IMPLEMENTING A GLOBAL ANTI-DOS SERVICE BASED ON RANDOM
OVERLAY NETWORK

By

WEN-CHUAN SHEN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2004

To my family and friends who supported me with their patience and love.

TABLE OF CONTENTS

vii

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

IMPLEMENTING A GLOBAL ANTI-DOS SERVICE BASED ON RANDOM
OVERLAY NETWORK

By

Wen-Chuan Shen

December 2004

Chair: Shigang Chen
Major Department: Computer and Information of Science and Engineering

Distributed denial of service (DDoS) is a major threat to the Internet nowadays.
Legitimate users have a hard time accessing the servers that are under DDoS attacks.
What makes it worse is that the attacking tools are easy to get. Even people without
enough professional knowledge can launch a DDoS attack. Obviously, automated anti-
DDoS systems become more and more important.

Many current available solutions to DDoS attacks require universal installation and
configuration across different administrative realms, which are impossible or very
difficult to do in many situations. This thesis studied and provided another solution to
DDoS attacks. An anti-DoS service (called AID) is presented in this thesis, and no global
deployment is required. The AID service protects general TCP traffic. It guarantees all
registered clients can access a registered server fairly even when the server is under
DDoS attacks. A domain, like a school or company, can get immediate protection after
having the AID service.

Two primary parts of the AID service are the random overlay network (RON) and the distributed virtual-clock packet scheduling algorithm. The former forms tunnel trees, which connect registered clients to a registered server. Packets from registered clients go through the tree to the server when the server is under DDoS attacks. It is adapted and easy to manage. The latter is a packet scheduling algorithm to simulate client puzzles. It confines the amount of data a registered client can send to a server through RON to achieve fairness.

CHAPTER 1
INTRODUCTION

What Is Distributed Denial of Service (DDoS) Attack

A DoS attack intends to make a server out of its resource, which could be
bandwidth, buffers, CPU time, etc. Attackers can send a lot of requests to exhaust a
server's bandwidth. Other legitimate users will be unable to access the server. Another
example is SYN flooding attack [1-2]. To establish a connection, a client sends a SYN
packet at first. The server is going to keep this information in a buffer for a period T in
order to recognize the following incoming packets. If attackers send enough SYN
packets to make the buffer overflow, the server has no way to process requests from other
users. In some cases, like route table updating or software's bugs, a simple request can
make the server do a considerable amount of computation. In this case, normal users
cannot access the server. The basic idea of DoS attacks is simple, using a small amount
of resources to overwhelm the server.

What makes a DDoS attack different from a traditional DoS attack? In a DDoS
attack, the clients launching the attack might be victims as well. Hackers compromise
and install DDoS attacking programs on these hosts first. Then, hackers can remotely
control these victims to attack the servers cooperatively. With DDoS attacks, it is
possible to flood a big commercial server in a brute-force way. Besides, it becomes very
difficult to trace back the attacker because the compromised hosts are not the real
attackers. Usually, there are hundreds or thousands of compromised hosts and they are

1

around the whole world. It makes attackers feel safe to do this. Today, how to protect hosts against DDoS attacks is very important.

Related Work

Ingress Filtering Proposed by Ferguson and Senie

In *ingress filtering* [3], before a packet is transmitted into another domain, the router checks the packet's source address. If it does not match the ingress filter rule, probably a spoofed source address, the packet will be dropped. *Ingress filtering* helps to trace back the attacker, while it cannot prevent an attack originating from a valid source address.

Route-Based Packet Filtering Proposed by Park and Lee

With partial deployment (about 18% in Internet AS topologies) [4], spoofed IP packets can be prevented from reaching their intended targets effectively.

SYN-Dog Proposed by Wang, Zhang and Shin

SYN-dog [5] is a software agent which can be installed at leaf routers of stub networks. It is stateless and light-weighted. Therefore, *SYN-dog* itself is immune to any flooding attacks. It detects SYN flooding attack by monitoring behavior of SYN-SYN/ACK packets. *SYN-dog* can also trace back the attacking source.

Self-Complete Defense Systems

There are a huge number of hosts on the Internet. It is almost impossible to make every host join a specific defense system. Here comes the problem. If a server has the defense system installed, can it resist the DDoS attack from clients that do not participate in the same defense system? The answer is no for most existing DDoS defense systems. Suppose we have a networked system of $S + C$. C is a set of client networks, and S is a set of server networks. C' is a subset of C and S' is a subset of S. $C' + S'$ has a defense

system installed. A defense system is *self-complete* if any client in C' can still access any host in S' even when under DDoS attacks, as long as the client itself does not participate the attack. It does not care if the attack is from C or $C - C'$. In other words, a self-complete defense system should be able to defeat attacks from either inside area C' or outside area $C - C'$. A self-complete defense system makes itself a clean area in the Internet. The area does not have to cover the entire Internet, and hosts in it are protected.

Let us review the DDoS defense systems mentioned above.

- *Ingress filtering*: Source addresses of packets from $C - C'$ can be spoofed, because $C - C'$ do not check them. Therefore, DDoS attacks can be launched against the S' from $C - C'$. In this case, clients in C' have difficulty to access S' even though C' and S' both join the defense system. Therefore, *Ingress filtering* is not self-complete.

- *Route-Based Packet Filtering*: Packets with spoofed source are prevented from reaching their intended targets effectively as long as 18% of Internet AS's join the defense system. However 18% of Internet AS's is a huge number. It is not self-complete until C' is as large as 18% of Internet AS's.

- *SYN-dog*: Attackers can still do SYN-flooding to S' from $C - C'$, because $C - C'$ is not under protection. Similar to *ingress filtering*, unless C' is as big as C, attackers from $C - C'$ can make S' not accessible to C'. In consequence, *SYN-dog* is not self-complete.

The benefits of a self-complete system are apparent. It suits normal companies, schools and organizations. They can set up a self-complete defense system in their realm and become under protection immediately, independent of others. A working self-complete defense system (called AID), the detail of its structure and how it was implemented are presented in the thesis. The AID system's overview is in chapter 2.

CHAPTER 2
AID SYSTEM OVERVIEW

Fundamental Ideas

The idea of the anti-DoS system (called AID) is from Chen et al [6]. The AID

system contains clients, servers and AID stations physically. Another two important

parts in the AID system are random overlay network (RON) and distributed virtual-clock

packet scheduling algorithm [6]. Random overlay network is formed on AID stations.

We did not draw a clear line between DoS attacks and DDoS attacks in the thesis.

Whenever a server senses an attack, like flooding packets beyond its capacity or unusual

requests from clients, the AID defense system will be triggered. The AID system's

architecture is shown in Fig. 2-1.
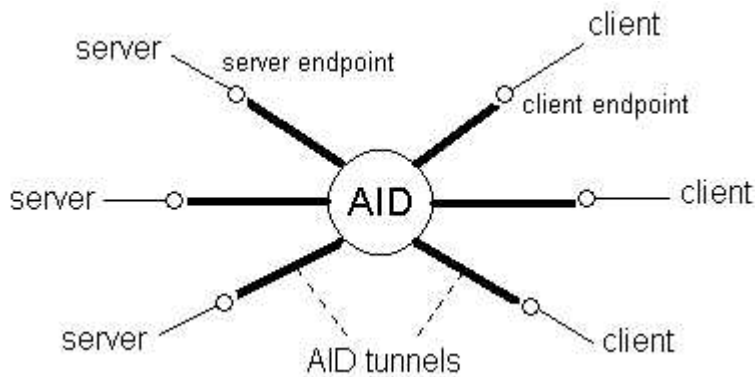
Figure 2-1. AID system architecture. The AID circle and AID tunnels symbolize RON,
which is composed by AID stations. A client point can mean a client network,
not just a client host. Likewise, a server point can be a server network behind
a router.

Who Is Protected?

Register clients and servers that we want to protect at their nearby AID stations.

The registration brings a shared secret key between the AID station and the registered

host. The secret key is used in AID tunnels for integrity checking when under DDoS attacks. Everyone can join the AID service by registering at an AID station.

What Is Random Overlay Network (RON) for?

RON consists of all AID stations. When a registered server is under DoS attacks and other registered clients try to access the server, packets from these clients will go through the RON instead of the Internet. We say that these packets are entering AID tunnels, which are tree structure. They go through AID tunnels all the way to the attacked server. Other packets from unregistered clients will reach the server via the Internet. AID tunnels are one-way path for packets from registered clients to attacked registered servers. We do not allow traffic from registered servers to registered clients entering RON to minimize the load of AID stations. It is transmitted via the Internet. Of course, traffic related to unregistered clients or unregistered servers cannot enter AID tunnels.

How Does AID Defense System Work?

When under attacks, packets from registered clients go through RON but packets from unregistered clients go through the Internet. We make packets from AID tunnels have higher priority. Servers process them first. Hence, the external traffic (from unregistered clients) cannot influence the internal traffic (from registered clients). How about if a registered client is an attacker itself? Well, that is why we have distributed virtual-clock packet scheduling algorithm, which simulates client puzzles [7-10]. Every AID station manages a virtual clock for every tunnel hooking on a client network. If a client has behavior of flooding a registered server, its virtual clock will run fast. When virtual clock's value is too big, packets from that client will be dropped. By doing this, we can separate the attacking traffic out and block it.

Traffic in AID tunnels has integrity protection. Remember the secret key a client or server got after registering at an AID station? The secret key and other important data in a packet are put together and digested by MD5 algorithm. The 128-bit-long packet digest is used for integrity checking. As a result, alteration of packets in AID tunnels will be detected. Because the third party cannot forge the packets, integrity checking is also authenticity checking, verifying that the packets are really from the hosts as they claimed.

Now we know how AID stations interact with clients and servers. We also know what random overlay network and distributed virtual-clock packet scheduling algorithm are for. More details of the AID system and how it was implemented were revealed in the following chapters.

Implementation Issues

We chose Linux as our developing platform and our programs only work on IPv4. As mentioned in the section "How Does AID Defense System Work," a registered client should send it packets via RON instead of the Internet. Meanwhile, the attacked server should be able to tell where a packet is from and give the one from AID tunnels higher priority. We do not want people to recompile their Linux kernels or rewrite application codes if possible. So, we introduced another layer between application layer and transport layer, called AID layer. Extra information is added into a packet as an AID layer header for the AID service. Higher application layer programs should not notice the existing of AID programs.

Netfilter is one tool we used in our AID system to intercept and modify packets. It was included in Linux 2.4. It supports five different hooks and they are NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN, NF_IP_FORWARD,

NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING. Fig. 2-2 shows how a packet goes

through these hooks.
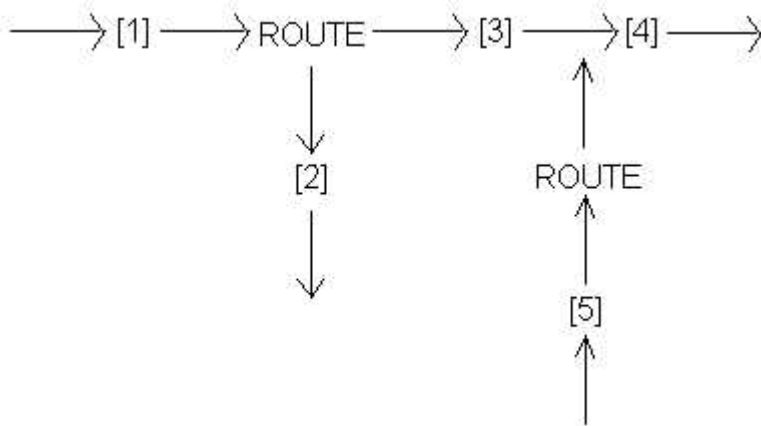
Figure 2-2. Netfilter hooks. Hook 1 is NF_IP_PRE_ROUTING, hook 2 is
         NF_IP_LOCAL_IN, hook 3 is NF_IP_FORWARD, hook 4 is
         NF_IP_POST_ROUTING and hook 5 is NF_IP_LOCAL_OUT.

NF_IP_PRE_ROUTING: A packet hits the hook after reaching the host and sanity

checks but before the routing decision.

NF_IP_LOCAL_IN: A packet hits the hook after the routing decision and the

packet's destination is this host.

NF_IP_FORWARD: A packet hits the hook after the routing decision if the

packet's destination is another interface.

NF_IP_LOCAL_OUT: A packet hits the hook when going down the kernel after a

process creates and sends out the packet.

NF_IP_POST_ROUTING: A packet hits the hook right before it is put on the wire.

We can inject our handling functions into any of these hooks. When a packet goes

through hooks, their handling functions will be executed. That is where and how we can

modify the packet.

Packets Intercepting Modules

Our first step is to write modules to intercept interested packets in proper hook positions. The interested packets are queued into userspace. Doing in this way may cause some performance penalty because of switching between kernelspace and userspace. However, there are also some advantages. First, it is easier to debug a program running in userspace. Second, we have more libraries handy. Third, misbehavior, if any, of a program will not crash the whole system. Three modules totally, clientFilter.o takes care of packets in/out clients. Likewise, AIDFilter.o is for AID stations, and serverFilter.o is for servers. After loading an appropriate module on the host, interested packets will be queued to userspace. To stop it, just unload the loaded module. These queued packets will be inspected or modified later.

Handling Queued Packets in Userspace

To deal with the queued packets in userspace we need the library *libipq* developed by James Morris. It can be found easily on the Internet and simple to install. With the library, we can grab one packet out of the queue every time. We can drop the packet, do nothing, or modify and send it back to the kernel. One thing should be noticed is that checksums in TCP/UDP and IP headers need to be recalculated if the packet is altered.

In the client end, all outgoing TCP packets are queued to userspace. If a packet's destination server is under attacks, its destination IP will be converted to an AID station's IP. The AID station is the one this client registered at. The AID station will notify the registered client if a registered server is currently under attack. If no attacks, packets are routed as usual. A registered client executes the program *client*.

In AID stations, AID tunnel trees are built up to route packets to their destination servers. Assume a server registered at an AID station named A_s. When the server is

under DoS attacks, an AID tunnel tree rooted at A_s is formed. Packets from registered client to this server are routed from tree leaves to the root A_s and finally to the server. Besides routing, virtual clocks for every client are maintained by AID stations as well. An AID station executes the program *AID*.

In the server end, packets from the Internet and AID tunnels are separated. Process the latter first. If under attacks, a server will send alert messages to its registering AID stations, A_s. Then, this AID stations broadcasts alert messages to other AID stations. AID tunnel trees are constructed. A server executes the program *server*.

Showing Statistics

Programs *client*, *server* and *AID*, all record statistic information of TCP traffic. Users can know how many packets got through or were dropped and the reasons of dropping. They all have a while loop in main() whose condition is always true. To keep programs simple, we did not use threads or fork a child process. Then how can the programs interact with users when they want to see the statistics of traffic? The answer is *signal*. The reaction of the signal SIGINT was redefined in these three programs. When Ctrl-c is typed, programs will not be terminated. Instead, statistic information is printed out. We can type Ctrl-\ to send the signal SIGQUIT to stop the programs.

CHAPTER 3
AID LAYER

AID layer is added between application layer and transport layer. In this chapter,

we defined AID layer headers, which are inserted between TCP/UDP headers and

application layer data in a packet. Since we do not want AID service users to recompile

their Linux kernels, Linux kernels have no idea of this new layer. AID layer headers are

treated as application data actually by Linux kernels. In clients, the program *client* adds

an AID layer header before a packet is sent out. In servers, the program *server* takes the

AID layer header off. Only the AID system can recognize AID layer headers. As for

AID stations, AID layer headers contain data needed for routing, constructing AID tunnel

trees, and etc. Consequently, application programs in the client end or server end do not

know they are already in the AID service and protected. Currently, only TCP traffic is

protected in the AID system because TCP's congestion control feature is needed.

When a packet enters RON, if it is a TCP packet, it belongs to traffic from a

registered client to a registered server. If it is a UDP packet, such as a PULL message, or

PUSH message, the packet is used to control the AID system. We explained what these

UDP messages are later in this chapter. We have different AID layer headers for TCP and

UDP packets. Both TCP and UDP traffic have integrity guard.

