APPROVAL SHEET

Title of Thesis: Experimental evaluation and implementation of the Spread Identity Framework

Name of Candidate: Nikhil Dinkar Joshi
Master of Science, 2011

Thesis and Abstract Approved: _____

Alan T. Sherman
Associate Professor
Department of Computer Science and Electrical Engineering

_____

Dhananjay Phatak
Associate Professor
Department of Computer Science and Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name:	Nikhil Dinkar Joshi.
Address:	4750, Westland Blvd, Baltimore, MD-21227.
Degree and date to be conferred:	MS in Computer Science, Dec 2011.
Date of Birth:	Nov 30, 1985.
Place of Birth:	Maharashtra, India.
Collegiate institutions attended:	University of Maryland, Baltimore County, MS Computer Science, 2011. University of Pune, BE Computer Engineering, 2007.
Major:	Computer Science.
Professional positions held:	Oct 2007 - Dec 2007. Junior Game Programmer, Gameloft Pune. *Jan 2008 — July 2008.* Member of Technical Staff, Great Software Lab, Pune.

ABSTRACT

Title Of Thesis: Experimental evaluation & implementation of Spread identity Framework.

Directed By: Nikhil Dinkar Joshi, MS Computer Science, 2011

Prof. Alan T. Sherman, Associate Professor,
Department of Computer Science and Electrical Engineering

Prof. Dhananjay Phatak, Associate Professor,
Department of Computer Science and Electrical Engineering.

We provide a novel implementation of the Spread Identity (SI) framework and experimentally evaluate it on an organizational level. Unlike previous implementations of SI, ours builds on RFC2663 and RFC2694, which describes approaches for application-level gateways for domain name systems and twice network address translation, respectively. We implement a minimal system suitable for a medium-sized organization and measure connection establishment time, network address translation (NAT) table size, name resolution time, packet forwarding rate, and memory requirements. We find that our implementation consumes similar resources to those of a typical single NAT system.

Dhananjay Phatak, et al. first proposed SI in 2005. The central idea is to spread identity of a host across multiple network addresses through dynamic mapping of IP addresses to host names, thereby obfuscating the host's true network identity to some degree. One application is to mitigate IP packet floods. Using the ns-2 network simulator, in 2010

Sonawane experimentally demonstrated the effectiveness of SI as a defense against packet flood attacks. By contrast, our work evaluates the performance of our new implementation of SI on a real network.

Using the Deterlab test network, we experimentally compare our SI implementation with a single NAT subsystem, with both systems using NetFilter for NATting. We generate 100,000 simultaneous connections to test the baseline NAT and SI systems (a typical load for medium-sized organizations). Except for name resolution, both systems perform similarly on all measured parameters. In particular, we find there is no difference between end-to-end connection establishment time, up to 90 us difference in gateway packet translation time. We also find that NAT table size cannot be considered in isolation: NAT table entries are a part of connection tracking entries which require 304 byte entries per connection for both our implemented baseline and SI systems. In SI, name resolution takes up to 1500ms in the worst case, compared to 50ms in the baseline system. From these results, we conclude that our system is technically feasible and results in similar load patterns on the gateway to those from a typical single NAT system.

Experimental evaluation and implementation of Spread Identity Framework

By

Nikhil Dinkar Joshi

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County, in partial fulfillment
of the requirements for the degree of
MS Computer Science
2011

ACKNOWLEDGEMENT

First, I thank my adviser, Alan Sherman, for all his valuable suggestions. During the course of my research, he has been a great source of inspiration, motivation and, support.

I also want to thank my co adviser Dhananjay Phatak for providing helpful suggestions during this thesis work. It would not have been possible without his guidance.

I would like to thank Chintan Patel for graciously agreeing to be on my thesis committee.

Last but not the least, thanks to Kadlecsik Jozsef and other members of Net filter Core team, and Rob Sherwood for their valuable input on various aspects of Netfilter/iptables software and various simulation platforms available to carry out this experiment.

Thanks to the managing staff of the Deter lab test bed for responding to various queries/bugs.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The goal of a denial of service attack is to consume resources of a system, resulting in denial-of-service to legitimate consumers of service. Distributed Denial of Service (DDoS) attacks make use of malware to control clients spanning various geographic areas. These compromised end user machines are called zombies or bots. The master can order these bots/zombies to attack or flood targets as and when required. To avoid detection of 'masters' a level of indirection is maintained between 'zombies' and 'masters' using IRC (Internet Relay Chat) channels or other similar methods. There are many types of DDoS attacks each of which targets different aspects of the victim's system. J Mircovic *et al* provide a taxonomy of methods and types of defenses against DDoS attacks [1] [4]. Common to most DDoS attacks, is the bot-network which is a network of infected machines that can simultaneously query a victim server.

As described by Dhananjay Phatak *et al* [2] [3], the Spread Identity (SI) Framework works by deliberately and dynamically spreading identity of a host or a client across one or more IP addresses. This spreading of identities is achieved implicitly by modifying the name resolution process. As a result, the usual one-to-one association between a client or a host and its IP address becomes difficult to establish. A machine which does not possess any clear and static public network address is hard to target. Moreover, if such a machine uses different links for different network addresses, efforts required to snoop over the communication increase as the number of links increase. One important aspect of Spread Identity is dynamic association of IP addresses with hosts or clients. This association is on a per host-client basis, and can be finer grained to be unique per session, which makes it dynamic and hard to guess.

We describe how to implement Spread Identity at an organizational (small - medium) level. This implementation builds on well known techniques that are published by IETF as informational RFCs. Our implementation borrows from techniques and ideas published in Twice Network address translation (RFC2663) and DNS Extension to Network Address Translation (RFC 2694).

The SI implementation we provide uses some major aspects of IP filtering and some lesser aspects of overlay network schemes viz. indirection. The placement of the system can be classified as host/company centric, and can be optionally deployed at client side. It can also be deployed at higher (ISP) levels. This system can also be classified as preventive and defensive. Detection methods can be included to further improve the effectiveness of this system.

NAT has its own share of criticism and solutions which are not a part of this thesis for further information on issues (and corresponding solutions) caused by deployment of NAT please refer to RFC 2993 and RFC 3027. In generic terms, deployment of NAT causes problem with any protocol that uses IP address or attributes derived from IP addresses (signatures) on levels above the Network Level. Alternatives such as Realm Specific IPs (RSIP) which are currently in experimental stage may be further studied to preserve the end-to-end principle. RSIP has been noted to be an intermediate solution, and requires infrastructure change.

Our implementation although based on well-known RFCs, uses open source applications and software libraries that were not originally designed to perform as per the RFCs. There are many different NAT application, components that can be used to implement this system, each such combination of components will have a different performance. To deal with this problem, we compare performance of our SI implementation with baseline performance of these application and software libraries. We consider the baseline performance as the performance of a simple Single NAT installed at the gateway. We also argue that if this solution is able to provide results with no appreciable difference than the baseline for corresponding applications and software libraries, then an industrial implementation of these components especially the NAT and name servers will result in comparable performance to baseline performance of industrial components. A simple redesign on the way iptable [4] stores NAT entries can lead to a significantly reduce memory requirements.

Bhushan *et al* [18], provide an implementation of the SI system, and perform ns2 simulations to assess efficacy of the system. They also compare SI system with other similar defensive systems. Our thesis provides 1) A novel SI framework implementation based on well known techniques 2) Experiments that verify the performance of our SI

implementation, both end to end parameters such as Connection establishment time and SIGateway load parameters such as memory consumption and packet forwarding time. 3) Comments on how this system can be effectively deployed to control packet floods.

This thesis contributes an effective implementation of Spread Identity based on well known RFCs (making it hard to refute its feasibility). We focus primarily on the performance of the NAT subsystem and the name resolution process which form the core subsystems of our implementation of Spread Identity Framework. We model our experiments on some ideas described in RFC 2285, which deals with defining terms, units and metrics used for firewall/Gateway performance. We perform experiments to measure the connection establishment time, packet translation time, name query resolution time and memory requirements. We also find out memory requirements of the SI system. We compare and contrast these results with a single NAT implementation.

