERAA execution interval" and the value of "Pc" are both adjusted based on the following conditions:

   a. *If (1 ≤ ρl ≤ ρu) then* ***Pc*** *= L (ρl) and*
      HEI: *(ρu.ρl.Δ)/(ρu-ρl)*
   b. *If (ρl ≤ ρu ≤ 1) then* ***Pc*** *= L (ρu) and*
      HEI: *(ρu.ρl.Δ)/(ρu-ρl)*
   c. If none of the conditions above are fulfilled then, do nothing.

HOPERAA algorithm

$t_{reply} \leftarrow$ the arrival time of $ReMsg$
$\rho_u \leftarrow \frac{t_{reply}-T_{send}}{timestamp-T_{arrive}}$
$\rho_l \leftarrow \frac{t_{reply}-T_{send}}{timestamp-T_{arrive}+2\mu}$
if $1 \leq \rho_l \leq \rho_u \| \rho_l \leq \rho_u \leq 1$ **then**
  $Interval_{HoPerAA} \leftarrow \frac{\rho_u \rho_l \Delta}{\rho_u - \rho_l}$
  if $1 \leq \rho_l \leq \rho_u$ **then**
    $L_c \leftarrow L \cdot \rho_l$
  else
    $L_c \leftarrow L \cdot \rho_u$
  end if
else
  $Interval_{HoPerAA} \leftarrow \min\{\frac{\rho_l \Delta}{1-\rho_l}, \frac{\rho_u \Delta}{\rho_u - 1}\}$
end if
  Call initiation stage Algorithm
  at time $(Time_{now} + Interval_{HoPerAA})$

Phase 3: Data Transmission

This phase is executed immediately after the Client has finished with the calculation from "phase 2" and, the following actions are taken:

1. As soon as the Client receives the reply, it performs the following:
   a. Sets its internal timer "Tc" to 0. This variable increases at the same rate as the client's local clock.
   b. Uses the seed "σ" and pseudo-random function *fψ* to generate the worker ports Pi = *fψ(*σ*)* and Pi+1 = *fψ(*σ+1*)*.
2. After calculating the worker ports, the Client will send the data messages immediately to Pi.
3. During the interval [i(Pc) - μ ≤ Tc ≤ i(Pc)], messages will be sent to both Pi and Pi + 1.
4. When "Tc becomes equal to i(Pc), Pi changes its value for the one of Pi+1 and Pi+1 is recalculated by using *fψ(*σ+i+1*)*, at every i ε N*. We can say that i acts as an index, whose initial value is 1, and it increases every time Pi and pi+1 are updated.

Depending on the value of the HOPERAAA execution interval, the transmission may be stopped to execute Phase 1 and 2; however, data transmission will be resumed after the latter two phases accomplish their purpose.
Actions described above, are presented as an algorithm below.

Algorithm used by the Client to send Data to the Server

- Receiving ***(reply, σ, timestamp, h1, t1)***:

```
if (Reply == false) then
    Reply = True;
    Tc = 0;
    Pc = L;
        /* Start sending DATA */
        Seq = 0;
        Pold = fψ(σ);
        Pnew = fψ(σ + 1);
        While true do
          SEND (Data, Pold)
            If (i(Pc) - μ ≤ Tc ≤ i(Pc)) then
              SEND (Data, Pold)
            end if
            If (Tc == i(Pc)) then
              Pold = Pnew;
                Pnew = fψ(σ + i + 1);
                i ++;
            end if
         end while
end if
```

Phase 4: Termination

The Client will end the communication, by sending a termination-message and getting it acknowledged by the Server.

B. *BIGWHEEL Algorithm :*

BIG WHEEL algorithm is considered to deal with multi-party communication, supporting several Clients connected to the same Server. Since each Client follows the Server's hopping procedure, and take the Server's clock as reference, they are capable of communicate independently from each other.