AID Layer for TCP Traffic

We have two kinds of AID layer headers for TCP packets. One is for packets

transmitted via the Internet and the other is for packets transmitted via the RON (AID

tunnels). Fig. 3-1 shows the contents of the headers and where they are inserted.

10

A

IP header	TCP header	AID layer header	application layer data

recognizing field 32 bits

B

IP header	TCP header	AID layer header	application layer data

server IP 32 bits (recognizing field)	md5 digest 128 bits	virtual clock timestamp 64 bits
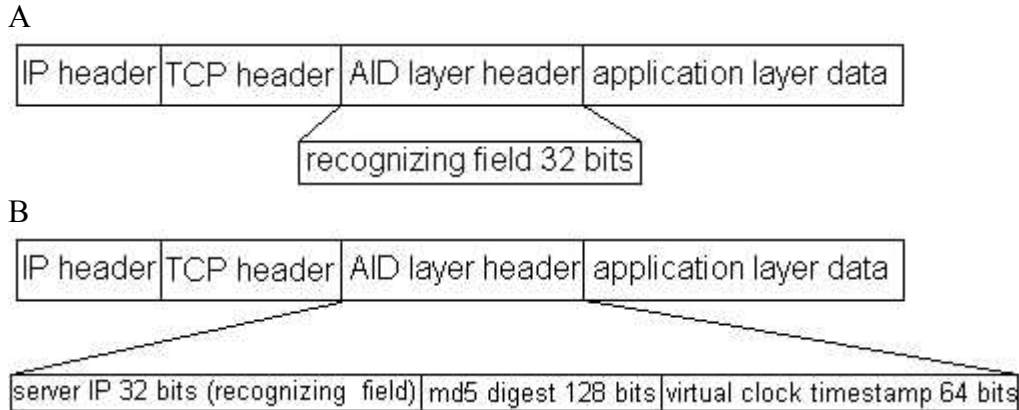
Figure 3-1. AID layer header for TCP packets. A) For normal TCP packets that do not enter AID tunnels. *Recognizing field* is 0. B) For TCP packets to attacked servers that enters AID tunnels. Notice that *recognizing field* in the first figure is at the same position as *server IP* in the second figure.

Server IP field is used to save the IP address of destination server. To travel through AID tunnels, a packet's destination is changed to the AID station where it is routed next. However, the final destination is still the server, so we need to keep this information. *Md5 digest* field is used to check integrity. If checking fails, the packet will be dropped. *Virtual clock timestamp* field is used in distributed virtual-clock packet scheduling algorithm.

Why do we need *recognizing field* even in normal packets? The problem is that when a server gets a packet, it has no way to know if the packet is from the Internet or AID tunnels. *Recognizing field* of normal packets is at the same location as *server IP* field of packets to an attacked server, right after the TCP header. When a packet arrives, the server checks this location. If it is 0, the packet is from the Internet; otherwise, the packet is from the AID tunnels. We assume server IP cannot be 0.0.0.0, so no conflicts.

What information is under integrity protection?

- Source IP (4 bytes) and destination IP (4 bytes) addresses in the IP header: Source IP is always a client's IP address. However, destination IP could be the destination

server's IP address if transmitted via the Internet or an AID station's IP address if in the AID tunnels.

- Source port (2 bytes) and destination port (2 bytes) in the TCP header: Unlike destination IP, a packet's destination port is not changed when entering AID tunnels.

- Sequence number (4 bytes) and acknowledgement number (4 bytes) in the TCP header.

- (a) *Recognizing field* (4 bytes) in the AID layer header.
  (b) *Server IP* (4 bytes) in the AID layer header.
  (a) and (b) are in the same position and have the same size. Its value is 0 for normal packets, or destination server's IP for packets entering RON.

- *Virtual clock timestamp* in the AID layer: Only packets go through AID tunnels have this field.

- Whole application layer data.

AID Layer for UDP Traffic

There are several different UDP messages used to control the AID system. They are distinguished by the *packet type* field in the AID layer header. All of these messages are integrity-protected.

PUSH Message

PUSH messages notify other AID stations a server is currently under attacks. Fig. 3-2 shows the content of a PUSH message. An AID station or registered server can send PUSH messages. *Md5 digest* is for integrity protection. *Packet type* is set to 2 for PUSH messages (defined in global.h). *AIDNO* means AID station number, which records how many AID stations a packet will pass before reaching the server. It is essential for establishing AID tunnels. More details are explained in later chapters. *Service port* and *server IP* are the attacked server's IP address and port number.
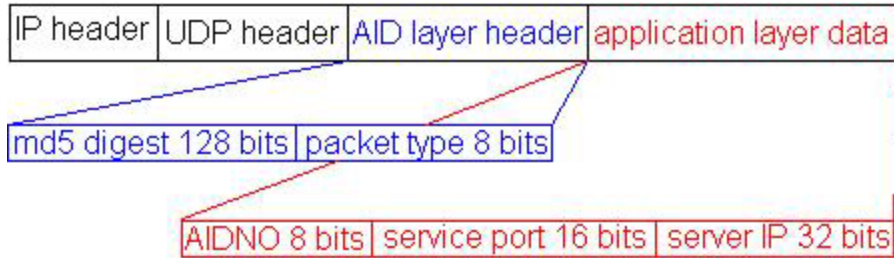
Figure 3-2. PUSH message. The AID layer header is inserted between the UDP header and application layer data.

PULL Message

PULL messages ask other AID stations what servers are currently under attacks.

Figure 3-3. PULL message. There is no application layer data in a PULL message.

Fig. 3-3 shows the content of a PULL message. There is nothing behind the AID layer header. *Md5 digest* is for integrity protection. *Packet type* is set to 0 here (defined in global.h).

PULLANS Message

When an AID station gets a PULL message from another AID station, the former will return information of all currently attacked servers it knows to the latter. Sending PULLANS messages does it. Fig. 3-4 shows the content of a PULLANS message. *Packet type* is set to 1 for PULLANS messages (defined in globa.h). Every AID station maintains a service list, which stores information of attacked servers. Each attacked server is a service list node.

If an AID station has information of *N* attacked servers, there will be *N* nodes in the service list to be sent out. In a service list node, *Distance* says that from this AID station

how many AID stations a packet still needs to pass to achieve the server, excluding the first AID station. The distance information helps to construct AID tunnel trees.

A

| IP header | UDP header | AID layer header | application layer data |

| md5 digest 128 bits | packet type 8 bits |

| service list node | service list node | ······ |
service list

B
```
service list node {
    unsigned int ip;
    unsigned short port;
    unsigned char distance;
}
```
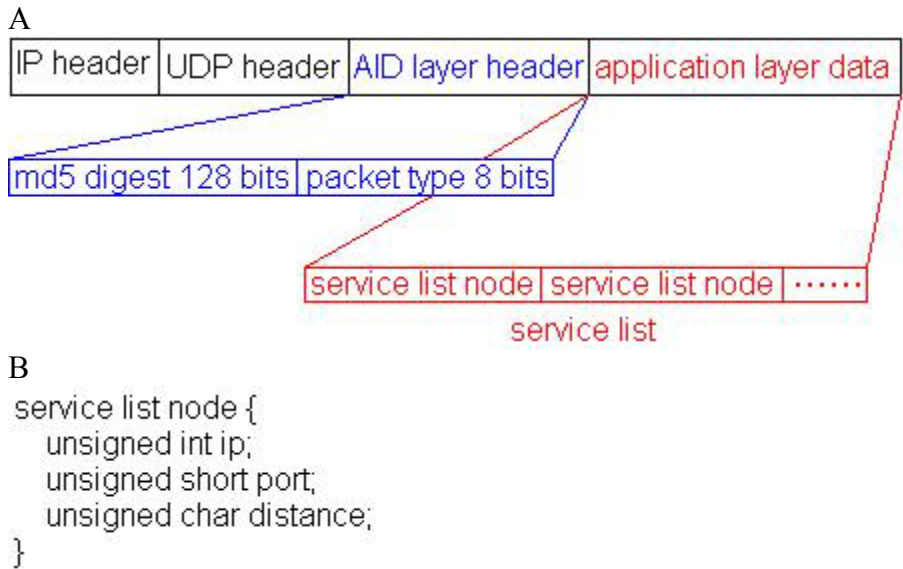
Figure 3-4. PULLANS message. A) The content of a PULLANS message. B) The structure of the service list node. It contains IP and port of an attacked server.

CTRLT Message

T, the waiting interval, is for adjusting the speed of a virtual clock. When the arrival rate is larger than a server's capacity, a bigger *T* will be sent to AID stations to accelerate virtual clocks.

| IP header | UDP header | AID layer header | application layer data |

| md5 digest 128 bits | packet type 8 bits |

| server ip 32 bits | T 64 bits |
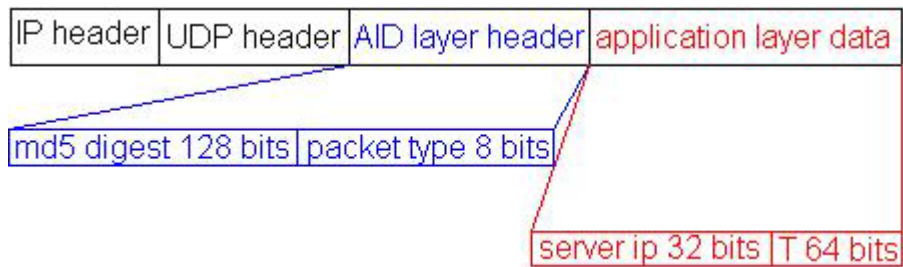
Figure 3-5. CTRLT message.

Packet type is set to 3 (defined as in global.h). A CTRLT message is for a specific tunnel tree of the attacked server with *server IP*. *T* field contains the new value of *T* for that specific tunnel tree. After getting a CTRLT message, an AID station updates its *T*.

Most of UDP messages are related to RON maintenance and distributed virtual-clock packet scheduling. We have not talked about them so far. They would be pointed out in later chapters.

What fields are under integrity protection?

- Source IP (4 bytes) and destination IP (4 bytes) addresses in the IP header.

- Source port (2 bytes) and destination port (2 bytes) in the UDP header.

- *Packet type* (1 byte) in AID layer header.

- Whole application layer data.

Implementation Issues

Most UDP messages in our AID system have fixed size, except for the PULLANS message. Its size depends on how many nodes in the service list. If there are many nodes, the message packet will be too big to be sent out. In the circumstance, it should be divided into two or more PULLANS messages. How big is too big? We defined a constant, UDPMAXSIZE, in global.h. When a UDP message is bigger than UDPMAXSIZE, it will be chopped up into several packets.

CHAPTER 4
CLIENT END

On the client end, we need to filter incoming and outgoing packets. For example, when a TCP packet is leaving the client end, its destination needs to be checked. If the destination server is under attacks, the packet will enter AID tunnels; otherwise, it is routed as usual.

ClientFilter.c and client.c are two main source files for client ends. ClientFilter.c is compiled as a module, queuing interested packets into userspace. Then, client.c takes queued packets out and does whatever is necessary. After a client registers at an AID station, A_c, it can get a secret key. The key is used to verify that the third party did not modify the communication between the client and A_c. The client also keeps A_c's IP address. If it tries to access an attacked registered server, its packet will be forwarded to A_c.

When an AID station is informed that a server is attacked, it will send PUSH messages to its registered clients. For instance, the client gets PUSH messages from A_c, which is the AID station it registered at. All UDP messages in the AID system are sent to port 4369. However, there is no program in application level listening on this port. UDP packets to port 4369 are handled by the program *client*.

Module ClientFilter.o

By compiling clientFilter.c, we can get the module clientFilter.o. It hooks on handling functions at hooks NF_IP_PRE_ROUTING, NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING. At NF_IP_PRE_ROUTING, only UDP packets to the port

16

4369 are queued. Other incoming traffic is not related to the AID system. At NF_IP_LOCAL_OUT, all outgoing TCP packets are queued, except for the local traffic. Local traffic goes from loopback interface, 127.0.0.1, to loopback interface. At NF_IP_POST_ROUTING, all outgoing TCP packets are queued, except for the local traffic.

Only outgoing TCP packets are queued since currently only TCP traffic is protected. When the module clientFilter.o is loaded in a host, the host must run the program *client* as well. Otherwise interested packets keep getting into the queue, but no programs take them out of the queue. The traffic is blocked if this happens. A host should load the module clientFilter.o and run the program *client* at the same time. It is meaningless to do just one of them.

Program *Client*

By compiling client.c and linking other relative source files, we can get the executable program *client*. It has a while loop in main() whose condition is always true. The program *client* deals with packets queued at different hooks by the module clientFilter.o. Now, we discuss what the program *client* does to packets from different hooks.

Packets from NF_IP_PRE_ROUTING

All packets in the queue grabbed at the hook NF_IP_PRE_ROUTING are UDP traffic to port 4369. For a client, the only UDP message of the AID system (to the port 4369) is PUSH. PUSH messages are sent by A_c to inform the client what servers are attacked. Every client has a servList recording attacked servers (servList.h/servList.c). When getting PUSH messages, the client is going to update its servList. PUSH messages have md5 digest in it, for integrity checking. Others cannot pretend A_c to send PUSH

messages or pretend the client to send packets into AID tunnels as long as they do not

know the secret key shared between A_c and the client. These UDP packets do not go

further from here in the kernel. We mentioned no application level programs listening on

port 4369 earlier. The program *client* tells the kernel just drop them after it gets the

information of PUSH messages.

Packets from NF_IP_LOCALOUT

The program *client* processes all TCP packets leaving the client host. First, it

examines a packet's destination IP. If it finds a match in the servList (the destination

service is under attacks), it changes the packet's destination IP to A_c's IP and copy the

packet with new destination IP back to the kernel. The destination port is not compared

when searching a match in the servList. It is not necessary to distinguish different service

ports on an attacked server. With A_c's IP as destination, the packet is going to enter an

AID tunnel tree. If no match in servList, the program *client* just tells the kernel it did not

do anything to the packet and the kernel can continue passing on the packet.

Packets from NF_IP_POST_ROUTING

All TCP packets come here after passing the hook NF_IP_LOCAL_OUT. The

program *client* inserts a bunch of data into every packet as AID layer header. If a

packet's original destination is an attacked server, its destination IP we can see here is A_c's

IP. It was modified in the hook NF_IP_LOCAL_OUT. If the server is not under attacks,

we can see the server's IP as the packet's destination IP.

The program *client* inserts different AID layer headers into packets. If destination

IP was changed into A_c's IP, it means the packets are going to enter AID tunnels. As

shown in Fig. 4-1, application layer data is moved back 28 bytes, and an AID layer

header is put into that 28 bytes space. *Virtual clock timestamp* is initialized to the client's

local time. *Md5 digest* ensures A_c that the packets are really from the client and not

altered.
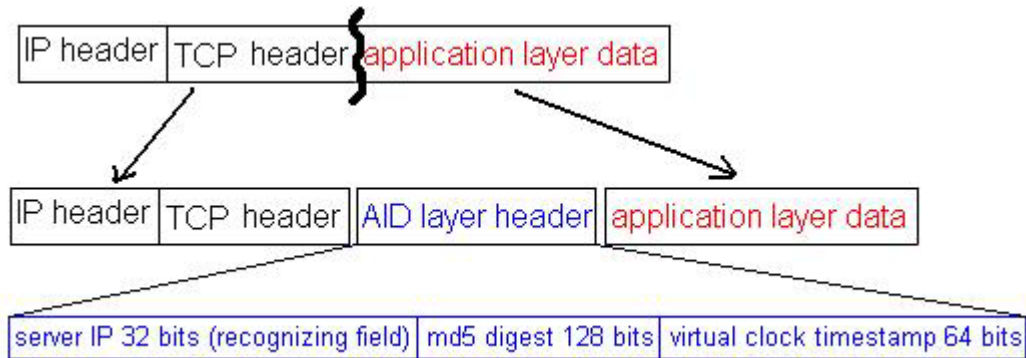
Figure 4-1. Inserting an AID layer header to a packet that enters AID tunnels. A 28-
byte-long AID layer header is injected.

For packets not entering AID tunnels, their destination IP is still the server's IP.