We use IP-CONTTRACK, Net filter [4] (iptables [4]) as the NATting subsystem in our experiments. We use Bind9 as our name server. We implemented the core SI algorithms using java. We programmed the experiments using C, and executed them on Deter lab's network tested. We used Ubuntu 10.0.4 OS on every machine. Up to 100,000 simultaneous connections were generated to test the system.

The rest of this thesis is organized as follows. Chapter 2 describes background information of SI framework and its comparison with technologies that are based on similar ideas. Chapter 3 describes a detailed overview of our implementation and technologies we used. Chapter 4 describes the experiment topology, procedure and results.

Chapter 2

RELATED WORK

2.1 Classification and Taxonomy of Denial of Service Attacks

DDoS attacks are an open problem in present day internet system. There have been many attempts to classify and categorize DDoS attacks, and defense methods. Abbass Asosheh *et al* [4] provide an introductory view on classification of various DDoS attacks and defense mechanisms. Valeria *et al* [5] provide a more detailed explanation of various attack and defense mechanisms. DDoS Countermeasure classifications are based on factors such as methods employed (filtering, overlay etc), nature (reactive, proactive etc), and systems affected (end user, router, servers etc). There does not appear to be a single standard classification of DDoS attacks.

2.2 Related Architectures

There are some architectures that achieve a similar but static (as opposed to dynamic) disassociation of IP addresses and hosts (service), by using a completely opposite approach. In general these architectures employ host replication to create N number of hosts providing similar service.These N hosts are associated with N unique IP addresses. Thus, a client can choose from N IP addresses to access same service (but a different host). A technique that uses this type of architecture is Load balancing [5]. However due to static nature of this architecture, figuring out mapping between IP addresses and host names is easy. Even with N IP addresses for a particular host name, one can still target each of them individually or all at once.

2.2.1 Early work on Spread Identity.

Phatak *et al* [2] proposed the original SI framework. DhawalBhakta *et al* [19] demonstrated the SI framework by implementing it on NS2 Simulator. They perform experiments for finding number of packets reaching their intended destination, connection establishment delays. Bhushan *et al* [18] performed a simulation of SI framework on NS2 Simulator. They also compared SI with other experimental strategies

to defend against DDoS attacks. They also study the anonymity related properties of the SI framework.

2.2.2 Internet Indirection Infrastructure

Another comparable scheme is Internet Indirection infrastructure (i3) [6] [7]. I3 is claimed to be a clean mechanism that provides ability to hide IP of hosts, through various public triggers. These triggers can be selectively dropped resulting in reduction of unwanted (as decided by host) traffic. In this system it is assumed that the IP address of a host is very hard to find out by using other means, in reality this may not always be the case. This system also requires implementation of indirection mechanism viz. I3, and significant change in infrastructure.

2.3 Comparison of i3 and SI

The I3 based system is somewhat similar to our implementation of the Spread Identity Framework. I3 provides public triggers which can be accessed by the end users. In our case, we provide a modified name resolution process which provides for creation of "tokens" (address remapping NAT entries) implicitly, per host-client pair basis. However, SI does not require major changes to internet infrastructure such as i3, but only requires the ability to insert/remove filters in the last-hop (ISP) routers, or a proxy placed before the last hop router.

Both systems are based on the assumption that end hosts are better equipped to make decision regarding which flows should be dropped in an event of overload, which is typical of a DDoS attack. Both systems work on classifying flows; in i3 based system it is classified based on which public trigger the flow uses, while in case of SI it is classified based on address pair (client, host). Both systems use Challenge Response tests (CAPTCHA etc), to handle new connections in an event of a server load/flood.

2.4 Measurement and Performance Evaluation

Benchmarking DDoS countermeasures / defense frameworks is another challenging area. Because of the huge number of DDoS attacks and scenarios, it is extremely difficult to perform rigorous analysis on each and every type of DDoS attack possible in a timely

manner. J Mircovik *et al* [8] provides explanation of Deter Framework and how it can be leveraged to rigorously test DDoS frameworks.

RFC 3511, RFC 2647 provide descriptions of various terms, and metrics that are meaningful when benchmarking firewall. Our measures are somewhat based on the terms as defined in these RFCs but, more suited towards exposing merits and demerits of our implementation.

We mention earlier attempts made to implement the SI System. These experiments were related to measuring efficacy of the system in actually tackling a DDoS-like scenario. Dhawalbhakta *et al* [19] provided some early demonstration of the SI framework. Bhushan *et al* [18] provided simulations that were related to demonstrating the DDoS defense capability of this framework.

Chapter 3

IMPLEMENTATION

In this chapter we will discuss the original SI proposal, some current techniques that can be leveraged to implement SI (RFCs 2663, 2694). We will also discuss how we implemented these techniques. We start by briefly describing two techniques published in RFC 2663, 2694. After this we describe general functioning of the SI framework, and some ways in which it can be implemented on an organizational level. We then describe in brief our implementation of the framework followed by a detailed example of how the system functions. Finally we describe the address remapping algorithm which forms the core of the SI Gateway (SIG) functionality, and end by outlining the limitations of our implementation.

We implemented a new system, because Bhushan *et al* [18] did not take into considerations already available techniques (RFC 2663, 2694) while performing their implementation. There implementation was based on ns2 simulator. By basing our implementation in well known techniques, we believe that our implementation is inherently feasible. For example consider the functioning of their SIR(I/O). They have modified Bind9 to return IP addresses based on query source and system state. For this to work, they would need to know for which external IP address pair, a free internal IP address pair is available. Now since the NAT system is on SIG, they would have required some form of communication to achieve this. This would have made their system, extremely underperforming.

3.1 Twice NAT (RFC 2663)

Srisuresh [16] (year: 1999) describes various IP network address translation related techniques and terms. Our interest is the technique of twice NAT. Usually, when using network address translation, only the source (outgoing packet) or destination (incoming packet) is changed before rerouting the packet into another network. However, in twice NAT both the source and destination address of the IP packet are changed. This is done to isolate networks, so that migrating from one network to another is easier. In this case, even though the old IP addresses are being used by some other network, the source

translation mechanism of twice NAT translates the old IPs into new ones. The concept of twice NAT can be easily tweaked and used in SI, to achieve 'double blindfolding.' Firewalls /Gateways that allow twice NAT are available from leading network equipment vendors Cisco and Juniper [13]. Converting these hardware/software Gateways into functional SI gateways is then only matter of a patching some of its functionality.

3.2 DNS Application Level Gateway (RFC 2694)

Whenever a public host server (hosted service) is located inside a private network, and uses IP addresses in layers above the network layer, a program or hardware is needed. This program translates the addresses when a packet transitions between public and private realms using NAT. This program/hardware is broadly known as Application Layer Gateway (ALG). Srisuresh [17] (year: 1999) describes how to implement a DNS Application Layer Gateway and how to handle various issues that may arise. ALG can be leveraged in our architecture to reduce complexity of implementing address remapping.

3.3 Spread Identity Architecture

The basic Spread Identity architecture [2] contains two important components: A Spread Identity Resolver (SIR) and a Spread Identity Gateway (SIG). The SIR handles the name-resolution process. The SIR can be split into inbound (SIRI) and outbound resolvers (SIRO) if necessary. SIRI handles inbound queries, and SIRO handles outbound queries. After receiving a name query from a client for a particular host, SIRI/O sends appropriate NAT message(s) to SIG. SIG responds by creating timed NAT entries. SIG then responds to the client with a reply containing the dynamic identity of the host. Depending on further communication between host and client, the entries may or may not become permanent (for that session). The process of dynamic identity selection for creating timed NAT entries can be random or based on some function of system state and source IP. The result is a many-to-many mapping of identities.

3.4 Implementation

In our implementation of SI, we move some functionality of SIR (I/O) to SIG to eliminate messages sent by SIR (I/O) to SIG. Instead, the name resolution process of SIR (I/O) provides enough information to SIG for generating NAT entries. By making this

change we are able to use existing technologies to implement our system.