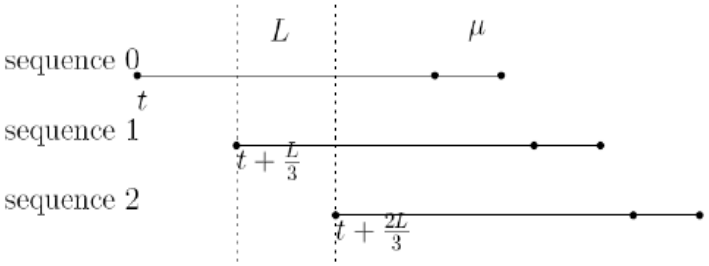

Fig 4: Shows the situation of m = 3 and the open time of P_0^i is t

When using BIG WHEEL, worker ports still remain open for L + μ units of time but now the Server will support m port number sequences instead of just one, as in the previous section (see fig 4); this afford more clients and also decrease the maximum waiting time for each one of them. In the Clients side, by using λ and the pseudo-random function f_ψ it is possible to generate different port number sequences if different values of λ are given.

Apart from these changes, the phases previously explained and the actions performed in each one of them are the same, so when the server receives a contact-initiation message from the Clients, it will send the reply at the closest opening time of a worker port (considering all m sequences) along with the corresponding value of λ for the sequence to which that worker port belongs. The pseudo-code is the following:

- Buffer B stores all contact-initiation messages received
- Whenever is time to open a new port from any of the m intervals:

```
if (time(now) == OpenTime P^i_J) then
    λ = Corresponding value for sequence "j"
    σ = Corresponding value for P^i_J;
    for all clients in B do
        if (session == undef) then
            OPEN(session, C)
            h1 = timestamp of the corresponding
                 contact-initiation message

        end if
        SEND(reply, σ, timestamp, h1, t1,λ)
    end for
CLEAR(buffer B)
```

VI. CONCLUSIONS

In this work, we investigate application-level protection against DoS attacks. More specifically, supporting port hopping is investigated in the presence of timing uncertainty and for enabling multiparty communications. We present an adaptive algorithm for dealing with port hopping in the presence of clock-rate drifts. For enabling multiparty communications with port-hopping, an algorithm is presented for a server to support port hopping with many clients, without the server needing to keep state for each client individually. The method does not induce any need for group synchronization which would have raised scalability issues, but instead employs a simple interface of the server with each client. The options for the adversary to launch a directed attack to the application's ports after eavesdropping is minimal, since the port hopping period of the protocol is fixed. Another main conclusion is that the adaptive method can work under timing uncertainty and specifically fixed clock drifts.

An interesting issue to investigate further is to address variable clock drifts and variable hopping frequencies.

VII. REFERENCES

[1] C. Kavitha, S. Mohana, Mrs. A. Karmel, "Survey on Mitigation of DOS and DDOS Attacks in the Presence of Clock drift", IJREAT, Volume 1, Issue 1, March.2013 (ISSN : 2320 – 8791).

[2] Z. Fu, M. Papatriantafilou, and P. Tsigas, "Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts," Proc. IEEE Int'l Symp. Reliable Distributed Systems (SRDS), Oct. 2008.

[3] CERT Advisory CA-1997-28 IP Denial-of-Service Attacks, http://www.cert.org/advisories/ca-1997-28.html, 2010.

[4] G. Badishi, A. Herzberg, and I. Keidar, "Keeping Denial-of-Service Attackers in the Dark," IEEE Trans. Dependable and Secure Computing, vol. 4, no. 3, pp. 191-204, July-Sept. 2007.

[5] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems," ACM Computing Survey, vol. 39, no. 1, p. 3, 2007.

[6] K. Hari and T. Dohi, "Sensitivity Analysis of Random Port Hopping," Proc. Seventh Int'l Conf. Ubiquitous Intelligence Computing and Seventh Int'l Conf. Autonomic and Trusted Computing (UIC/ATC), pp. 316-321, Oct. 2010.
[7] H. Lee and V. Thing, "Port Hopping for Resilient Networks," Proc. IEEE 60th Vehicular Technology Conf. (VTC2004-Fall), vol. 5, pp. 3291-3295, 2004.
[8] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu, "A Client-Transparent Approach to Defend against Denial of Service Attacks," Proc. IEEE 25th Symp. Reliable Distributed Systems (SRDS '06), pp. 61-70, 2006.