These packets do not need *md5 digest* and *virtual clock timestamp*. Nevertheless, they do

need the 4-byte-long *recognizing field*. Fig. 4-2 shows how this sort of packets is dealt

with. *Recognizing field* is an unsigned integer with value 0. We explained why we need

it clearly in the chapter **AID LAYER**.
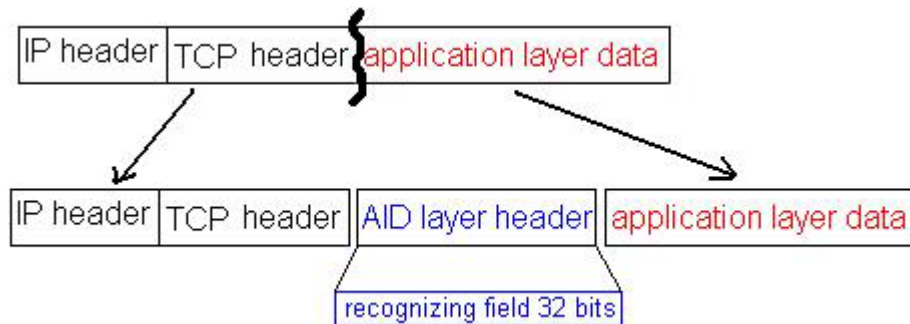
Figure 4-2. Inserting an AID layer header to a packet not entering AID tunnels. A 4-
byte-long AID layer header is injected.

Either packets going to AID tunnels or not, this hook is the final chance we can

modify them. After copying the modified contents back to the kernel, these packets are

sent out right away.

Implementation Issues

What Is ServList?

ServList is a simple list structure. It uses sequential search going through every list node to find a match. A servList node contains an attacked server's IP and port, but currently the port is not checked for a match in our AID system. A servList is updated by PUSH messages from A_c.

Why Changes a Packet's Destination in Hook NF_IP_LOCAL_OUT?

The program *client* used to change a packet's destination IP in the hook NF_IP_POST_ROUTING, but it did not work as we expected sometimes. For example, we have a server, an AID station and a client. Their IP addresses are 192.168.1.100, 192.168.1.101 and 192.168.1.103 respectively. Assume the server is under attacks, and the client sends a packet to the server. Before getting in the hook NF_IP_POST_ROUTING, the packet's destination IP is server's IP, 192.168.1.100. After leaving the hook but before really sent out, the packet's destination IP is the AID station's IP, 192.168.1.101. Then the packet leaves the client. Unexpected things happen here. The AID station does not get the packet, but the server gets it. If the server has FORWARD chain in iptables set well, the packet may be forwarded to the AID station. Anyway, it is not what we want. The packet should go to the AID station directly since we changed the packet's destination IP. We concluded that we should not change the packet's destination IP right before it leaves the host. We should do this in the hook NF_IP_LOCAL_OUT instead, and everything goes well.

How Is the Registration Done?

It is lots of work and difficult to make a complete secure system. Registration may be a secure hole in the system. In our AID system, clients get the secret key shared with

A_c by registration. We just took the easiest step here. The shared key was pre-configured in both the client and AID station, A_c. If users want to change the key, they need to redefine it in both sides, with the same value. Then recompile the codes. It is not hard. We wrote a makefile compiling and linking object files to generate the program *client*. It can be done by just one command. Clients also need A_c's IP address. It is not pre-defined in the codes. It is given as a command argument when users run the program *client*. It is possible to introduce public key system here, a better but complex way. We do not consider it currently.

Not Perfectly Isolated from Higher-Level Applications

Our goal is to make the AID system completely independent of higher-level applications. It means application level programs do not observe the existence of the AID service. Unfortunately, considering AID layer is not handled in the kernel and it is treated like application layer data, our AID system cannot be perfectly isolated from higher-level application. All outgoing TCP packets are inserted an AID layer header. When servers, which did not join the AID service, get these packets, their application programs will find extra junk data, the AID layer header we appending. Network communication follows protocols. The AID layer header is useless to the application programs. Protocols may be violated and the communication will fail. Before connecting servers that did not join the AID service, the module clientFilter.o should be unloaded first.

The Maximum Transmission Unit (MTU) Problems

Akin to the last issue, some more problems are caused by inserting an AID layer header that is not handled by kernel. We add 28 bytes into packets entering the AID tunnels and 4 bytes into the other packets. Is it harmless to enlarge a packet like this?

The answer is not always true. Usually, the OS tries to buffer enough data to form big packets to avoid small size packets by Nagle algorithm. Every packet has headers, if only a few data inside, the ratio of headers in a packet goes high and it is inefficient.

If we telnet or ssh a server, it may be fine. If we ftp or sftp a server, the kernel will try to buffer as many data as possible. Usually, a packet's size is as large as MTU. If we add an AID layer header in a packet in this case, the packet's size will be larger than MTU. The packet will be just dropped. We noticed this problem when testing ftp service in the AID system. There are two ways to solve the problem. One is to disable the Nagle algorithm, and the other is to strict the size of packets from higher-level applications. After opening a TCP socket, we have a chance to set socket options by calling the function setsockopt() in C library. We can pass in the option TCP_NODELAY to disable the Nagle algorithm or the option TCP_MAXSEG to change the maximum segment size for outgoing TCP packets. We chose the second way to keep the efficiency brought by the Nagle algorithm.

It is another cause that the AID system is not totally isolated from higher-level applications. Most application programs do not strict the size of outgoing TCP packets. They fully take the advantage of Nagle algorithm. If programs tend to buffer data used in the AID system, ftp clients for example, most of packets cannot be sent out because of the huge size. It is easy to fix by setting the socket option TCP_MAXSEG. However, the application programs need to be recompiled. It can be fixed as well if we handle the AID layer in the kernel, but we need to recompile kernel though.

CHAPTER 5
SERVER END

On the server end, it needs to tell where packets come from, the Internet or AID

tunnels. The server end also has to alert AID stations if it is attacked.

ServerFilter.c and server.c are two main source files for server ends. ServerFilter.c

is compiled as a module, queuing interested packets into userspace. Then, server.c takes

queued packets out and does whatever is necessary. After a server registers at an AID

station, A_s, it can get a secret key. The key is used to verify the communication between

the server and A_s are not modified by the third party. The server also keeps A_s's IP

address. If it is under attacks, A_s will be informed.

When a registered server is attacked, it will send PUSH messages to the AID

station, which the server registered at. This AID station called A_s. All UDP messages in

the AID system are sent to port 4369. However, there is no program listening on this

port. UDP packets to port 4369 are handled by the program *server*. We can characterize

occurrences of DoS attacks in several ways. In our AID system, we use arrival rates,

average incoming bytes per second, to determine if a server is attacked. Each server has

its capacity, 10000 bytes per second for example. When the arrival rate is higher than its

capacity, the server is under attacks. We can include other definitions of DoS attacks into

the AID system easily in our implementation.

Module ServerFilter.o

By compiling serverFilter.c, we can get the module serverFilter.o. It hooks on

handling functions at NF_IP_PRE_ROUTING, and NF_IP_POST_ROUTING. At

NF_IP_PRE_ROUTING, all incoming TCP packets are queued, except for the local traffic. Local traffic goes from loopback interface, 127.0.0.1, to loopback interface. At NF_IP_POST_ROUTING, only UDP packets to the port 4369 are queued. Other outgoing traffic is not related to the AID system.

Only incoming TCP packets are queued since currently only TCP traffic is protected. When the module serverFilter.o is loaded in a host, the host must run the program *server* as well. Otherwise interested packets keep getting into the queue, but no program takes them out of the queue. The traffic is blocked if this happens. A host should load the module serverFilter.o and run the program *server* at the same time. It is meaningless to do just one of them.

Program *Server*

By compiling server.c and linking other relative source files, we can get the executable program *server*. It has a while loop in main() whose condition is always true. The program *server* has two packet buffers, one for packets from the Internet and the other for packets from AID tunnels. The former is called bufferN and the latter is called bufferT. Because packets from AID tunnels have higher priority, the program *server* intends to handle packets in bufferT first. Let us see what the program *server* does to packets from different hooks.

Packets from NF_IP_PRE_ROUTING

In this hook, the program *server* has to remove the AID layer header from every queued packet. Before doing this, program *server* needs to know if the packet is from the Internet or AID tunnels. The program *server* inspects the *recognizing field*. If it is 0, the packet is from the Internet. If it is the server's IP (IP of the host runs the program *server*),

the packet is from the AID tunnels. If neither 0 nor the server's IP, the packet has wrong

contents and is dropped.

If the packet is from the Internet, it is put into the bufferN. It is dropped if bufferN

is full. The 4-byte-long *recognizing field* is removed from the packet.

If the packet is from AID tunnels, the whole AID layer header is 28 bytes long,

including the *recognizing field*, *md5 digest* and *virtual clock timestamp*. First, the

program *server* inspects the packet's integrity with *md5 digest*. If failing, drops the

packet. Then, the packet is put into the bufferT. Drops the packet if bufferT is full.

Similarly, the whole 28-byte-long AID layer header is removed from the packet in the

program *server*. Fig. 5-1 shows how an AID layer header is removed for an incoming

TCP packet.

A
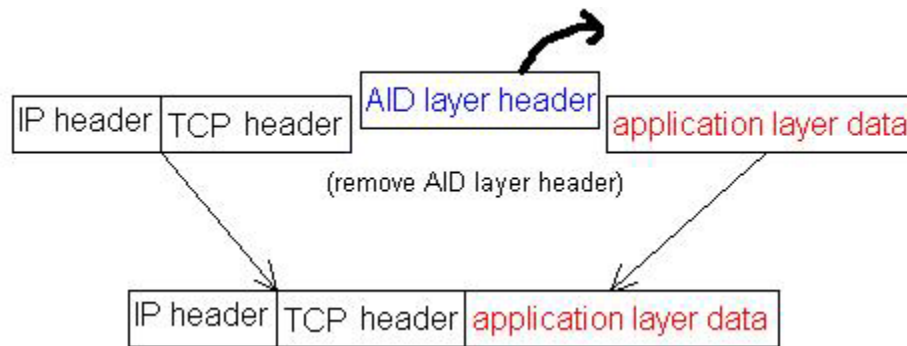
B
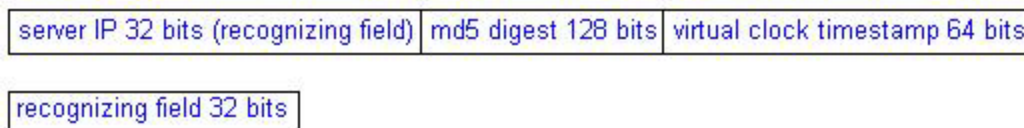
Figure 5-1. Removing the AID layer header in server end. A) All incoming TCP packets
should have an AID layer header. Other parts of a packet are not tainted. B)
A packet from AID tunnels has a 28-byte-long AID layer header; otherwise its
AID layer header is 4 bytes long.

The program *server* does exactly contrary things to what the program *client* dose to packets from hook NF_IP_POST_ROUTING. The program *client* adds AID layer headers on packets, and they are ripped out here. As a result, higher-level applications have no idea of the existence of AID layer. When they get a packet, they see no data of an AID layer header.

Packets from NF_IP_POST_ROUTING

All packets in the queue grabbed at hook NF_IP_POST_ROUTING are UDP traffic to port 4369. The only UDP message belongs the AID system (to the port 4369) that would be sent out by a server end is PUSH. PUSH messages are sent to A_s to say the server is attacked. Before leaving a server, these queued packets will be appended 16-byte-long md5 digest. It prevents the third party from forging PUSH messages and send them to A_s.

<div align="center">

Implementation Issues

</div>

No Threads

In the chapter **AID System Overview**, we pointed out no threads or child processes in programs *client*, *server* and *AID* for printing out statistic information under users' requests. Unlike the program *client*, the program *server* has one more thing to handle, sending packets in bufferN and bufferT to higher-level applications. The program *server* also has a while loop with a consistent true condition in main(). In every iteration, the while loop examines two things. First, sees if there are packets in bufferT and bufferN. A part of them are passed to higher-level applications. Second, sees if any packet was queued by the module serverFilter.o and read in one packet. Each of them takes only little time, so they look like running simultaneously. Threads make a program harder to

maintain and may cause serious problems like resource competition and deadlocks, if not used very carefully.

Important Variables

There are some important variables defined in the source file server.c. They decide the way the program *server* works. To make the program *server* work properly, they need to be assigned reasonable values. We discuss them below.

PCKBUFSIZET

BufferT's size, if too small, packets from AID tunnels will be dropped frequently with heavy incoming traffic. BufferT stores packets from AID tunnels.

PCKBUFSIZEN

BufferN's size, if too small, packets from the Internet will be dropped frequently with heavy incoming traffic. BufferN stores packets from the Internet.

IPQREADTIME

In the section **No Threads**, we said two things are done every iteration in the while loop of main(). One of them is to read in a packet queued by the module serverFilter.o if the queue is not empty. If the queue is empty, the program *server* is blocked until a packet is put into queue by the module serverFilter.o. If blocked, the packets in the bufferN and bufferT cannot be sent to their destination, higher-level applications. Consequently, we need to constrain the time of this reading behavior. Do not wait more than IPQREADTIME microseconds if the queue is empty. If it is larger than 500000, the client side may feel painful lags.

READINTERVAL

BufferT and bufferN are examined every iteration in the while loop of main(), and some packets in the two buffers are sent to higher level applications. We are not sure

how long one iteration may take. We may want packets stay in the buffers longer than the time of one iteration because we need to balance the traffic from AID tunnels and from the Internet (packets from AID tunnels have higher priority). It can be done by this variable. It defines how often packets in the two buffers are sent out. It is in microseconds, too. It cannot be too small or the program *server* cannot control the traffic. If the variable is too big, apparent lags appear.

SENDPCKBUFNO

Every READINTERVAL microseconds, packets in two buffers are taken out, but how many? The variable is the answer. If it is 10, the 10 packets from the two buffers can be sent out. Notice it is a total number for packets from both bufferT and bufferN. Since bufferT has higher priority, if 10 packets are picked up this iteration from bufferT, packets in bufferN have to wait until next iteration. If it is too small, the two buffers gets full easily. Packets will be dropped frequently when traffic is heavy.

AVGINTERVAL

The program *server* calculates the arrival rate every AVGINTERVAL seconds. If it is too small, the arrival rate may not be representative. If it is too big, the program *server* may not be able to detect attacks in real time (not sensitive enough).

TOTALCAP

Capacity of the server end, in our AID system, was defined as how many bytes per second the server end can handle. Once the arrival rate is higher than TOTALCAP, the program *server* alerts the AID system to create a tunnel tree by sending PUSH messages to A_s.

RESERVEDTIMES

When under attacked, a server has traffic from both the Internet and AID tunnels. We said the latter has higher priority, but how? The server handles data in bufferT and in bufferN with the ratio RESERVEDTIMES: 1. For instance, if TOTALCAP is 1000 bytes per second and RESERVEDTIMES is 4, $1000 \times 4/(1+4) = 800$ bytes per second is reserved for the traffic from tunnel trees, and $1000 \times 1/(1+4) = 200$ bytes per second is reserved for the traffic from the Internet.

How the Registration Is Done

In our AID system, servers get the secret key shared with A_s by registration. We just took the easiest step here. The shared key was pre-configured in both the server and AID station, A_s. If users want to change the key, they need to redefine it in both sides, with the same value. Then recompile the codes. It is not hard. We wrote a makefile compiling and linking object files to generate the program *server*. It can be done by just one command.

Not Perfectly Isolated from Higher-Level Applications

Same as the program *client*, the program *server* cannot be totally isolated from higher-level applications because the Linux kernel does not actually handle AID layer headers. AID layer headers are viewed as application layer data by the kernel. The program *client* inserts an AID layer header into a packet and the program *server* removes the AID layer header from the packet. It is a little confusing here. Is the program *client* different from other client programs like telnet and ssh? Our program *client* does not try to connect the server host. It takes care of queued packets in a client host instead. Likewise, the program *server* is not a server daemon program. It does not listen on a

port. Its job is taking care of queued packets in a server host. When a client host tries to

connect a server host, there are four possibilities:

- The client host has clientFilter.o loaded and is running the program *client*, and the server host has serverFilter.o loaded and is running the program *server* as well. It works just fine in this case because both sides can recognize the AID layer headers inserted in packets.