3.4.1 Architecture

Figure 1 below, shows one of the ways in which the Spread identity framework can be deployed in an organization. The SIR (I/O) is in this case just a plain implementation of an authoritative name server *e.g.* an instance of Bind 9. The SIG contains implementation of DNS Application Layer Gateway, NAT system and a firewall. Each name resolution requests from public clients passes through the SIG. Each time a name resolution request is received the SIG sends SIR (I/O) the same name resolution request. SIR (I/O) responds by generating a name resolution reply to SIG‘s request. Upon receipt of this reply, SIG installs appropriate NAT entries allowing translation from a unique public host-client IP pair to a unique internal host-client IP pair. It then replies to the original public client with a new modified reply that contains a public address chosen dynamically during installation of NAT entries. The edge router can be used to implement default drop rule to drop all unknown traffic. Filters can also be installed based on source or destination addresses on the edge router. Cisco white paper [11] describes a similar but manual method based on this filtering technique to stop DDoS attacks.

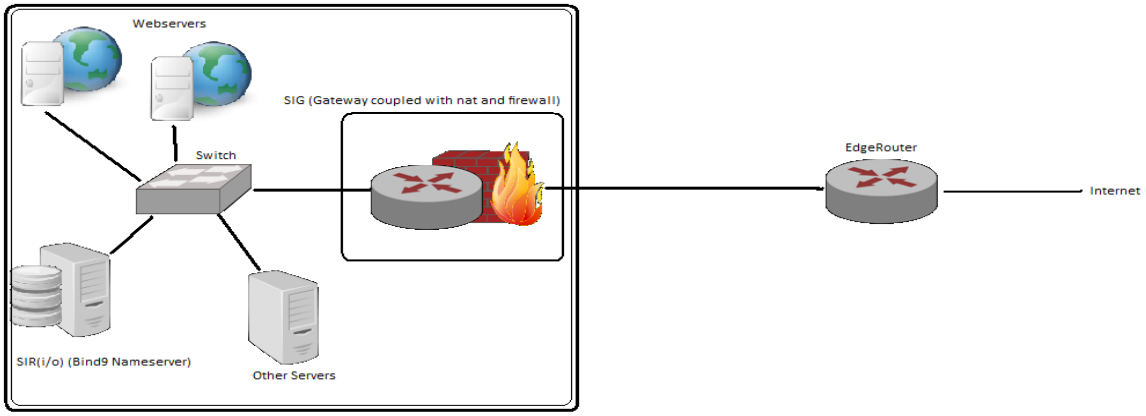

Figure 1 An abstract view of components of the Spread Identity Framework when implemented at an organizational level. On the left, a minimal view of an organization, two web servers, a 3rd proprietary server and a name server. The perimeter gateway is equipped with a firewall along with a Spread Identity Gateway.

3.4.2 Alternative architectures.

There can be many other ways to deploy this system. For example, the Authoritative DNS Server or SIR (I/O) can be split into two, such that the inbound server is located in a DMZ outside the internal network and the outbound server is located inside the network. Another possible alternative is distributing the SIGateway across more than one physical gateway machine and connecting these machines to each other using a separate high speed network, similar to distributed firewall implementations. One advantage of this kind of deployment is resilience and load balancing. Each of these architectures can be based on any of the available NAT and DNS subsystems.

3.4.3 Implementation(S/W)

To implement our system, we require two major sub-systems: a name server system and a NAT/packet filtering system. There are various priced and FOSS alternatives available for each of the two systems. We select Bind 9 implementation [9] as our name server. This was due to popularity of this name server; it was one of the most deployed name servers as of Oct 2010 [14]. For packet filtering and NAT subsystem, we select Net Filter and its IPtables interface [10]. Net Filter is the "de facto" Linux kernel level solution for packet filtering, NATing and routing network packets. It is available on every Linux kernel (2.6+). Many firewalls are based on Net filter. We choose java as our language for implementation of SIG on the gateway. This choice was based on APIs available for java, and its clean object oriented structure. We use Xbill's DNS API to manipulate DNS traffic.

3.4.4 Address Resolution Scenario

We now briefly go over the address resolution process of a public client accessing hosts of organization implementing SI. We do not explain the address resolution process of an internal host accessing a public host, since it is the mirror opposite of the original case, and already documented in [2][3]. However, we briefly go over it at end of this section, along with other invalid scenarios.

Figure 2 shows a logical schematic of our system, with a public client trying to access an internal host. Assume sicompany.org is the domain name registered for this organization. For the sake of simplicity we use notation 'IP<number>' or 'IP<alphabet>'

as the IP address identifiers. SIG has pooled 4 IP addresses (IP1, IP2, IP3, IP4). There can be any combination of interface and IP addresses that can lead to this combination (*e.g.* 2 interfaces/links, with 2 IPs each) etc. SIR (I/O) is a simple Bind9 name server configured to act as the authoritative name server for this domain, with recursion disabled. As seen in Figure 2. IP515 is the IP address of a public client trying to access a host with http service using URL http://www.sicompany.org.

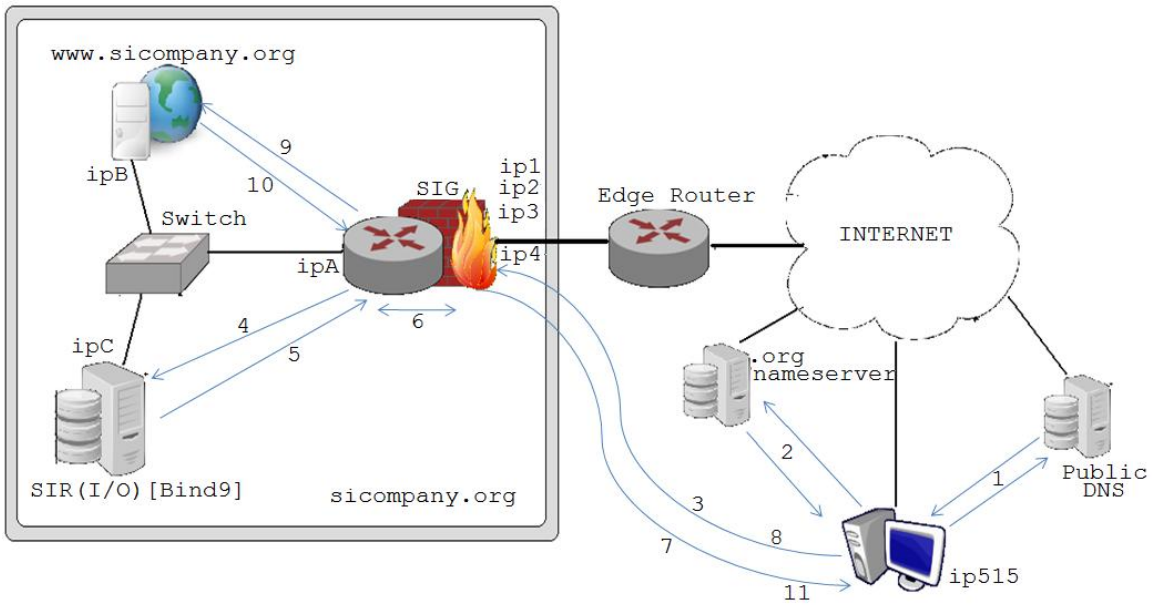

Figure 2 Spread Identity Address resolution example. A client with ip ip515 tries to access a resource behind SIG by using url http://www.sicomapny.org. The blue lines represent communication path, the numbers on blue line indicate chronological step number that uses the path.

In general, we refer to addresses IPB, IPC, and IPA as "internal IP addresses," these are private and not publicly routable. Similarly, addresses IP1, IP2, IP3, IP4, IP515, and IP322 are public addresses, which are publicly routable. We call these IPs as "external" or "public". In addition to this, IP322 is also called a blindfolding address, as it is used to mask the public IP addresses of any host or client outside the organization. The uni-directional lines are communication paths, each of which has been labeled by one or more numbers which represent a major step in communication. We now describe each chronological step as shown in Figure 2.

1. Client IP515 queries the public name server provided by its ISP for www.sicomapny.org. The public name server replies with a reference to ‘org.’ name server, the authority of all domains under org. domain.