- The client host has clientFilter.o loaded and is running the program *client*, but the server host does not load serverFilter.o. It does not work in this case because the server host will get packets with AID layer headers from the client host, and the server host cannot recognize them. AID layer headers are junk data for the server host, which make communication fail.

- The client host does not load clientFilter.o, but the server host has serverFilter.o loaded and is running the program *server*. It does not work in this case either because the client host sends out packets without AID layer headers. When the server host gets the TCP packets from the client host, it tries to know if the packets are from AID tunnels or the Internet by inspecting the *recognizing field* in AID layer headers. Of course, these packets do not have the *recognizing field* and application layer data is used as *recognizing field*. Then, communication fails.

- The client host does not load clientFilter.o and the server host does not load serverFilter.o either. Both sides know nothing about AID layer headers. It works well. In this case, the AID service has nothing to do with both sides. Communication just goes as without the AID service protection as before.

In conclusion, if a client host wants to connect a server host that has serverFilter.o

loaded and is running the program *server*, the client host should load the module

clientFilter.o and run the program *client* before making a connecting. On the other hand,

if a client host wants to connect a server host that does not load serverFilter.o, the client

host should unload the module clientFilter.o and terminate the program *client* before

making a connection. The client host should match the server host to make everything go

well.

One thing worth a mention is that loading clientFilter.o and running the program

client do not mean the client host already joined the AID service. It should register at an

AID station to make the AID service effective first. Same thing applies to the server host

as well. However, an unregistered client host can still connect to a registered server host by loading clientFilter.o and running the program *client*. All packets from that client host cannot enter AID tunnels because the client has no secret key. They can only be transmitted via the Internet..

Loading or unloading the module clientFilter.o can be done by one command. A client host can adapt itself to different server hosts dynamically.

Program Alert

By compiling the source file alert.c and linking other relative source files, we can get the executable program *alert*. A server host can send PUSH messages to AID stations by executing the program *alert*. We leave the flexibility of defining DDoS attacks to the users. Users can define the situations of being attacked to meet their need. All they need to do is to run the program *alert* when the server host detects attacks. It will send A_s a PUSH message to trigger the AID system. Afterward, all packets from registered clients to that server go through the AID tunnels. The program *server* inserts Md5 digest into the PUSH message packets at hook NF_IP_POST_ROUTING.

CHAPTER 6
AID STATION

AID stations are the cores of our AID system [6]. They form an AID tunnel tree for each attacked registered server. How is a tunnel tree created? How are packets routed in a tunnel tree? How to resist attacks from registered clients? How was it implemented? Answers to the above questions are in this chapter.

AID Tunnel Tree

We have seen AID tunnels many times in the previous chapters. We know AID tunnels are tree structures. Packets from registered clients to the attacked registered servers would enter AID tunnels. In this section, we explained how an AID tunnel tree is constructed and how packets are routed inside. The push-n-pull process [6] establishes a tunnel tree from the registered clients to an attacked server.

Push Phase

Assume a server S is attacked; it sends a PUSH message to A_s, the AID station that it registered at. An AID tunnel tree for the server is going to be built up, and the tree's root node is A_s. The scenario is as follows.

1. Server S senses an attack and sends PUSH messages to A_s. A_s is the root node of the AID tunnel tree, which called the **first level node**.

2. A_s is the only AID station that knows S is under attacks so far. A_s picks up k other AID stations randomly, and sends a PUSH message to each of them. Any other AID station could be selected. Subsequently, $k+1$ AID stations know S is under attacks at the end of the step. We call these k AID stations the **second level nodes** in the tunnel tree. How big should k be? We have deep discussion about it later.

3. Every **second level node** randomly picks up k other AID stations, and sends a PUSH message to each of them. Any other AID stations could be selected except

for A_s. So, the k **second levels nodes** selected k^2 nodes totally. We call these k^2 nodes the **third level nodes** in the tunnel tree.

Notice that a node might be picked up more than once in the step 2 and in step 3 because both steps randomly select k AID stations. Fig. 6-1 shows how is a tunnel tree created in push phase.
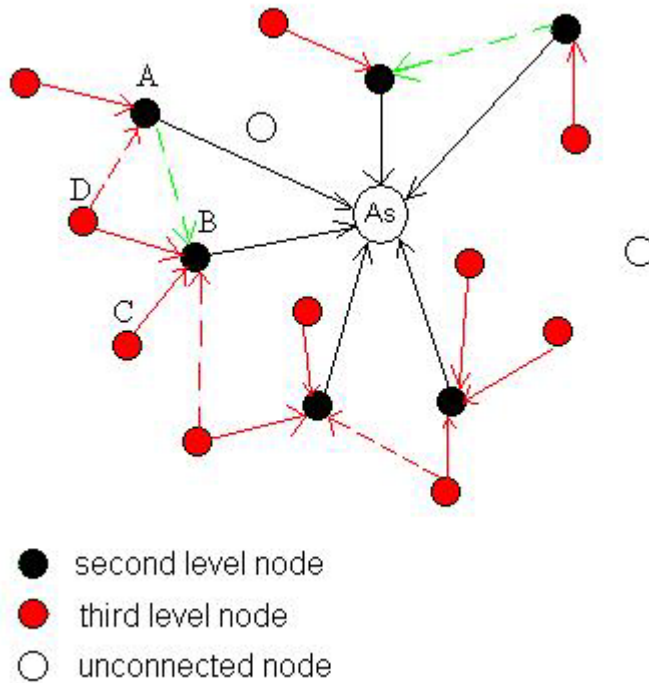
Figure 6-1. Tunnel tree created in push phase. Not all **third level nodes** are shown in the figure. Arrows from nodes to nodes indicates the direction packets would be routed. See the broken arrows in the figure. Node A got PUSH messages from both A_s and B. A would choose A_s as its parent node because of shorter routing path. Similarly, node D got PUSH messages from B and A. D could pick either of them to be its parent node, but not both. In push phase, there might be some AID stations not receiving PUSH messages, which are unconnected nodes in the figure.

Suppose we have N AID stations. We want to notify every AID station when a server is attacked. In step 1, only A_s is notified. In step 2, $k+1$ AID stations are notified. In step 3, ideally, $1+k+k^2$ AID stations are notified. If k is the square root of N, we get $1+k+k^2 > N$, which means the tunnel tree covers every AID station. However, some AID

stations picked in step 2 may be picked again in step 3 and some **second level nodes** may select the same **third level nodes**. We cannot guarantee every AID station is included in the tunnel tree. That is why we need pull phase. In push phase, $k(k+1)$ PUSH messages are sent out totally, because only the **first level node** and the **second level nodes** would send PUSH messages. If we allow the **third level nodes** to send PUSH message, push phase will be expensive. The majority of AID stations can be reached in push phase.

Pull Phase

When a server detects an attack, a tunnel tree rooted at A_s is built up. Some nodes might not get the PUSH messages in push phase. These nodes did not connect to the tunnel tree yet. We try to include them into the tunnel tree in pull phase. In pull phase an AID station will ask other AID stations by sending PULL messages what servers are attacked. In push phase, an AID station gets information of attacked server passively.

If an AID station B gets PULL messages from another AID station A, B will send PULLANS messages back to A. PULLANS messages contain information of all attacked servers B knows. When A gets these PULLANS messages from B, it will update its attacked servers recording. Actually, an AID station sends PULL messages to q other AID stations. Like the variable k in push phase, we should choose a proper q. We will discuss q and k later. Each AID station sends PULL messages out periodically.

Routing

A_s is the first AID station that knows the register server S is attacked, and A_s is also the root of the tunnel tree for the server S. When an AID station A tells another AID station B that server S is attacked by either PUSH or PULLANS messages, A becomes B's parent node in the tunnel tree for the server S. Hence, A_s is the parent node of the **second level nodes**, and the **second level nodes** are the parent nodes of the **third level nodes**.

The structure of a tunnel tree changes dynamically because of PULLANS messages. In pull phase, if a **third level node** gets PULLANS messages from A_s, the only **first level node**, it will switch its parent to A_s and become the **second level node.** Then, the routing path becomes one AID station shorter. A_s is the root node, all TCP packets to the server S are routed to A_s finally.

If an AID station gets a packet whose final destination is server S, where the packet is routed to next? The AID station routes the packet to its parent node. The **third level nodes** route packets to the **second level nodes**, and the **second level nodes** route packets to A_s. Packets go from leaf nodes to the root node in the tunnel tree. Finally, A_s forwards packets to the server S.

An AID station could be a tree node of more than one tunnel trees. If N servers are attacked, there will be N tunnel trees built up on the random overlay network (RON). One tree is independent from another.

Why Does a Tunnel Tree Try to Include Every AID Station?

Two reasons here, first, a client is free to register at an arbitrary AID station. Suppose server S is attacked and client C is trying to connect S. C registered at AID station A_c. A tunnel tree would be created for server S. If the tunnel tree does not embrace A_c, A_c does not know S is under attacks. In this case, C would not be informed by A_c that S is attacked. Thus, client C keeps sending packets to server S via the Internet. These packets are not protected even though C did register and joined the AID service. We do not constrain which AID station can have registered clients, so we tries to include all AID stations.

Second, it is about routing. AID station A routes packets to AID station B if B is A's parent node in the tunnel tree. A got PUSH or PULLANS messages from B before. B

also routes these packets to its parent node. If B crashes and then restores, it will lose information about its parent node of the tunnel tree. Now, B does not know where to route the packets from A. B sends PULL messages to others when getting a packet that it does not know where to route. If one PULLANS message B got contains information it needs, where to route packets to server S, B hooks on the tunnel tree again. However, it is possible that all PULLANS messages B got contain nothing about server S. If we choose right k and q, the chance that this happens is very low.

Variables k and q

Assume we have a set of n AID stations. AID stations A, B and A_s are elements of the set. In step 2 of push phase, A_s sends PUSH messages to other k AID stations. The probability that A does not get the PUSH message from A_s is $\left(\dfrac{n-1-k}{n-1}\right)$. Now, suppose AID station B got a PUSH message from A_s and is the **second level node**. In step 3, B would not send PUSH messages to A_s and itself, so B could send PUSH messages to other k nodes out of $(n-2)$. The probability that A does not get the PUSH message from B in step 3 of push phase is $\left(\dfrac{n-2-k}{n-2}\right)$. Since there are k **second level nodes**, the probability that A does not receive any PUSH messages in step 3 of push phase is $\left(\dfrac{n-2-k}{n-2}\right)^k$. A does not connect to the tunnel tree right after push phase if it obtained no PSUH message in step 1, 2 and 3. Finally, we got the probability that A is not covered by the tunnel tree after push phase is $\left(\dfrac{n-1-k}{n-1}\right)\times\left(\dfrac{n-2-k}{n-2}\right)^k$. So, the probability, called P_{InTree}, that an arbitrary AID station A could be in the tunnel tree after push phase is

$1 - \left(\dfrac{n-1-k}{n-1} \right) \times \left(\dfrac{n-2-k}{n-2} \right)^{k}$. The expectation number of AID stations included in the

tunnel tree after push phase is $n \times P_{InTree}$.

Theorem 1: if $n > 2$ and $k = \sqrt{n}$, then $P_{InTree} > 1 - \dfrac{1}{e}$.

$$\text{Proof: } 1 - \left(\frac{n-1-k}{n-1} \right) \times \left(\frac{n-2-k}{n-2} \right)^{k} > 1 - \frac{1}{e}$$

$$\Leftarrow \quad \left(\frac{n-1-k}{n-1} \right) \times \left(\frac{n-2-k}{n-2} \right)^{k} < \frac{1}{e}$$

$$\Leftarrow \quad \left(\frac{n-1-k}{n-1} \right)^{k+1} < \frac{1}{e}$$

$$\Leftarrow \quad \left(\frac{n-k}{n} \right)^{k} < \frac{1}{e}$$

$$\Leftarrow \quad \left(1 - \frac{k}{n} \right)^{k} < \frac{1}{e}$$

$$\Leftarrow \quad \left(1 - \frac{\sqrt{n}}{n} \right)^{\sqrt{n}} < \frac{1}{e}$$

$$\Leftarrow \quad \left(1 - \frac{1}{\sqrt{n}} \right)^{\sqrt{n}} < \frac{1}{e}$$

$\left(1 - \dfrac{1}{\sqrt{n}} \right)^{\sqrt{n}}$ is a monotonically-increasing function and is equal to $\dfrac{1}{e}$ when n is

approaching infinity. Hence, if $n > 2$ and $k = \sqrt{n}$, $P_{InTree} > 1 - \dfrac{1}{e}$. We use $k = \sqrt{n}$ in

our AID system.

Let us talk about variable q now. $\left| \Pi_{s} \right| = n \times P_{InTree}$ AID stations are expected to be

covered in the AID tunnel tree right after push phase and we know $n \times P_{InTree}$

$> n \left(1 - \dfrac{1}{e} \right)$ by Theorem 1. An AID station sends out PULL messages to q other AID

stations. The probability that at least one of these q AID stations is in the tunnel tree is

$$1-\left(\frac{n-1-|\Pi_s|}{n-1}\right)^q > 1-\left(1-\frac{n[1-1/e]}{n-1}\right)^q > 1-\frac{1}{e^q} \quad (1)$$

If $q = 10$, the probability is greater than 0.99995. It is high enough that we can almost say it will happen. When an unconnected AID station receives PULLANS from another AID station that is in the tree already, it connects to the tree. With push-n-pull process, the chance that all AID stations are included in the tree is very high.

Advantages of Random Overlay Network (RON)

First, small diameter and modest nodal degree: A good overlay network topology should have small diameter and nodal degree. Unfortunately, they conflict with each other. We made a tradeoff in our RON topology. We have a fixed small diameter that is three and modest nodal degree that is about the square root of the number of all AID stations. With small diameter, packets can arrive at servers by passing few AID stations. With modest nodal degree, we save some resource and keep the availability against node failure. We explain why the diameter is three in the end of this chapter.

Second, easy to set and maintain: A tunnel tree is established by sending PUSH and PULL messages. They are sent randomly by an AID station to other AID stations. We do not need a complicated algorithm to create the tree. Besides, every tree node just has to know who is its parent to forward packets. Not many things need to be remembered by an AID station. Capacity of the AID system can be increased by adding more AID stations.

Third, against node failure: If a tree node, an AID station, is down somehow, its children nodes are disconnected from the tunnel tree. However, the children nodes can hook on the tree again in next pull phase. Therefore, we can easily shutdown an AID station for maintenance without affecting the entire AID system. It is also true for adding

an AID station. A new-added AID station can connect the tree in pull phase as well. An

AID station can join and leave the AID service with little damage.

<div align="center">**Distributed Virtual-Clock Packet Scheduling**</div>

Basic Idea

Assume we have an attacked server *S* and a tunnel tree for *S*. Every AID station

maintains a virtual clock VC_u [11] (initialized to be the local system time) for every

tunnel *u* connecting with a client network. When an AID station gets a packet from

tunnel *u*, VC_u is updated as follows [6].

$$VCu = \max \{VCu, current\_time\} + T \times L \quad (2)$$

The AID station then marks the packet's virtual clock timestamp as *VCu*. All AID

stations' local clocks should be synchronized. *L* is the length of the packet. *T* is called

waiting interval. A_s broadcasts a new *T* to all AID stations periodically by sending

CTRLT messages. We use *T* to control the speed of a virtual clock. Since *T* can be

changed dynamically, we can adjust a virtual clock's speed dynamically as well. The

maximum rate a client can send data to server *S* via RON is 1/*T*. If an AID station gets a

packet from tunnel *u* connecting to another AID station, the packet has a timestamp on it

already.

An AID stations puts all incoming TCP packets into a buffer in ascending order

based on their virtual clock timestamps. When the buffer is full, the packet with largest

timestamp will be dropped. We call a packet's virtual clock timestamp VCTS. If *VCTS*

– "the AID station's local time" > α is true for an incoming packet, the packet is just

dropped, not put into the buffer even though the buffer is not full. The value of α can be

configured in the program. If a registered client hosts an attacker, its virtual clock will

run very fast because of huge amount of traffic. Most of packets from the client will be

dropped since their virtual clock timestamps are too big. In this way, server's capacity is shared fairly among all clients.

How to Adjust *T*

Server *S* reserves part of its capacity, called C_s, for RON. *T* is set to $1/C_s$ at first and broadcasted to all AID stations by A_s. There are two phases to adjust *T*. In each phase, new *T* is broadcasted to every AID station.

- Exponential phase: In this phase, A_s doubles the value of *T* to make virtual clocks run twice faster. When virtual clocks run twice faster, the maximum arrival rate of server *S* from RON is cut by half. A_s keeps in exponential phase until the arrival rate is below C_s. Then, A_s enters linear phase.

- Linear phase: Suppose *T* is changed from *I* to *2I* by the last update of *T* in exponential phase. In linear phase, A_s decrease *T* by $\varepsilon \cdot I$ periodically to slowdown virtual clocks until arrival rate is above C_s. We call the system converges at the moment. Then, A_s may enter exponential phase again.

Programs for an AID Station

AID stations route packets from registered clients, form a tunnel tree for an attacked registered server and control the traffic flows of tunnel trees.

AIDFilter.c and AID.c are two main source files for AID station. AIDFilter.c is compiled as a module, queuing interested packets into userspace. Then, AID.c takes queued packets out and does whatever is necessary. An AID station keeps information about its registered clients and servers. An AID station routes TCP packets in a tunnel tree, and uses UDP messages to control the AID system.

Module AIDFilter.o

By compiling AIDFilter.c, we can get the module AIDFilter.o. It hooks on handling functions at NF_IP_PRE_ROUTING, and NF_IP_POST_ROUTING. At NF_IP_PRE_ROUTING, all incoming TCP packets and UDP packets to port 4369 are queued, except for the local traffic. Local traffic goes from loopback interface, 127.0.0.1,

to loopback interface. At NF_IP_POST_ROUTING, all outgoing TCP packets and UDP packets to port 4369 are queued, except for the local traffic.

Queued TCP packets are traffic inside tunnel trees. They come from registered clients and head for registered servers. When the module AIDFilter.o is loaded in a host, the host must run the program *AID* as well. Otherwise interested packets keep getting into the queue, but no program takes them out of the queue. The traffic is blocked if this happens. A host should load the module AIDFilter.o and run the program *AID* at the same time. It is meaningless to do just one of them.

Program *AID*

By compiling AID.c and linking other relative source files, we can get the executable program *AID*. It has a while loop in main() whose condition is always true. The program *AID* has a packet buffer, storing incoming TCP packets.

TCP packets from NF_IP_PRE_ROUTING

Every queued incoming TCP packet in an AID station will go through the following processes.

- Verify the third party did not alter the packet. The packet is dropped if md5 digest stored in the packet is not equal to the one calculated by the AID station.

- Packet's virtual clock timestamp is refreshed as described in the section Distributed Virtual-Clock Packet Scheduling.

- Examine the packet's virtual clock timestamp. If *VCTS – "the AID station's local time"* $> \alpha$, the packet will be dropped. Constant α was defined as MAXVCTSEXCEED in AID.c. Most of offending packets are filtered out here.

- The AID station look up its routeList to know where to route the packet. Every AID station has a routeList, which is a list structure. A routeListNode contains information for a tunnel tree, inclusive of server's IP, distance to A_s, and the parent node's IP. An AID station may be embraced in more than one tunnel trees, and its routeList will contain more than one routeListNode. If The AID station does not know where to route the packet, no information for the destination server stored in

the routeList, the AID station will drop the packet and send PULL messages to other AID stations.

- If a packet passes all of the above and the packet buffer is not full, it can be put into the packet buffer. If the packet buffer is full, the packet with maximum virtual clock timestamp in the packet buffer is selected. The incoming packet and selected packet are compared in their virtual clock timestamps. If the incoming packet has smaller timestamp, it can replace the selected packet in the packet buffer; otherwise, it will be just dropped. The packet's destination IP is modified to the parent node's IP, since the packet will be routed to the parent node in the tunnel tree.

UDP packets from NF_IP_PRE_ROUTING

Every queued incoming UDP packet in an AID station will go through the

following process.

- Verify the third party did not alter the packet. The packet is dropped if md5 digest stored in the packet is not equal to the one calculated by the AID station.

- Recognize what kind of UDP message the packet is. In the chapter **AID Layer**, we said there are a couple of different UDP messages in the AID system. We can tell it by the packet's *packet type* field in the AID layer header. If it is a PULL message, the AID station sends whole information of its routeList in PULLANS messages to the asking AID station. It happens in pull phase. If it is a PULLANS message, the AID station updates its routeList with the data of the packet. It happens in pull phase. If it is a PUSH message, the AID station updates its routeList with the data of the packet and sends PUSH message to other AID stations. It happens in push phase. If it is a CTRLT message, the AID station updates the variable T for the specific tunnel tree. A CTRLT message contains IP of the server that the tunnel tree is for. See the chapter **AID Layer** for more details about different UDP messages. A UDP message has md5 digest. It cannot be forged without knowing the secret key.

TCP packets from NF_IP_POST_ROUTING

The AID station is going to forward every queued packet to its parent node. The

destination IP was changed to the parent node's IP in the hook NF_IP_PRE_ROUTING

already. Here, the program *AID* recalculates md5 digest because the packet's destination

IP and virtual clock timestamp were changed. Finally, the program *AID* computes the

checksums in the TCP header and IP header.

UDP packets from NF_IP_POST_ROUTING

Every queued UDP packet in this hook heads to port 4369. Md5 digest need to be calculated and inserted into every queued UDP packet. The checksums in the TCP header and IP header are recomputed in this hook. Afterward, the packets are ready to leave the AID station.

Implementation Issues

No Threads

In the chapter **AID System Overview**, we pointed out no threads or child processes in programs *client*, *server* and *AID* for printing out statistic information under users' requests. Besides getting a packet from the queue, the program *AID* has three more things to do, sending packets in the packet buffer to higher-level applications, sending out CTRLT messages and sending out PULL messages. The program *AID* also has a while loop with consistent true condition in main(). In every iteration of the while loop, four things are examined. First, sees if the packet buffer has packets waiting and passes some packets to higher-level applications. Second, sees if it is time to send out CTRLT messages. We can define how often an AID station can send out CTRLT messages. Third, sees if it is time to send out PULL messages. We can also define how often an AID station can send out PULL messages. Fourth, sees if any packet was queued by the module AIDFilter.o and read in one packet. Each of them takes only little time, so they look like running simultaneously. Threads make a program hard to maintain and may cause serious problems like resource competition and deadlocks, if not used very carefully.

Registration for Clients and Servers

As we said before, the secret keys shared with clients and servers are pre-configured in the codes. If an AID station wants to register a client, the client's information needs to be added into AID.c. It is also true for registering a server. The program AID has to be recompiled after registration, which can be done by one command.

Important Variables

There are some important variables defined in the source file AID.c. They decide how the program *AID* works. To make the program *AID* work properly, they need to be assigned reasonable values. We discuss them below.

PCKBUFSIZE

The packet buffer's size, if too small, incoming TCP packets will be dropped frequently with heavy traffic.

IPQREADTIME

We mentioned four things are done every iteration in while loop of main(). One is to read in a queued packet. If the queue is empty, the program might be blocked until a packet is queued by the module AIDFilter.o. If blocked, the packets in the packet buffer cannot be got by higher-level applications. Consequently, we need to constrain the time of this reading behavior. Do not wait more than IPQREADTIME microseconds if the queue is empty. If it is larger than 500000, the client side may feel painful lags.

READINTERVAL

The packet buffer is examined every iteration in the while loop of main(), and some packets are sent to higher-level applications. We are not sure how long one iteration may take. We may want packets stay in the packet buffer longer than the time of one

iteration. It can be done by this variable. It defines how often the packet buffer is checked. It is in microseconds, too.

SENDPCKBUFNO

Every READINTERVAL microseconds, packets in the packet buffer are taken out, but how many? The variable is the answer. If it is 10, the 10 packets from the packet buffer can be sent to higher-level applications. If it is too small, packets will be dropped frequently when heavy traffic.

NEARBYAID

The number of other AID stations that are known by this AID station. PUSH, PULL, and CTRLT messages are sent to these neighbors.

SNEDTINTERVAL

If the AID station is A_s, a root node of a tunnel tree, it sends CTRLT messages to all other AID stations every SENDTINTERVAL seconds.

SENDPULLINVAL

Every SENDPULLINVAL seconds, an AID station sends PULL messages to other PULLNO (defined in global.h) AID stations. Periodically sending out PULL messages can make sure every AID station connects tunnel trees.

DECREASERATIO

In linear phase, A_s decreases T by $\varepsilon \cdot I$. DECREASERATIO is ε.

MAXVCTSEXCEED

If a packet's $VCTS$ – "AID station's local time" > MAXVCTSEXCEED, the packet is dropped. MAXVCTSEXCEED is the constant α.

Adding New AID Stations

A new added AID stations can be included into an AID tunnel tree by sending PULL messages to others. However, we still need to make the AID station known by all other AID station. Like handling registrations, an AID station pre-configures its neighbors in AID.c. When a new AID station is added into the AID system, its neighbor's AID.c needs to be updated and recompiled. It can be improved in a better but complex way.

Diameter of a Tunnel Tree

Remember the *AIDNO* field of a PUSH message and *distance* field of a server list node of a PULLANS message? Actually, they two mean the same thing, the distance to A_s of a tunnel tree. We explain why our random overlay network's diameter is three here. Every tree node, an AID station, records its distance to A_s. For A_s itself, the distance is 0. For the **second level nodes**, it is 1. For the **third level nodes**, it is 2. Suppose we have tree node A with distance of 1, tree node B with distance of 2 and tree node C not included in the tree yet. C has two ways to join the tunnel tree.

- Another node sends a PUSH message to C. It could be root A_s or node A. If C gets PUSH messages from A_s, C's distance to A_s is 1. If from A, C's distance to A_s is 2. Notice only the root node and the **second level nodes** can send PUSH messages or push phase becomes expensive (more than $k(k+1)$ PUSH messages sent out).

- C gets PULLANS messages from another AID station. It could be root node A_s, node A or node B. If C gets PUSH messages from A_s, C's distance to A_s is 1. If from A, C's distance to A_s is 2. If from B, C's distance to A_s is 3.

Assume we have another node D not included in the tree. D can join the tree by PUSH or PULLANS from node A or node B, but not node C. D ignores PULLANS message from the nodes with distance of 3. If D joins the tree by C's messages, C becomes D's parent node. That means D has distance of 4 to A_s, which is not allowed.

If *C*'s distance is 3 and it gets PULLANS or PUSH messages from *A*, *C* will switch

its parent node to *A* to have smaller distance, 2. C becomes the **third level node** after

switching. An AID station's distance to A_s can only go smaller. By doing this, no cycle

appears in a tunnel tree. As a result, distance between A_s and every other tree nodes is

not larger than 3. Actually, we can reset the diameter by redefining PUSHDEEP in file

global.h. A tunnel tree's diameter is PUSHDEEP+1. There are four possibilities of a

packet being routed from a registered client to a registered server, shown in Fig. 6-2.

```
1. server ⟵ As(Ac) ⟵ client
      -1        0          1

2. server ⟵ As ⟵ 2nd AID station(Ac) ⟵ client
      -1      0           1                  2

3. server ⟵ As ⟵ 2nd AID station ⟵ 3rd AID station(Ac) ⟵ client
      -1      0           1                  2                3

4. server ⟵ As ⟵ 2nd AID station ⟵ 3rd AID station ⟵ 4th AID station(Ac) ⟵ client
      -1      0           1                  2                 3                 4
```
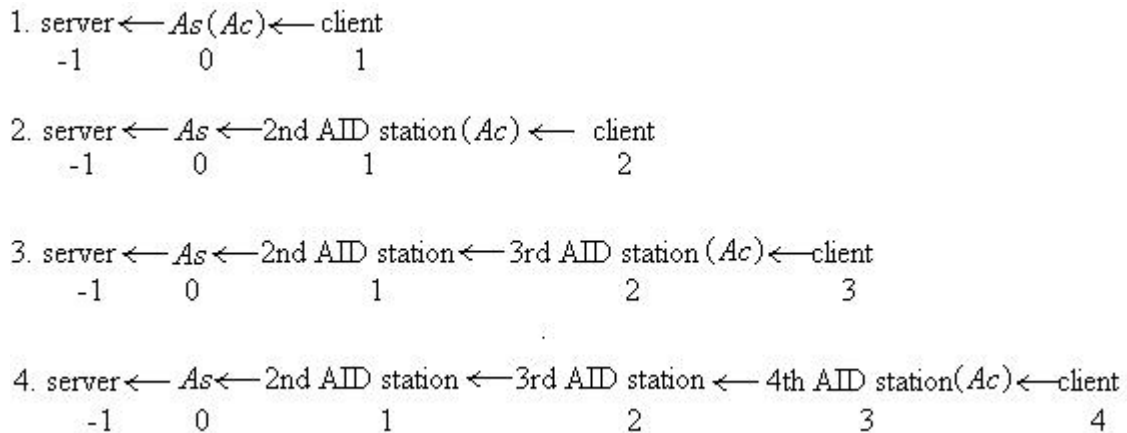
Figure 6-2. Four possibilities a packet can be routed. The numbers are the value of *AIDNO* field of PUSH messages or *distance* field of PULLANS messages sent by the host. Packets are routed toward A_s in an AID tunnel tree.

Forwarding Packets

Pay attention to the word "forwarding." An AID station works like a router

somewhat. Packets in RON are forwarded to A_s by AID stations. The difference is

normal routers do not adjust md5 digest or virtual clock timestamp of a packet. Usually,

the forwarding function is turned off in a Linux machine by default. It has to be on.

After version 2.4, the tool iptables is available in Linux. We use it to set forwarding rules

in an AID station. To be simple, no sophisticated rules are used. We just allow all kind

of forwarding. Of course, we can set more secure and elaborate forwarding rules. An

AID station is not allowed to send out its own TCP packets. It can only forward TCP

packets.

CHAPTER 7
TESTING RESULTS AND ANALYSIS

After going through the previous chapters, we understand how the AID system

works theoretically and practically. In this chapter we talk about how we tested our AID

system and analyze testing results. When an idea is transformed into real programs,

unexpected problems show up always. We already discussed some of them in

Implement Issues sections of previous chapters. The other practical problems are

illustrated in this chapter.

Important Issues about Testing

- The root access is required to load a module. A client host, a server host or an AID station needs to load clientFilter.o, serverFilter.o and AIDFilter.o respectively. We do not have enough Linux machines with the root access for tests. As a consequence, we just tested our AID system on three Linux machines, for a client host, a server host and an AID station separately. We might not test some functionality well with such a simple model.

- Since only one Linux machine is for client hosts, we have to simulate n client hosts on it by running n client processes. The AID station treats TCP packets from different source ports as they are from different hosts, even though they have the same source IP.

- A client process uploading a huge file to the server symbolizes an attacker. However, normal ftp programs cannot be used because of MTU problems. Packets sent out by a normal ftp client could be as big as MTU. There is no space for the AID layer header. We discussed this problem in the chapter **CLIENT END** in detail. Hence, we wrote two programs for this purpose, *testServer* and *testClient*. The server runs the program *testServer* to accept connections, and the client runs the program *testClient* to dump data to the server. In the program *testClient*, we can restrict the size of packets sent to *testServer* by resetting the socket option. If the packet size is smaller, *testServer* will get more packets (but same amount of application layer data).

- What we want to see from the testing are how fast the AID system converges, how T (waiting interval) changes, how many packets from legitimate users are dropped,

how many packets from attackers are dropped, how *T* affects the average arrival rate, and etc. Many factors can influence the above behaviors, for example the packet buffer's size. Most of these factors are malleable variables in the codes.

Testing Elements

We have five testing cases. Client Linux machine had several processes of the program *testClient* to simulate more than one client ends. Each process of the program *testClient* might be given different parameters. Given parameters decided if a client end was a legitimate user or an attacker.