2. The org name server contains an entry for domain name sicompany.org, and name server ns1.sicompany.org. These entries also contain the published IP address for sicompany.org say IP1, and for ns1.sicompany.org say IP2. The org. name server replies with reference to ns1.sicompany.org and its IP address IP2.

3. The client sends a name resolution query to ns1.sicompany.org located at IP2. This network endpoint (ip2, port 53) is being monitored by our SIG service which implements DNS_ALG and SIG’s functionality.

4. This SIG Service receives the name query, and in response the SIG queries the SIR (I/O) to resolve the name specified in the received name query. This is done by generating a new name query based on contents of the received name query.

5. The authoritative name serve or SIR (I/O) which implements BIND9 sends a reply to SIG with answer record(s) corresponding to its query or with a reply that contains host not found status.

6. If the reply from SIRI/O contains an answer record, the SIG selects an externally pooled IP based on the state of the system and the IP in answer record .Assume this IP to be IPB. SIG then picks a random external routable IP as a blindfold such that it was not previously selected for public client‘s IP (ip515) and requested host‘s IP, i.e. IPB. Assume this IP to be IP322. SIG then inserts a dynamically generated NAT entry as shown below in its NAT Table based on above four IP addresses:

(IPB, IP322) ←→(IP4, IP515).

Thus, any packet traveling from IPB to IP322 is translated to a packet traveling from IP4

to IP515. Similarly, any packet traveling from IP515 to IP4 is translated to a packet traveling from IP322 to IPB. After a set amount of time this entry is deleted from the NAT rules.

7. An answer record is generated by the SIG daemon corresponding to public client (IP515)'s name query, with the answer record now containing the previously chosen externally pooled IP (IP4). This reply is then sent to the original query source, i.e. to public client (IP515).

8. After receipt of query answer, public client (IP515) now has an answer to its name query for www.sicompany.org it connects to port 80 of IP4 (IP4: 80). When the SYN packet from this public client is received by SIG, it is handled by the NAT subsystem, which sees the host-client IP pair (IP4, IP515). It then translates this pair to (IPB, IP322) since a matching rule exists. The ACK from IPB to IP322 is similarly translated to a handshake from IP4 to IP515 etc. After completion of the 3way handshake, the TCP connection is established and now communications can begin normally.

At this point, any external 3rd party snooping on the communication channel sees IP515 communicating with IP4, and any internal snooping party will see IPB communicating with IP322.

At Step 6, SIG has four IP addresses. Based on whether two of those are internal or external, the SIG has to handle following cases.

Let,
X= Client's IP or Original Query Source. (e.g. IP515),
Y= Host IP obtained from answer of name query request by SIG. (e.g. IPB),
B= randomly chosen publicly routable blindfold IP (e.g. IP322), and
P= externally pooled IP addresses chosen by SIG. (eg.IP4).

Case 1: Client X is Internal, Host Y is Internal.

This case occurs when an internal client X is trying to access an internal host Y.

In this case, SIG simply replies with SIR (I/O)'s reply without doing any further work.

Case 2: Client X is Internal, Host Y is Public.

In this case, notice that SIG has performed 2 name queries (one for internal hosts which were unsuccessful, one for public hosts). In this case, an internal client X is trying to access a public host Y. The following NAT rule is installed:

(Src X, Dest B) ←→(Src P, Dest Y).

In this case, the name query generated by SIG will be sent with the name query's source field set to IP. This needs to be done so that the appropriate address remapping entries are generated at host SIG if they implement SI.

Case 3: Client X is Public, Host Y is Internal

In this case an external host X wants to access an internal host Y. SIG performs only one query (with SIR (I/O)), if it is successful, following NAT entry is installed:

(SRC X, DEST P) ←→(SRC B, DEST Y).

Case 4: Client X is Public, Host Y is null

This case occurs when the reply from the SIR (case3) is without any answer record. This means that the requested host does not exist in SIG's organization. SIG simply forwards the reply to the client at X without doing any work.

We need to differentiate this case from one with internal X because in that case we provide a recursive service to find out the IP of Y, after internal SIR (I/O) lookup fails.

3.4.5 Address Remapping Algorithm

Now we describe the address remapping algorithm that SIG implements, which forms the

core of our SI implementation. The following are some supporting functions used by the SIG address remapping algorithm:

doNameResolution (server, query): performs a DNS name query.
doNameResolution (server, query, IP): performs a DNS name query with IP as source IP.
IsInternalIP (IP address): checks if the IP is internal.
SendAnswer (IP address, dnsAnswer): sends dnsAnswer to IP.
ContainsIP (answer): checks if answer contains a valid IP Address (A) record.
ExtractIPFrom (answer): extracts IP from the Answer records
ReplaceIPinAnswer (...): replaces IP in answer with new IP.
GetMappedIP (srcIP, ansIP): return IP mapped to this particular set of client and host.
MappingExsists (srcIP, ansIP): checks for IP mapping for this pair of client and host.
GetRandomRoutableIPFor (xIP): Get a random IP for a particular xIP.
GetPooledIPFor (xIP, systemState): Gets pooled IP based on either system state or xIP or both.
SetMappingFor (sIP, dIP, rIP): Sets Mapping [sIP] [dIP] =rIP.

```
Algorithm: Address Remaper (srcIP, nameQuery)

1. answer = doNameResolution(SIRIO, nameQuery);
2. if(containsIP(answer) = NULL AND isInternalIP (srcIP) = TRUE)
{
      answer =
      doNameResolution(PublicDNS,nameQuery,RandomPublicIP(srcIP));

      if(containsIP(answer) = NULL)
      {
            sendAnswer(srcIP,answer);
            return;
      }

      ansIP = extractIPFrom(answer);
      if(mappingExsists(srcIP,ansIP) = TRUE)
      {
```

```
            replaceIPinAnswer(answer,ansIP,getMappedIP(srcIP,ansIP));
            sendAnswer(srcIP,answer);
            return;
      }
      else
      {
      RandomRoutableIP = getRandomRoutableIPFor(ansIP);
      PooledRoutableIP= getPooledIPFor (srcIP, stateOfSystem());
      addNATEntry(srcIP, RandomRoutableIP, PooledRoutableIP,
                  ansIP, timeOutValue);

      replaceIPinAnswer(answer, ansIP,RandomRoutableIP );
      setMappingFor(srcIP, ansIP, RandomRoutableIP,timeOutValue );
      sendAnswer(srcIP,answer);

      return;
      }
}
3. else if(containsIP(answer)!=NULL AND isInternalIP(srcIP) = TRUE)
{
      sendAnswer(srcIP,answer);
      return;
}
4. else if(containsIP(answer)!=NULL AND isInternalIP(srcIP)=FALSE)
{
      ansIP=extractIPFrom(answer);
      if(mappingExsists(srcIP,ansIP)=TRUE)
      {
            replaceIPinAnswer(answer,ansIP,getMappedIP(srcIP,ansIP));
            sendAnswer(srcIP,answer);
            return;
      }
      else
      {
            RandomRoutableIP=getRandomRoutableIPFor(srcIP);
            PooledRoutableIP= getPooledIPFor(ansIP,stateOfSystem());
            addNATEntry(srcIP,PooledRoutableIP,RandomRoutableIP,ansIP,
                        timeOutValue);
```

```
            replaceIPinAnswer(answer,ansIP,PooledRoutableIP);
            setMappingFor(srcIP,ansIP,PooledRoutableIP,timeOutValue);
            sendAnswer(srcIP,answer);
            return;
      }
}
5. else
{
      sendAnswer(srcIP,answer);
      return;
}
```

This algorithm starts by forwarding the name query received by SIG to SIRI/O. The SIR(I/O) may or may not reply with a valid IP resolution based on the contents of the domain name in the query.

At 2, we consider the case where SIR (I/O)'s response does not contain a valid IP and the query source IP is internal. In this case, the algorithm makes another recursive or iterative query to a public DNS Name server. If the end response (after recursion /iteration) from the public DNS Name server does not contain a valid IP, we simply forward the public DNS Name server's response to the initial query's source IP.