Program *TestClient*

Usage of program *testClient* is

  testClient testServerIP blockSize blockNO MAXSEG sleepTime

TestClient is the filename of the executable. *TestServerIP* is the IP address of the host that runs the program *testServer*. *TestClient* will dump data there. The remaining four parameters are more meaningful. There is a for loop, which runs *blockNO* iterations in the program *testClient*. In every iteration a block whose size is *blockSize* is sent to *testServer*. It indicates the size of application layer data, not the whole packet. Because of headers, more than *blockSize* bytes are sent out in an iteration. With *blockSize* and *blockNO*, *testClient* knows how many application layer data it needs to send out, which are $blockSize \times blockNO$ bytes. The parameter *MAXSEG* defines the maximum size of TCP packets. Notice that the size of the whole packet, including IP header, will be a little bit bigger than *MAXSEG*. We need a suitable *MAXSEG* to save enough space for an AID layer header. The last parameter, *sleepTime*, defines how many seconds the program *testClient* sleeps before running the next iteration.

Program *TestServer*

TestServer accepts connection requests from *testClient* and prints out application

layer data sent by *testClient*.

Setting of the AID System

We have three Linux machines, one client, one AID station and one server. Table

7-1 shows the basic settings we used for testing. We explained what these factors mean

in previous chapters. Server's capacity was 2000 bytes per second. 1600 bytes per

second of it was reserved for traffic from AID tunnels. The setting was fixed during

testing but the number of attackers and legitimate users varied. The attacking modes

changed in different testing cases too.

Table 7-1. List of important factors of the AID system for testing

Name	Value
PCKBUFSIZET	50
PCKBUFSIZEN	50
IPQREADTIME	300000
READINTERVAL	300000
SENDPCKBUFNO	3
AVGINTERVAL	10
TOTALCAP	2000.0
RESERVEDTIMES	4.0
MAXVCTSEXCEED	4
DECREASERATIO	0.1

Case 1

There were two registered attackers. Their parameters were:

- Attacker1: *testClient 192.168.1.102 1000 500 1000 0*
- Attacker2: *testClient 192.168.1.102 1000 500 1000 0*

Fig. 7-1 shows how many packets were dropped because of big VCTS at the AID

station. MAXVCTSEXCEED is 4. Fig. 7-2A shows how variables T, I and *decrease*

changed their values. T was changed from I to $2I$ by the last update of T in exponential

phase and then entered linear phase. In linear phase, A_s decreased T by *decrease*

periodically to slow down virtual clocks. *Decrease* is equal to DECREASERATIO times *I*. Fig 7-2B shows the arrival rate at the AID station and the server. *AvgT* is the arrival rate of the tunnel tree at the server. *AvgN* is the arrival rate of the Internet at the server. *AvgAID* is the arrival rate at the AID station. All of them are average received bytes per second in the 10-seconds period. '*AID cap*' is part of server's capacity that was reserved for the registered clients. '*Server cap*' is server's whole capacity. In the third 10-seconds, *avgN* exceeded server's capacity, and the AID system was triggered. Two attackers started to send packets via RON, instead of the Internet. We can see *avgN* went down and finally to 0. Now, we examine Fig. 7-2A and Fig. 7-2B together. The AID system converged after the twelfth 10-seconds. *T* ranged between 0.000875 and 0.00175 when converging. When *T* went down, *avgAID* went up. *T* kept decreasing to *I*, in linear phase, until *AvgAID* was bigger than '*AID cap*'. At the moment, the AID system entered exponential phase again. When *T* doubled, *avgAID* declined dramatically. When *avgAID* became smaller than '*AID cap*', the AID system entered linear phase. The AID system prevented *avgAID* from exceeding '*AID cap*' by tuning *T*. If no new attackers joined, after the AID system converged, *T* would fall into a fixed range as we can see in Fig. 7-2A. Table 7-2 shows how many packets and why they were dropped. About 2/3 of incoming packets were dropped.

Table 7-2. Case 1 packets statistics in the AID station. A packet is dropped when the AID station's packet buffer is full, *VCTS – "AID station's local time" > α*, integrity checking fails or the AID station does not know where to route the packet.

	Attacker1		Attacker2	
Packet# in	1294		1230	
Packet# out	487		456	
Packet# dropped	BufferFull	0	BufferFull	0
	BigVCTS	807	BigVCTS	774
	MD5Fail	0	MD5Fail	0
	CantRoute	0	CantRoute	0
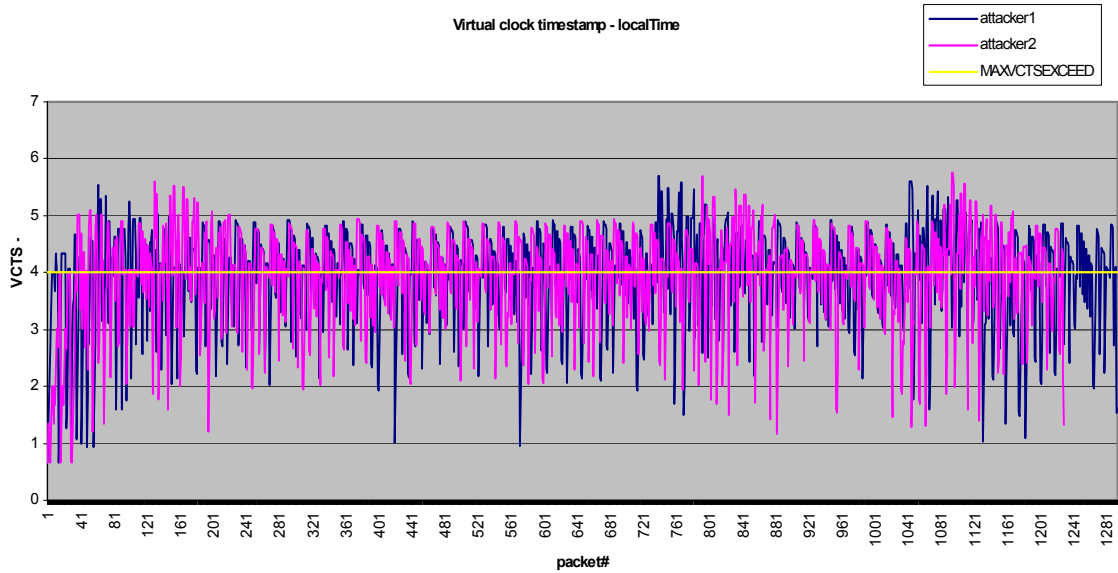
Figure 7-1. Distribution of incoming packets in case 1 at the AID station. Packets lay above the straight line MAXVCTSEXCEED were dropped.
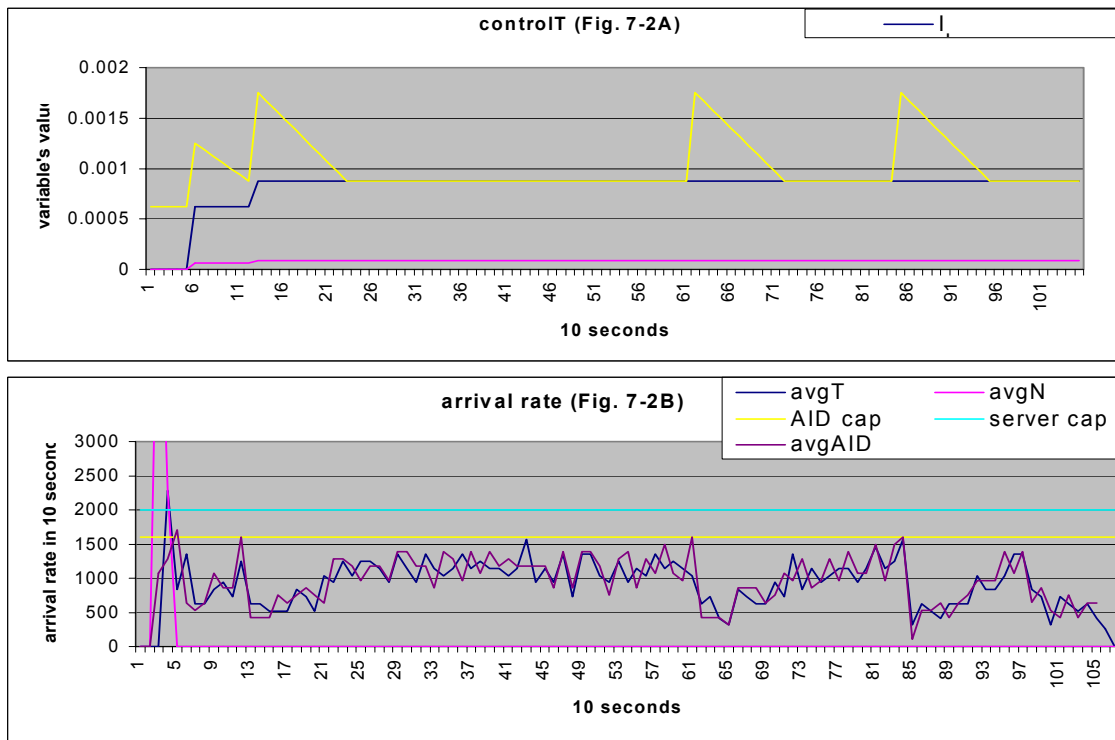
Figure 7-2. How did *T* and arrival rate interact with each other in case 1. A) After system converged, *T* pulsed in a fixed range. B) When *avgAID* exceeded '*AID cap*', *T* doubled. Then *avgAID* fell again. Most of time, *avgAID* was below the yellow straight line, showing the arrival rate was controlled well.

Case 2

Similar to case1, however, we added one legitimate registered client:

- Attacker1: *testClient 192.168.1.102 1000 500 1000 0*
- Attacker2: *testClient 192.168.1.102 1000 500 1000 0*
- Normal_user: *testClient 192.168.1.102 250 700 250 1*

The normal user was distinguished from two attackers in three ways. First, it sent smaller amount of data. Second, the size of packets from it was smaller. Third, it slept 1 second every iteration. T ranged between 0.001125 and 0.00225 when converging. It was bigger than in case 1 because we had three registered clients here. In Fig. 7-3, in the same time period, normal_user sent about twice packets as many as an attacker did because TCP had flow control mechanism. After many packets were dropped; attackers would slow down their traffic. In Fig. 7-3, we know packets from the legitimate was really safe because *VCTS - localTime* was in the range of {0.5, 1} approximately, which was far away from 4. Normal_user was influenced by T much less harshly than attackers were. Likewise, Fig. 7-4A and Fig. 7-4B show how the AID system quelled arrival rate by adjusting T. Table 7-3 shows packets statistics of case 2.

Table 7-3. Case 2 packets statistics in the AID station. Normal_user had no packets dropped.

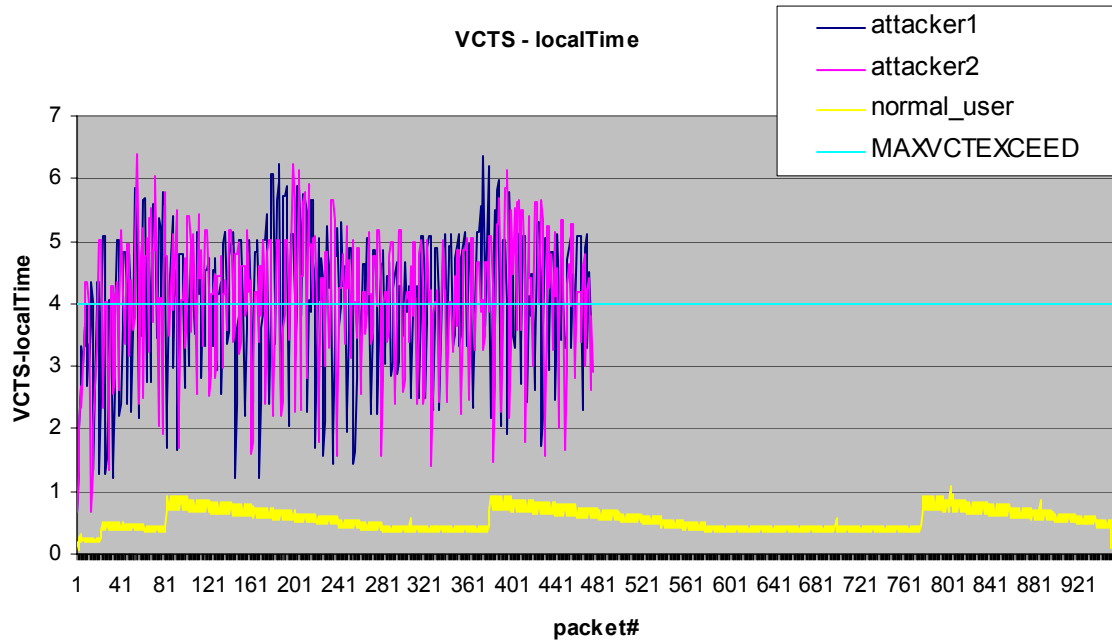|	Attacker1		Attacker2		Normal_user	
Packet# in	473		474		952	
Packet# out	173		172		952	
Packet# dropped	BufferFull	0	BufferFull	0	BufferFull	0
	BigVCTS	300	BigVCTS	302	BigVCTS	0
	MD5Fail	0	MD5Fail	0	MD5Fail	0
	CantRoute	0	CantRoute	0	CantRoute	0

Figure 7-3. Distribution of incoming packets in case 2 at the AID station. Traffic from normal_user was pretty stable (the lowest series).
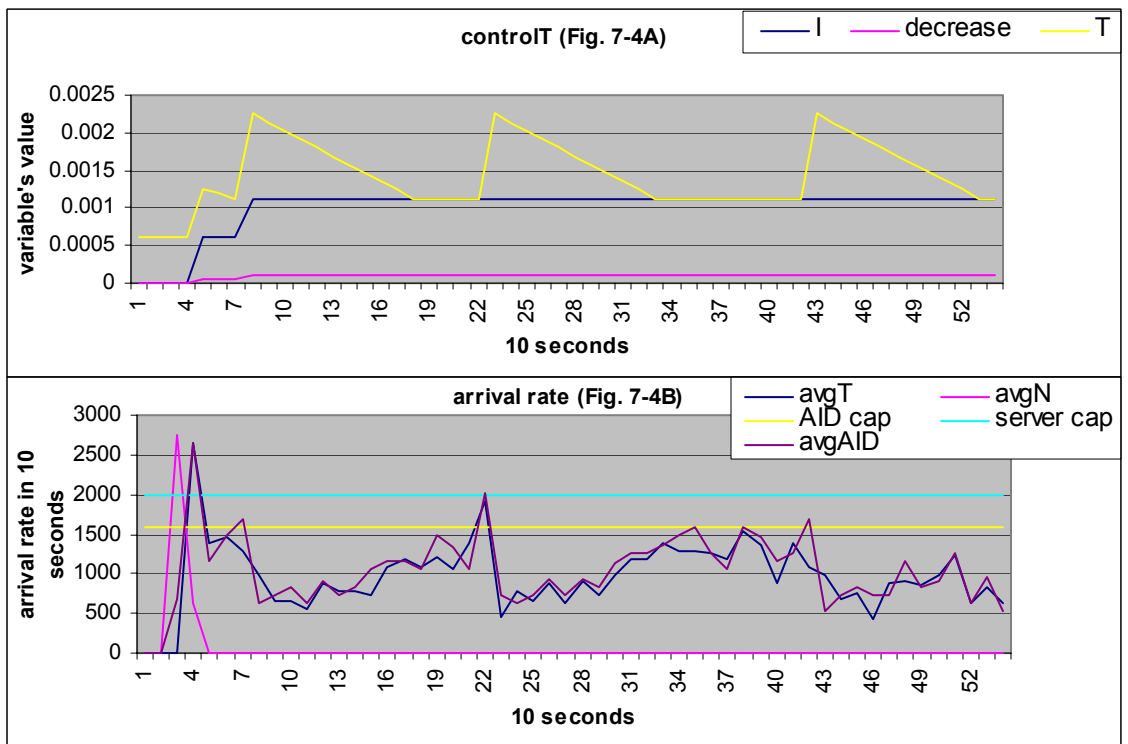

Figure 7-4. How did T and arrival rate interact with each other in case 2. When T doubled, *avgAID* and *avgT* fell.

Case 3

There were four registered attackers, and two registered legitimate clients. Notice

that two legitimate clients sent out different amount of data with different packet sizes.

Their parameters were:

- Attacker1: *testClient 192.168.1.102 1000 500 1000 0*
- Attacker2: *testClient 192.168.1.102 1000 500 1000 0*
- Attacker3: *testClient 192.168.1.102 1000 500 1000 0*
- Attacker4: *testClient 192.168.1.102 1000 500 1000 0*
- Normal_user1: *testClient 192.168.1.102 250 700 250 1*
- Normal_user2: *testClient 192.168.1.102 100 700 100 1*
  T ranged between 0.00175 and 0.0035 when converging. It was even bigger than in

case 2 because we had six registered clients here. Normal_user1, having exactly the

same parameters as normal_user in case 2, had packets dropped. In case 2, normal_user

had no packets dropped. What made the difference to the clients with the same

parameters? When the traffic load in the AID system is heavier (T goes bigger), packets

have higher chances to be dropped even though they are not from hosts intending to

attack. That is because the AID system tries to make a fair share of resource among all

registered clients. If a client sends more, it has to wait longer for next sending. Packets

from normal_user2 had small enough VCTS, so none was dropped.

Table 7-4. Case 3 packets statistics in the AID station. Normal_user1 had more packets
dropped, but it also had more incoming packets.

	Attacker1	Attacker2	Attacker3	Attacker4	Normal_user1	Normal_user2
Packet# in	411	357	388	366	1106	1215
Packet# out	116	97	106	97	636	1215
Packet# dropped	BufferFull = 0	BufferFull = 0	BufferFull = 1	BufferFull = 0	BufferFull = 0	BufferFull = 0
	BigVCTS = 295	BigVCTS = 260	BigVCTS = 281	BigVCTS = 269	BigVCTS = 470	BigVCTS = 0
	MD5Fail = 0	MD5Fail = 0	MD5Fail = 0	MD5Fail = 0	MD5Fail = 0	MD5Fail = 0
	CantRoute = 0	CantRoute = 0	CantRoute = 0	CantRoute = 0	CantRoute = 0	CantRoute = 0

Normal_user1 had dropped packets, but it was still distinguished from other true

attackers. First, its traffic was not slowed down as much as attackers. In the same time

period, an attacker just sent out about 375 packets, but normal_user1 sent out 1106

packets (normal_user2 sent out 1215). Second, in Fig. 7-5 we can see *VCTS-localTime*

for packets from normal_user1 rippled around 4. However, it is 6 for packets from

attackers. In conclusion, normal_user1 had packets dropped, but it still maintained its
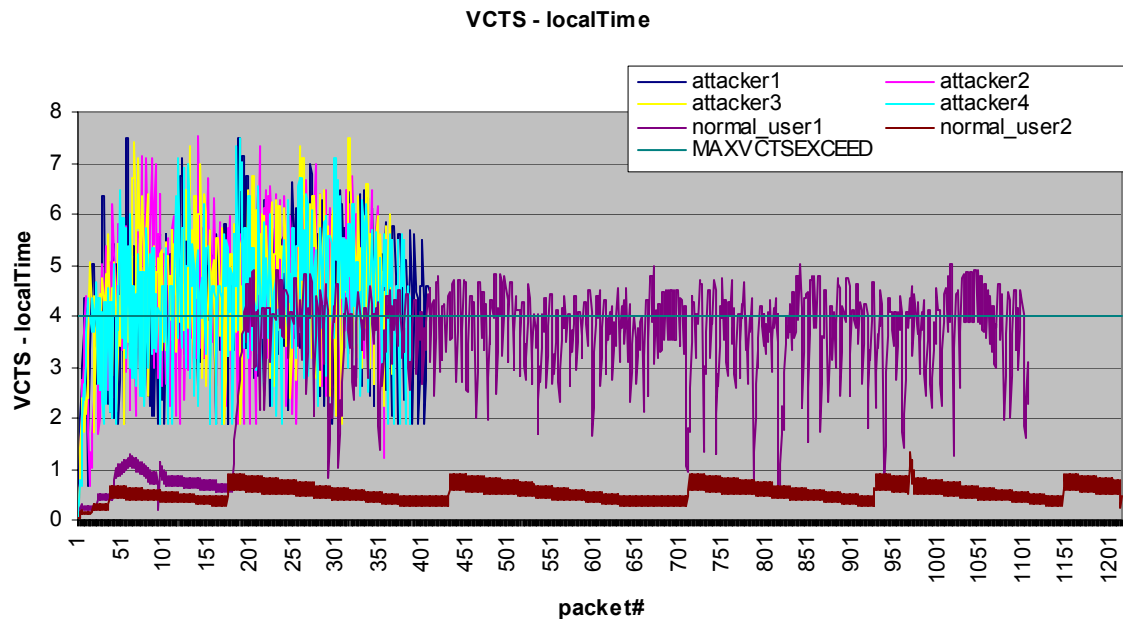
communication with the server.

Figure 7-5. Distribution of incoming packets in case 3 at the AID station. Traffic from normal_user2 was pretty stable (the lowest series). Even though some of packets from normal_user1 were discarded, it was still different from real attackers.
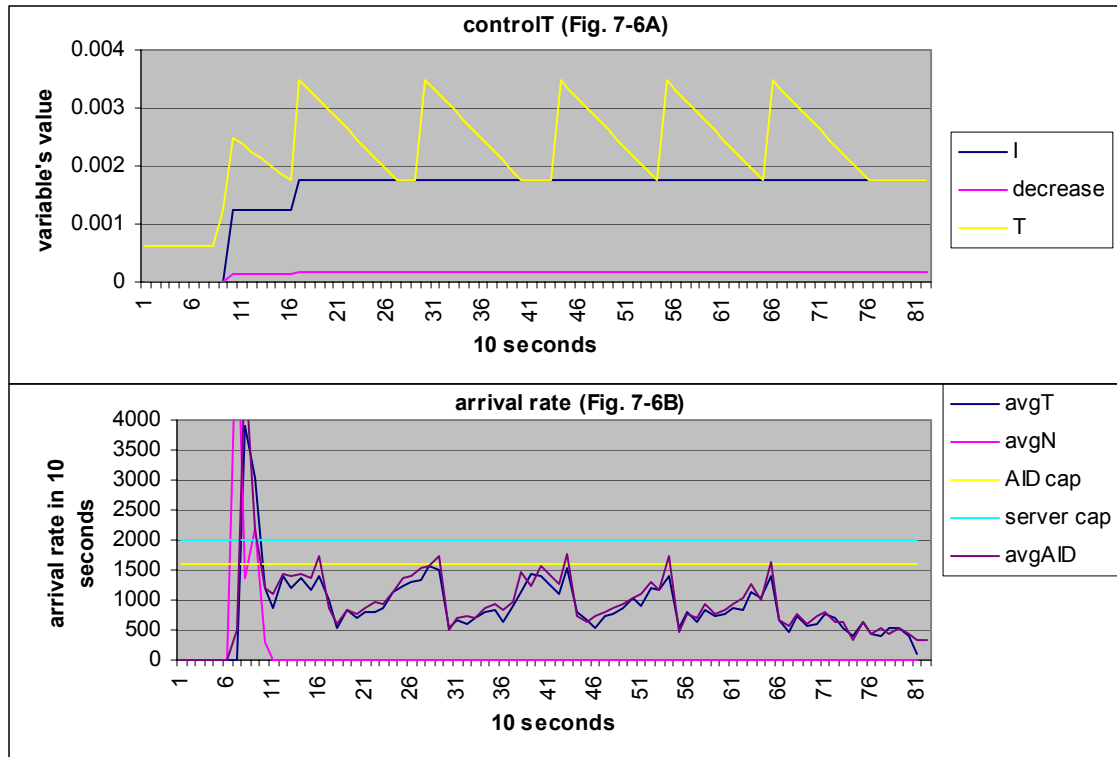
Figure 7-6. How did *T* and arrival rate interact each with other in case 3. When *T* doubled, *avgAID* and *avgT* fell; otherwise *avgAID* and *avgT* rose.

Case 4

There were two registered attackers, two registered normal users and two

unregistered normal users. Their parameters were:

- Attacker1: *testClient 192.168.1.102 1000 500 1000 0*
- Attacker2: *testClient 192.168.1.102 1000 500 1000 0*
- Normal_user1: *testClient 192.168.1.102 250 700 250 1*
- Normal_user2: *testClient 192.168.1.102 250 700 250 1*
- Normal_user3: *testClient 192.168.1.102 250 700 250 1*
- Normal_user4: *testClient 192.168.1.102 250 700 250 1*
  T ranged between 0.00125 and 0.0025 when converging. In Fig. 7-7, we can see

incoming packets distribution of two registered normal users and two registered attackers.

Packets from the other two unregistered normal users did not enter tunnel tree. As a

result, the AID station had no statistics data about them. In this testing case *avgN* did not

become zero after the AID system was triggered. Only the two registered attackers had

packets dropped. Ideally, *avgT*:*avgN* = 1600:400 = 4:1 should be true in the case 4 (this

ratio can be changed by modifying RESERVEDTIMES in server.c and signing a new

contract between the server and AID station). However, because attackers slowed down

their traffic (less than half amount of packets sent out as other normal users), *avgT* went

down too.

Table 7-5. Case 4 packets statistics in the AID station. Normal_user1 and normal_user2
are registered had no packets dropped.

	Attacker1		Attacker2		Normal_user1		Normal_user2	
Packet# in	493		453		1187		1194	
Packet# out	172		154		1187		1194	
Packet# dropped	BufferFull	0	BufferFull	0	BufferFull	0	BufferFull	0
	BigVCTS	321	BigVCTS	299	BigVCTS	0	BigVCTS	0
	MD5Fail	0	MD5Fail	0	MD5Fail	0	MD5Fail	0
	CantRoute	0	CantRoute	0	CantRoute	0	CantRoute	0

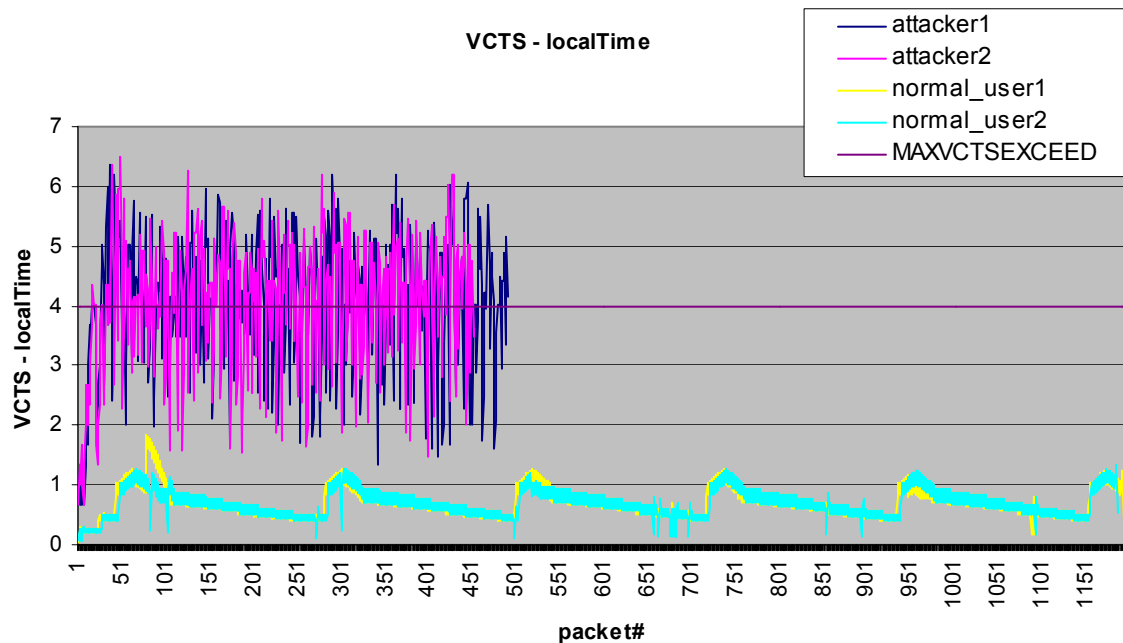Figure 7-7. Distribution of incoming packets in case 4 at the AID station. Traffic from
normal_user1 and normal_use2 were pretty stable (the lower two series). It is
very similar to Fig 7-3 with the exception that series for two normal users are
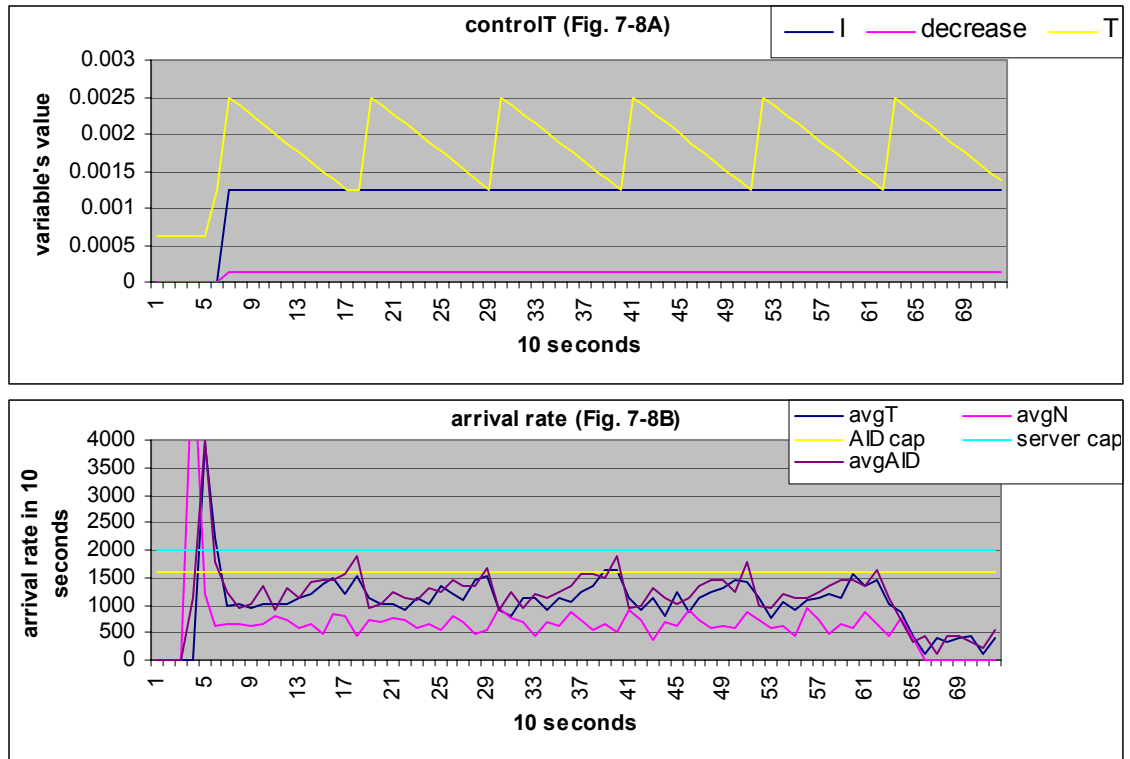a little bit higher.

Figure 7-8. How did *T* and arrival rate interact with each other in case 4. *AvgN* is the
server's arrival rate of the traffic from the Internet. *T* would not affect *AvgN*
directly, since the traffic did not enter the AID tunnels. However, because the
traffic from AID tunnels had higher priority, the unregistered attackers could
not flood the server.

Case 5

This is an interesting case. It is very analogous to case 2, two attackers and one

normal user. However, attackers chopped same amount of data into smaller pieces here.

- Attacker1: *testClient 192.168.1.102 1000 500 300 0*
- Attacker2: *testClient 192.168.1.102 1000 500 300 0*
- Normal_user: *testClient 192.168.1.102 250 700 250 1*
  In case 2, *MAXSEG* was 1000 for attackers, and it was 300 in case 5. In Fig. 7-10,

we can see that after system converged, value of *T* varied between 0.001375 and 0.00275.

It is bigger than in case 2. What made case 5 so special? Let us compare it with case 2.

In case 2, attackers sent out 1000 bytes, exclusive of headers, every iteration, and

MAXSEG was 1000. Here, attackers also sent out 1000 bytes per iteration, but *MAXSEG*

was 300. That means an iteration needs to send packets as many as three times in case 5. Then, what happened? See Fig. 7-9.