However, if the DNS Name server's response contains a valid IP, we check if this address has been re-mapped already for this particular client's source IP, and if it is, we modify the answer to include the remapped IP.

If this valid IP was not re-mapped already, we select two IP addresses, one random routable public IP for internal blindfolding, and one publicly pooled IP for the internal client. A timed entry is installed in the NAT subsystem for a time 'timeOutValue' based on the two selected IPs. We then modify the response of the public DNS to include random routable IP, and send it to the client's source IP.

At 3, we consider the case where SIR (I/O)'s response contains a valid IP, and the query source is internal. In this case we simply forward SIR (I/O)'s response to the query's source IP.

At 4, we consider the case where SIR (I/O)'s response contains a valid IP, and the

query source is external. In this case, we first check if the IP in response to SIR (I/O), is already mapped for the given query source IP. If it is, we modify the response to include the remapped IP and send it to the source IP. If not, then we select a random routable IP and a public routable IP. As before we now create a NAT entry for time timeout. We then modify the response of SIR (I/O) to include public routable IP and send this response to the query source IP.

At 5, we consider case where SIR (I/O)'s response does not contain a valid IP and query source IP is external. Since we are not responsible for name resolutions of hosts that are not part of our organization, we simply forward SIRO's response to the source IP.

The above algorithm is implemented in SIG as a java application running on a server-jvm. Figure3 below shows the interaction of the SIG java application with various subsystems in SIG.

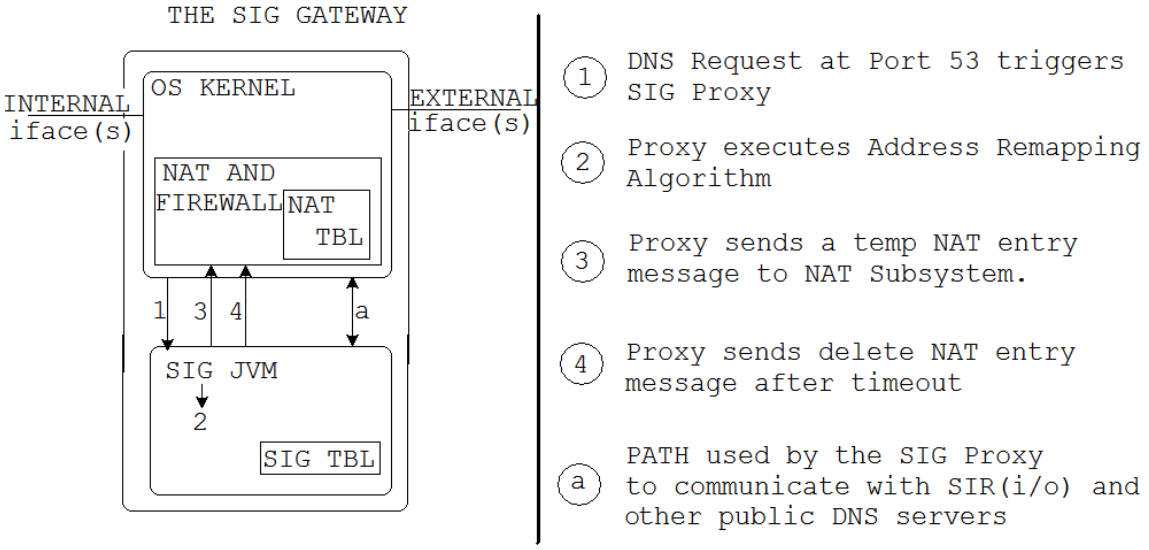

Figure 3 Left general component/interaction view of the Spread identity Gateway. To the right brief explanation of various inter-component interactions.

As shown in Figure 3, SIG JVM runs in user space, while other subsystems (i.e. the NAT and Firewall) run in kernel memory space. As a result we need to maintain our own NAT table (SIG table), which is a copy of the NAT Table in user space.

3.6 Limitations of this Implementation

There are some limitations and inefficiencies in this implementation. Most of these

limitations are a result of using a preexisting NAT subsystem (iptables/net filter). Iptables was designed to add or remove static entries in the NAT table in a single invocation. The NAT entries are assumed to be inherently static and, as such, mechanisms such as timeout, single entry deletion, single entry addition are not supported via an API or by other efficient means. Iptables can provide the later two functionalities, though inefficiently. Moreover, Iptables is not reentrant, and this means we cannot add NAT entries in parallel. Iptables thus acts as a bottleneck in the system and reduces system performance considerably.

Another issue is generation and tracking of the pooled IP addresses and random routable addresses from a set of available addresses. As mentioned earlier, there is a need to track IP addresses consumed by hosts and clients, to avoid a single IP-pair to resolve into more than one IP-pairs when performing the address remapping step (NAT). Having access to the NATting table can make this process easy, but since ip_conntrack is implemented in the kernel, we have to maintain our own table. Doing so doubles the memory required by our implementation.

One benefit of using net filter is that it uses configurable hash functions for NAT table lookups/comparison and provides a choice of tuning these functions as per the required load. We can adjust the maximum number of connections that can be tracked, and we can also choose the number of buckets into which the hash function should distribute its values. If we have knowledge of the maximum number of connections, then selecting the number of buckets determines the max number of NAT-entry-search operations per connection.

Modification of the iptables and the net filter source is needed to support the dynamic behavior needed by our SI implementation more efficiently. This will require a higher degree of co-operation from the net filter core team, and at significant amount of time due to the size of Netfilter codebase and lack of supporting documents.

Chapter 4

EXPERIMENTS AND RESULTS

In this chapter we describe experiments carried out to test the performance of the DNS and the NAT subsystems. We start by describing the topology we used to perform the experiments. We argue that since our defense is not distributed, a specific topology or variation in topology is not required, thus significantly simplifying the testing process. Our tests concentrate on the TCP protocol since this protocol dominates all communications across the internet today. We then describe the experiment setup and at the end of each experiment we describe the results.

4.1 Connection Topology

We use deter lab to perform our experiments. Deter lab provides a network test bed that was designed to support DDoS as well as other malicious code execution and testing. Figure 4 below shows the topology we used to perform our experiments.

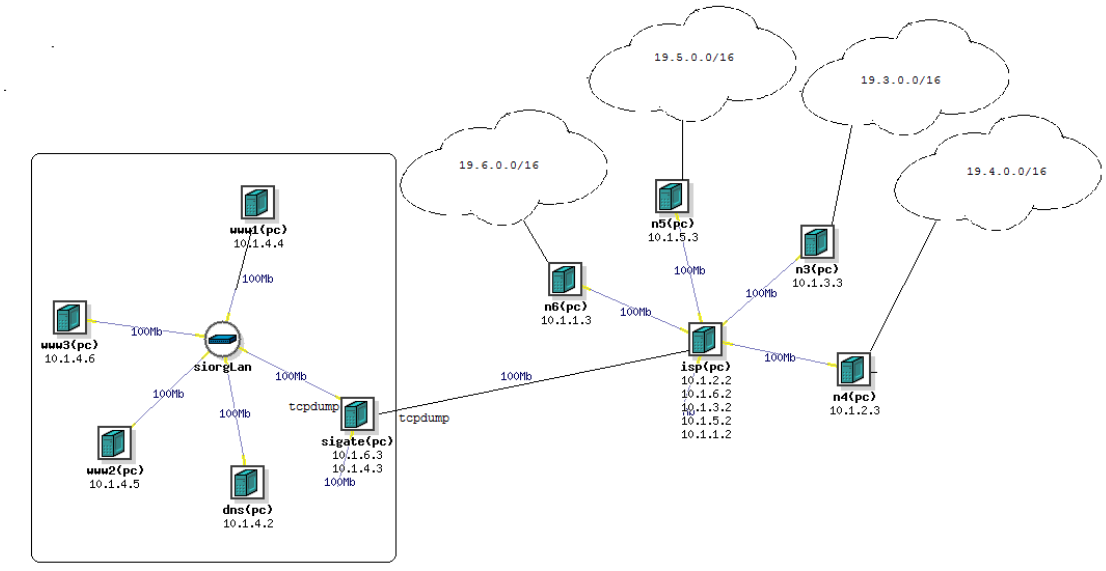

Figure 4 Experimental Setup on Deterlab used to perform experiments. The organization employing Spread Identity is marked by a square.

As seen in Figure 4 we have 3 hosts one DNS server (SIR I/O), and a gateway (SIG). To

the right is an arrangement that can generate 256k connections each with a different source IP from the 19.0.0.0/8 range. N6, N5, N4, N3 act as gateways to each of these virtual networks. These virtual networks are implemented by aliasing the IP addresses of the range of the virtual network to any single machine, to reduce the number of machines required.

We argue that variety in the topology is not important for our experiments, since we are testing for parameters that are similar to firewall testing. For example, in case of NAT systems, the only input parameter that we need to consider is the size of NAT table; topology has no bearing on the size of NAT table.

We are also interested in testing end-to-end parameters such as connection establishment time, and name resolution time, and while connection establishment time or name resolution time will vary according to the topology, we know that the time required to communicate from (to) the last hop router will remain similar and unaffected by topology.

4.2 Experiment Setup

Initially we had designed separate experiments for each measurement. We however regrouped it into two experiments. One important reason to do this was the nature of hardware assigned for the experiment differed each time. We designed experiments such that all the parameters can be measured in one trial. We ended up generating two experiments. One experiment measured all the parameters when SI was deployed. Another measured all the parameters Single NAT base system was deployed.

4.3 Experiment Procedure:

Each machine that represents the subnet generates a variable number of processes (25 in our case). Each process has its own sets of IP addresses which it binds to and then tries to repeat a sequence of steps.

For each IP to which client binds, the client process uses following algorithm:

```
Algorithm: IPClient (cIP,dev,URL,stdout)
[sIP is server IP, cIP is ClientIP, dev is interface/link name, URL is
URL ,stdout is standard output stream]
```

```
[Dig(ip,dev,URL)          : does a  Namequery on URL using ip,dev as
source address]
[Connect(cip,dev,sip)  : Active connect from cIP to sIP]
[write(stream,record)  : write record to stream]
[getIP(receieveData(h)): Recieve data from given handle, and then
extract IP]


1.  {sIP,queryTime} = Dig(cIP,dev, URL);
2.   time = startTime();
      handle=Connect ( cIP,dev,sIP);
     time = endTime() – time;
3.  if (handle == INVALID)
      {
            write(stdout,{-1,time,queryTime,FAIL});
            return;
      }
4.  IP=getIPFrom(receieveData(handle));
      write(stdout,{handle,time,queryTime,SUCCESS});
      return;
```

In IPClient Algorithm we perform a domain name query for a given URL, followed by a timed connect call, followed by reception of data packet which contains a server handle number or connection number. The handle number, time required to perform name query, and time required to execute connect() are then stored in a file represented by stdout. In case of failure, -1 instead of handle number is stored with a FAILED status.

The SIG application runs on SIGateway with a pre-configured pooled-IP range, and blindfolding-IP range. We also run two instances of packet capture utility (tcpdump) on the incoming interface and on the outgoing interface of the SIG. There data are then used to find packet forwarding time and input load w.r.t time. We reconfigure the net filter system to handle 150,000 connections, and generate ~50k (47533) buckets.

The server simply listens for a connection, and as soon as a connection is established, it sends a connection number. We did not use http servers since they close inactive/idle TCP connections. Our server deliberately keeps the connection open so that

the Connection Tracking entries remain alive until the end of experiment.

We do not try to parallelize the process because we are trying to measure the connection establishment time against number of connections already established. We start a single 64k subnet controller at first, after establishing 30-40k of its connections we start the next subnet of 64k clients and so on. We used only 3 out of 4 possible subnets. We repeated the experiment 3 times with SI and only once with static NAT We did not find any significant difference in each of the 3 runs, but we did not average the values, due to difference in connection numbers. We also removed some records that failed to connect, since these values do not represent the connection establishment time.

The Connection Establishment time and the Name Query Resolution time are directly available from the output of the client processes. However for packet forwarding time of SIG and input load characters we used tcpdump output. We run a script that matches the dumps from external interface to the dumps from internal interface using SYN/ACK match rules.

4.4. Results

Following is the summary of experiments and results obtained by executing them on the above mentioned setup. We discuss 5 related parameters. At the end of this section we try to contrast performance differences between Spread Identity and Single NAT implementations. We have not measured difference between NAT and non-NAT scenario, since those will vary based on the NAT systems used. Instead we measure baseline (Single NAT) performance and compare it with Spread identity performance.

4.4.1 Input Load on SIG/Gateway.

Figure 5A shows the input load characteristics as seen on the input interface in case of SIGateway and Figure 5B in case of SNAT Gateway. In both cases we observe three spikes. Each spike indicates a time at which a new subnet started establishing connections. Because the first request that each client process sends is independent of any SIG reply, a relatively high load is observed on the input interface of the SIGateway represented by spike.

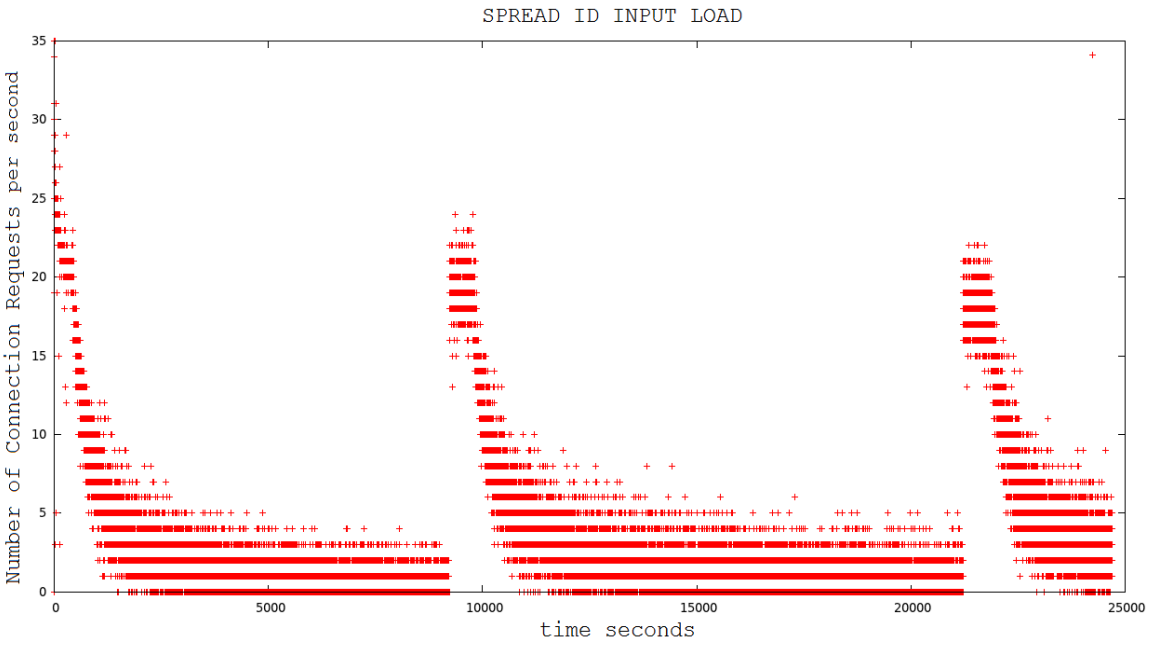

Figure 5A Input load as seen on Spread Identity Gateway. The spikes in the graph represent a high connection/sec value, which indicates start of a new subnet, with a maximum of 30 requests handled per second.