First, unlike in case 2, an attacker almost sent out as many packets as normal user did. A packet's VCTS is decided by its size and *T* of the tunnel tree. When a packet is small, VCTS will be small too. Consequently, the packet has a higher chance to be accepted by an AID station.

Second, compared with case 2, *T* became bigger when system converged, but *VCTS-localTime* for packets from attackers became smaller. Smaller packets made smaller VCTS but more packets sent out (heavier traffic) in an iteration made bigger *T*.

Third, Since *T* became bigger and packets from normal_user had the same *MAXSEG* as in case 2, 250, we can see *VCTS-localTime* for packets from normal_user twisted a lot, unlike in case 2.

In our AID service, packets from either attackers or legitimate users might be dropped. Because a server's capacity is fixed, if more clients try to access the server at the same time, every client could get less resource from the server. If a client intends to use more than its share, its packets will be discarded. The AID system controls the arrival rate not to surpass a server's capacity by this policy. We can see in case 2 and case 3. A client is treated differently when the traffic load changes. A legitimate client slows down its outgoing TCP traffic when sensing its packets were dropped (no acknowledgement from the other end), if it implemented TCP correctly. For an attacker, if it implemented TCP right, it would slow down outgoing traffic. If it did not, VCTS for its packets would grow very fast, and most of its packets would be dropped. Damage

from attackers is soothed in both cases, and at the same time, a server is still accessible to

legitimate clients.

Table 7-6. Case 5 packets statistics in the AID station. Normal_user had no packets
dropped.

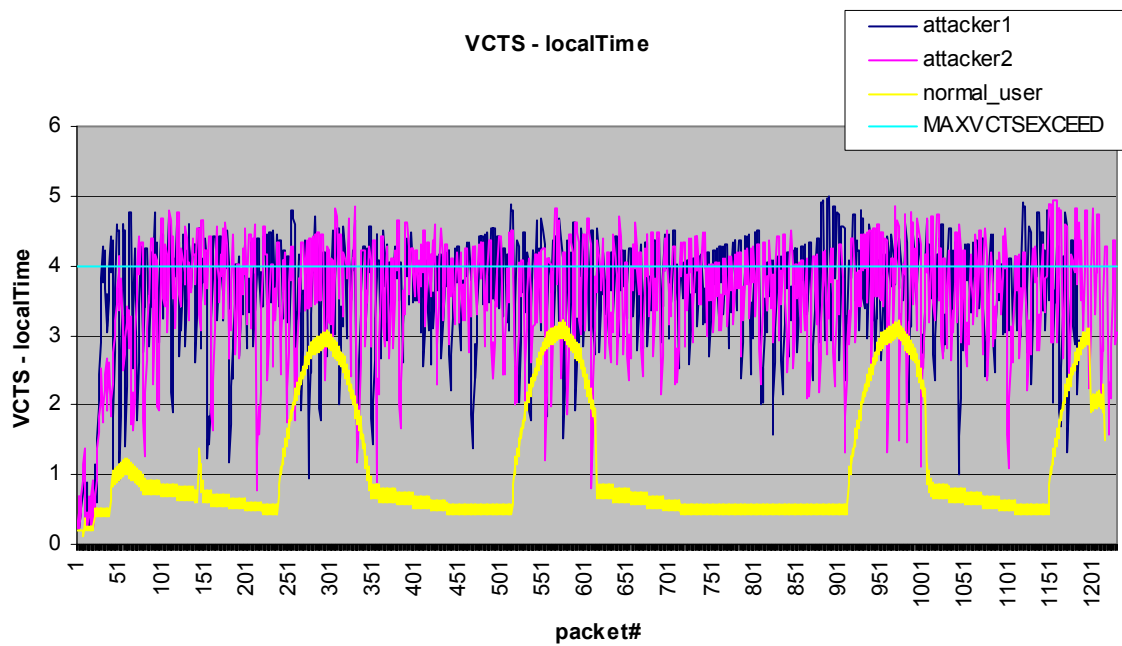|	Attacker1		Attacker2		Normal_user	
Packet# in	1229		1185		1215	
Packet# out	723		659		1215	
Packet# dropped	BufferFull	1	BufferFull	3	BufferFull	0
	BigVCTS	505	BigVCTS	523	BigVCTS	0
	MD5Fail	0	MD5Fail	0	MD5Fail	0
	CantRoute	0	CantRoute	0	CantRoute	0

Figure 7-9. Distribution of incoming packets in case 5 at the AID station. Notice the big
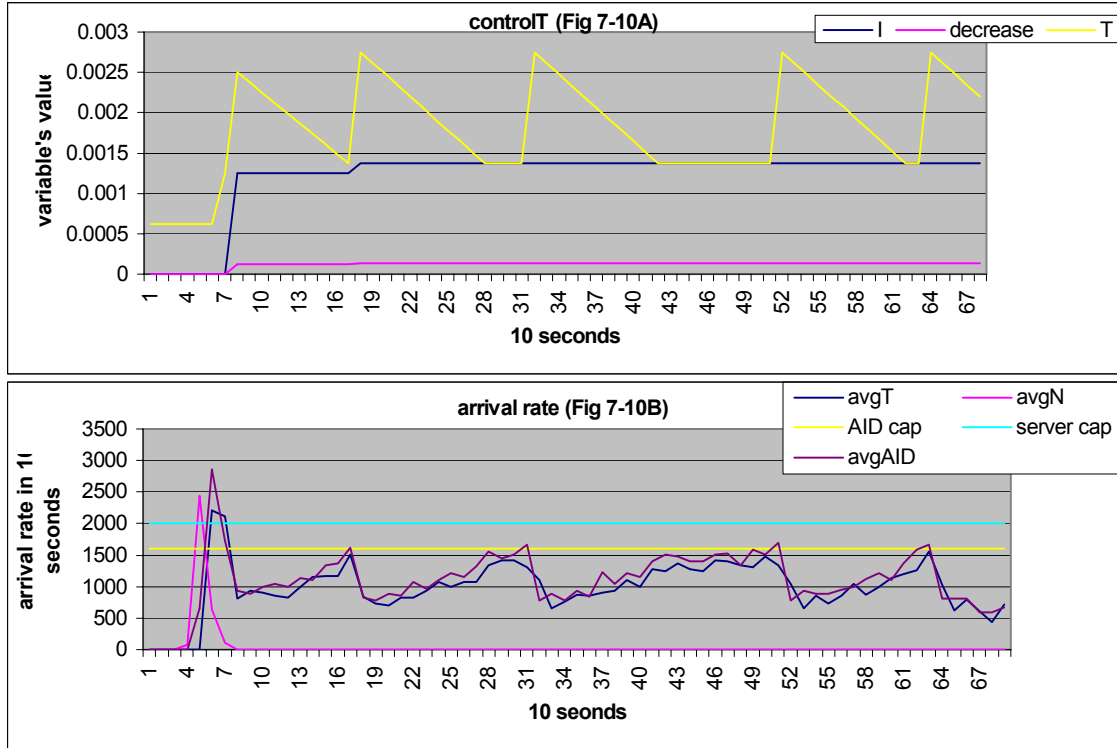"wave" of normal_user.

Figure 7-10. How did *T* and arrival rate interact with each other in case 5.

CHAPTER 8
FUTURE WORK AND CONCLUSION

Limitations and Future Work

Our AID system has some limitations theoretically and practically. Improving they is our goal in future work.

Protecting UDP traffic: We need the self-adaptation feature based on TCP congestion control to separate legitimate users and attackers. That is why our AID system only protects TCP traffic at present. Future work is to include UDP traffic into our AID service.

Robustness against the compromise of AID stations: In the current design of our AID system, we did not address how to deal with the case that AID stations are compromised. A compromised AID station can send forged UDP messages (PUSH, PULL, CTRLT and etc.), drop packets from legitimate users and adjust virtual clock maliciously. The good thing is an AID station can be removed or added into the AID system easily. We could disconnect a suspicious station for an inspection

Traceback: The AID system can resist against DoS attacks but cannot trace back to the origin of attacks. Flooding traffic might be from registered clients that are zombies remotely controlled by real attackers. We may implement existing IP traceback mechanisms in the AID system.

Independency of higher-level application: Our AID system is not perfectly independent of higher-level applications because we introduced AID layer and it is not processed by the Linux kernel. However, we also do not want users to recompile their

Linux kernels to join the AID service. We may find some other way to program our AID system to avoid the dilemma.

Flexibility of programs: Most controlling factors are defined as constants in the programs. We need to recompile them if we want to do registration, change the secret keys, adjust virtual clock setting and etc. These factors can be saved in files to make our programs more flexible.

Conclusion

Most existing defense systems for DoS attacks are not self-complete. They usually need universal deployment. In the thesis a self-complete anti-DoS service (AID) was implemented and tested. The AID service can be applied to Internet services, such as ssh, ftp, www and so on. Everyone can join the AID service by registration and get immediate protection. The AID service provides a fair share of a server's resource to all registered clients. It requires no modification of end systems and routing infrastructure to join. Random overlay network accommodates an efficient and scalable structure to route traffic from registered clients. It changes dynamically. An AID station can be removed or added easily. Distributed virtual-clock packet scheduling algorithm blocks the traffic from attackers and manages the arrival rate of a server. A registered client host, which is not an attacker, can access a registered server even when the server is attacked. Finally, we still have some problems need to be solved in the future, for example, including UPD traffic into the AID service, making AID station robust, tracing back attackers and having programs more flexible.

APPENDIX A
HOW TO RUN

Make sure the library *libipq* was installed before continue. It can be found in

iptables-1.2.9.

Program *server*:

- make server
- cd src_module
- make serverFilter.o
- insmod ip_queue
- insmod serverFilter.o
- cd ..
- ./server AIDSIP SERVERIP
  AIDSIP is IP of the AID station A_s. SERVERIP is the server's IP.

Program c*lient*:

- make client
- cd src_module
- make clientFilter.o
- insmod ip_queue
- insmod clientFilter.o
- cd ..
- ./client AIDIP
  AIDIP is IP of the AID station A_c.

Program *AID*

- make AID
- cd src_module
- make AIDFilter.o
- insmod ip_queue
- insmod AIDFilter.o
- cd ..
- ./AID clientIP serverIP

ClientIP is IP of the registered client and serverIP is IP of the registered server. In our testing cases, we had only one client machine, AID station and server machine. If an AID station wants to register more than one client or server, information of the client/server should be added into the function initialize() of the source file AID.c.

Program *alert*

- make alert
- ./alert AIDIP serverIP serverPort
AIDIP is IP of AID station A_s. ServerIP:serverPort identifies the attacked service.

Remove loaded module:

- rmmod ip_queue
- rmmod clientFilter
- rmmod AIDFilter
- rmmod serverFilter

Turn on forwarding in iptables at an AID station:

- su -
- echo "1" > /proc/sys/net/ipv4/ip_forward
- iptables –I FORWARD – j ACCEPT

APPENDIX B
FILE GLOBAL.H

File global.h defined many important constants. Their names and values we used

in testing are:

- **#define MTU 1500**: Max transfer unit.

- **#define BUFSIZE 4096**: The size of the buffer that a queued packet is copied to.

- **#define TCPINFOSIZE 28**: The length of an AID layer header, in byte long, for TCP packets entering a tunnel tree. The whole packet looks like: IP header | TCP header | new destination IP (32 bits) | packet digest for integrity (128 bits) | virtual clock timestamp 64 bits | application data. So, $(32 + 128 + 64)/8 = 28$ bytes.

- **#define UDPINFOSIZE 17**: The length of an AID layer header, in byte long, for UDP packets. The whole packet looks like: IP header | UDP header | packet digest for integrity (128 bits) | *packetType* 8 bits | application data. So, $(128+8)/8 = 17$ bytes.

- **#define MD5DGSIZE 16**: Bytes of md5 digest created by md5 library, md5.h and md5.c.

- **#define RCZSIZE 4**: Bytes of *recognizing field* of normal TCP packets.

- **#define UDPMAXSIZE 256**: The max size in bytes of a UDP packet in the AID system, inclusive of md5 digest (16 bytes), *packetType* (1 byes), service list (7*n bytes). It should be big enough for different UDP messages.

- **#define UDPTYPELEN 1**: Length of the *packetType* field in a UDP packet in byte long.

- **#define PULLNO 0**: Number of the nearby AID stations this AID station should send PULL messages to (variable q).

- **#define PUSHNO 0**: Number of the nearby AID stations this AID station should send PUSH messages to (variable k). Using square root of NEARBYAID (defined in AID.c) is ok.

- **#define PUSHDEEP 2**: how many AID station a PUSH message can go through, exclusive the first one which is A_s.

- **#define PULL 0**: Value of *packetType* field for a PULL message.

- **#define PULLANS 1**: Value of *packetType* field for a PULLANS message.

- **#define PUSH 2**: Value of *packetType* field for a PUSH message.

- **#define CTRLT 3**: Value of *packetType* field for a CTRLT message.

- **#define PCKTYPEOFFSET 16**: The offset of *packetType* in UDP packets, the location is: UDP header 8 bytes | md5 digest 16 bytes | *packetType* 1 byte =16

- **#define VCSTAMPOFFSET 20**: The offset of virtual clock timestamp in TCP, the location is: TCP header (offset bytes) | serverIP 4 bytes | md5 digest 16 bytes | VCTStamp 4 bytes, 4+16 = 20

- **#define DATAOFFSET 28**: The offset of application data in TCP packets, the location is: TCP header (offset bytes) | serverIP 4 bytes | md5 digest 16 bytes | VCTStamp 8 bytes | application data, 4+16+8 = 28

LIST OF REFERENCES

1.   C. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni, "Analysis of A Denial of Service Attack on TCP," *Proc. of IEEE Symposium on Security and Privacy*, pp. 208-223, IEEE Computer Society Press, Oakland, CA, May 1997.

2.   J. Lemon, "Resisting SYN Flood DoS Attacks with A SYN Cache," *Proc. of USENIX BSDCON2002*, pp. 89-97, USENIX Association, Berkeley, CA, February 2002.

3.   P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing," *IETF, RFC 2267*, January 1998.

4.   K. Park and H. Lee, "On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets," *Proc. of ACM SIGCOMM' 2001*, vol. 31, pp. 15-26, August 2001.

5.   H. Wang, D. Zhang, and K. G. Shin, "SYN-dog: Sniffing SYN Flooding Sources," *Proc. of 22nd International Conference on Distributed Computing System (ICDCS'02)*, pp. 421-428, IEEE Computer Society, Washington D.C., July 2002.

6.   S. Chen, R. Chow, Y. Xia and Y. Ling, "A Global Anti-DoS Service Based on Random Overlay Network," unpublished paper, Department of Computer and Information Science and Engineering, University of Florida, September 2004.

7.   A. Juel and J. Brainard, "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks," *Proc. of Network and Distributed System Security Symposium (NDSS'99)*, pp. 151-165, Networks and Distributed Security Systems, San Diego, CA, February 1999.

8.   T. Aura, P. Nikander, and J. Leiwo, "DoS-Resistant Authentication with Client Puzzles," *Cambridge Security Protocols Workshop 2000. LNCS, Springer-Verlag*, vol. 2133, pp. 170-177, 2001.

9.   D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," paper presented at *10th Annual USENIX Security Symposium*, Washington D.C., August 2001.

10. X. Wang and M. K. Reiter, "Defending Against Denial-of-Service Attacks with Puzzle Auctions," *Proc. of IEEE Symposium on Security and Privacy*, pp. 78-92, IEEE Computer Society, Washington D.C, May 2003.

11. L. Zhang, "VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks," *ACM Transactions on Computer Systems*, vol. 9, no. 2, pp. 101-124, May 1991.

BIOGRAPHICAL SKETCH

I earned my BS degree in computer science and information engineering from National Chiao Tung University in Taiwan. As an undergraduate, I was like a sponge to absorb all kinds of knowledge of computer science accessible to me. In my junior and senior years, I worked on a project of providing QoS (quality of service) in wireless ATM network. Integer programming is the key of the project. Through those four years, I finished many projects in different areas: network, security, compiler, windows programming, audio processing, graphics, database system, etc. I am seeking my MS degree at the University of Florida by writing the thesis. After about two years' training ar the University of Florida, I polished my professional knowledge and skills better and became more confident to face new challenges.