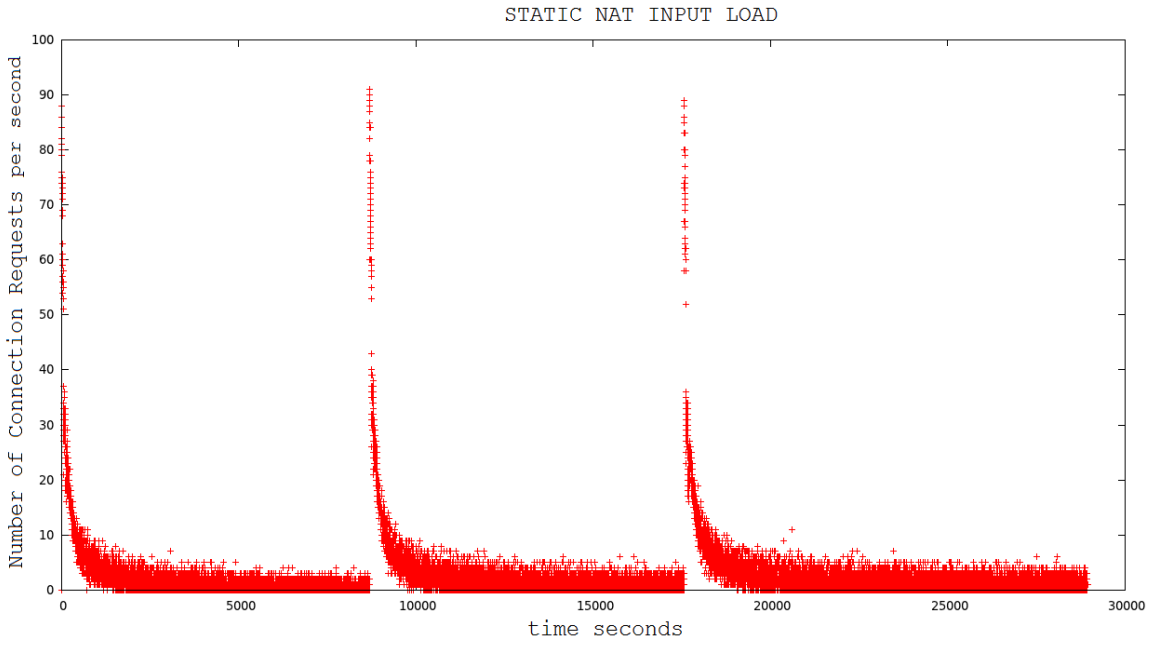

Figure 5B: Input load as seen on SNAT Gateway. The spikes in the graph represent a high connection/sec value, which indicates start of a new subnet, with a maximum of 90 requests handled per second. Observe that the maximum connection requests handled per second is more than what is observed in Figure 5A.

Notice the maximum rate of number of connections that are established: in case of single-NAT the maximum rate is ~90 connections, while for of Spread Identity it reduces to 30 to 35 connections.

4.4.2. Connection establishment time.

Connection establishment time is the time required for a TCP connection to reach "ESTABLISHED" state. For more description please refer to RFC2647. The goal of this experiment is to observe if the connection establishment time is affected (increases) by the increase in the size of the NAT table. The size of NAT table is in turn dependent on number of established connections.

In our experiments we measure Connection Establishment time by timing the POSIX defined connect() call. The connect() executes the usual 3 way handshake protocol. We track this value as a function of the number of connections (handle number) returned by the server process. Since our server process does not close TCP connections, this handle number is equivalent to number of NAT Connection Tracking Entries.

To provide a baseline view, we replace SI with static NAT entries that expose DNS and web server to the external network, and repeat the same experiment.

Figure 6A gives the connection establishment time as a function of connection number for Spread Identity System. Figure 6B shows baseline performance of the NAT subsystem (net filter), using a simple static NAT entry to expose DNS and WWW servers.

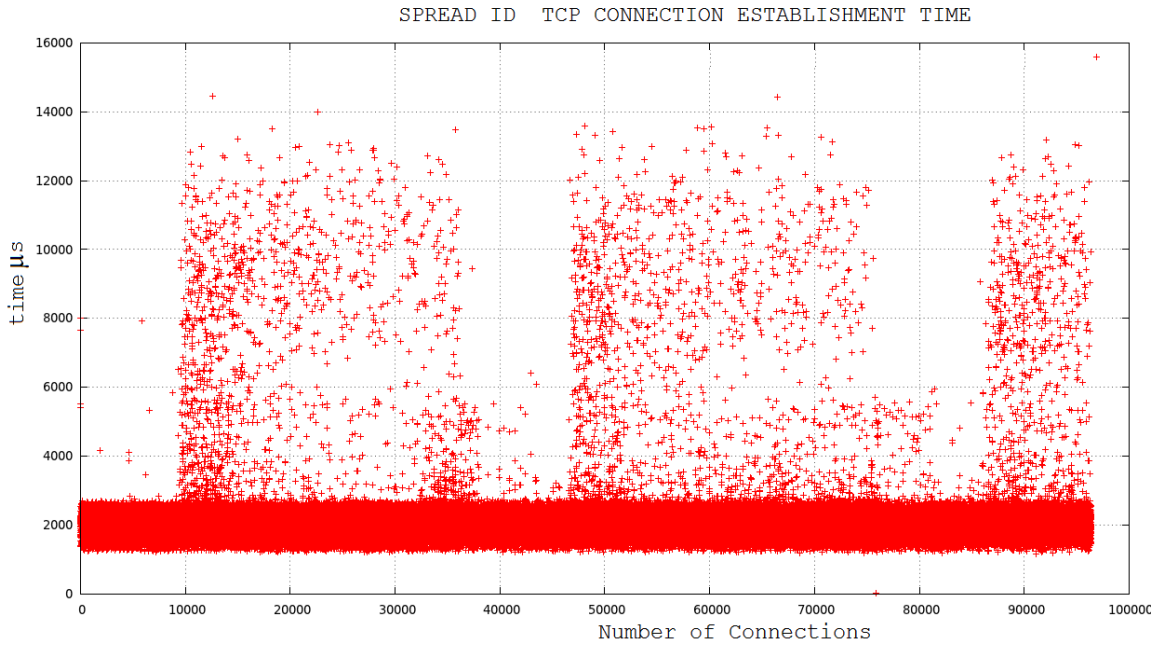

Figure 6A: Spread identity Connection establishment time measurement. On average the connection establishment time is between 1000 to 3000 microseconds.

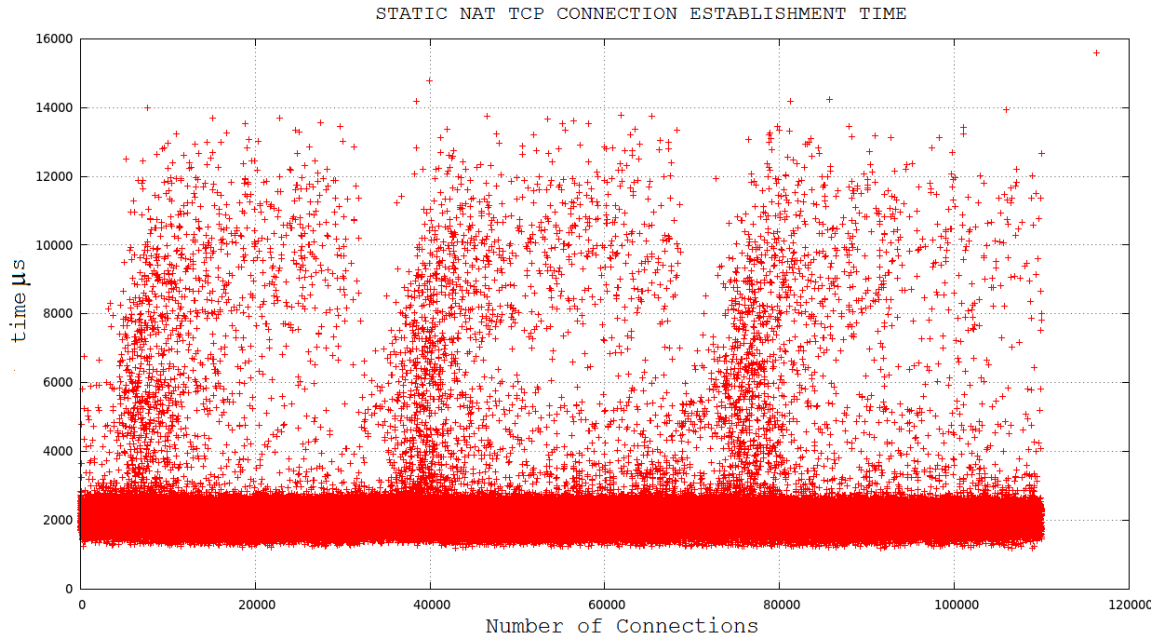

Figure 6B: Baseline Connection establishment time. In this figure as in Figure 6A, on average the connection establishment time is between 1000 to 3000 microseconds.

As seen in Figures 6A, 6B above, the pattern of connection establishment time is almost similar in both the cases with no significant difference. The connection establishment time ranges from 3000 to 1000 microseconds in majority of the cases. We

do not see any significant degradation of the connection establishment time when compared to the base line in Figure 6B.

We can see that there are some distinct areas where the time required to establishing connection increases considerably. These areas represent the start of a new 64k subnet processes. These areas coincide with input load spikes, as shown in Figure 5A and 5B. We infer that connection establishment time is dominated by the number of simultaneous connection requests per second rather than the SI or Single NAT implementation.

This experiment proves that the time consumed by the NAT Subsystem in case of SI or Single NAT, is insignificant when compared to other delays such as bandwidth and server load.

4.4.3 DNS Response Time

The Name Resolution Time is the time required to perform a DNS name resolution. In case of the base line (Single NAT) performance, this time is simply the time required for the DNS system to respond to a query. In case of Spread Identity this time encompasses the time required for executing the Address Remapping Algorithm and associated book-keeping activities, along with DNS name resolution.

Figures 7A and 7B show name-query resolution time as a function of connection Number or handle number. Figure 7B shows the resolution time measured when Spread Identity is deployed. A baseline performance (single NAT) is also seen for comparison in Figure 7A.

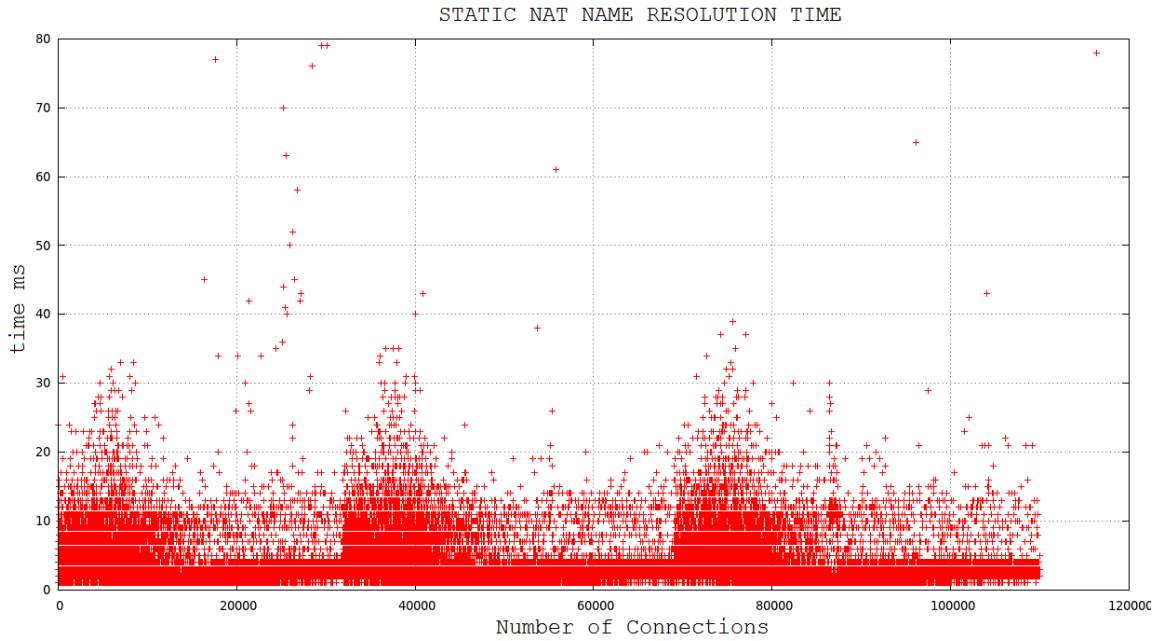

Figure 7A: Static NAT Name query Resolution Time as observed on the client side. Seen are three distinct spikes which when compared with Figure 5B, we see that the query response time is affected adversely by increase in query rate.

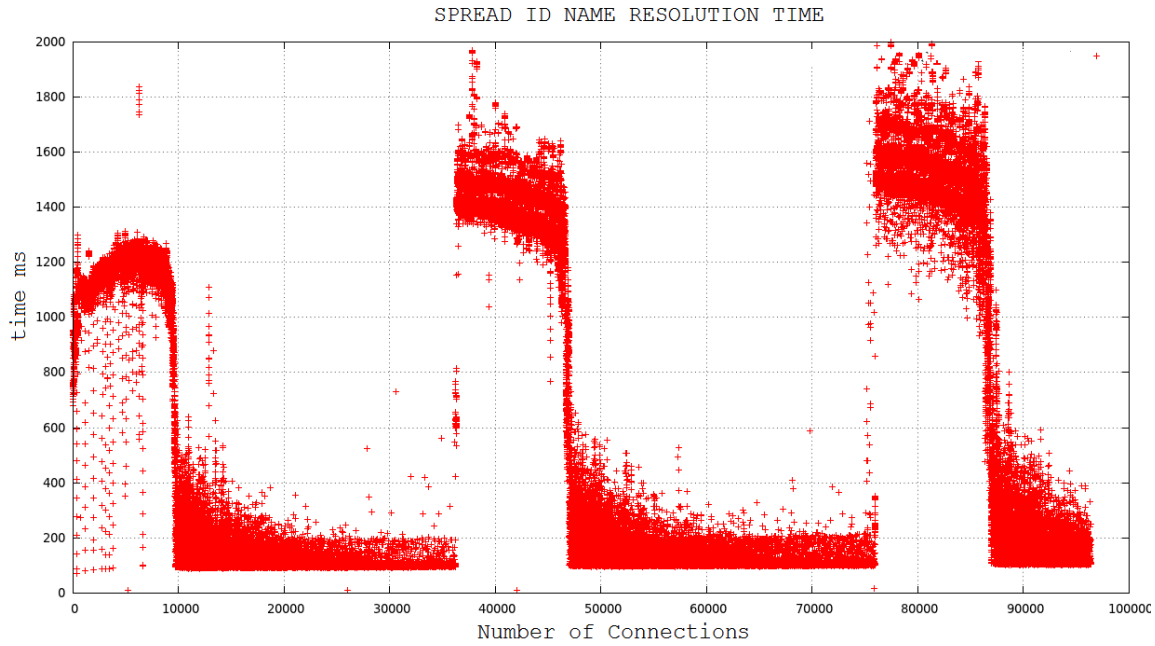

Figure 7B: Spread Identity Name Resolution Time as observed on client side. Seen are three increasing spikes, when compared to input load Figure 5A, we see that the query time is far more adversely affected by increase in query rate than the Static NAT case.

As observed in Figure 7A, the name query resolution time in case of Single NAT ranges from 2 to 30 milliseconds. In case of Spread identity, this time ranges from 150

milliseconds to 1.8 seconds.

Three distinct spikes can be observed in Figure 7A and 7B. These spikes represent the activation of a new subnet, a time when the SIGateway or SNAT Gateway observes high input load (ref. Figure 5B, 5A for input load characteristics). However, there is a difference in the nature of spikes in both graphs. In both cases, the highest response time indicates the point after which the number of input requests decrease.

In case of SIG (Figure 7B) we observe that a higher name query load on SIG affects response time more drastically. One reason for this is the serial nature of net filter. Installing NAT entries is a serial process, thus time taken to process earlier queries backlogs into response time of later queries.

4.4.4. SIG Packet Forwarding Time.

Packet Forwarding Time is the time required for a packet to enter the input interface and to leave to a proper destination through the destination interface. This measurement is very similar to the "Packet Forwarding Rate" measurement described in RFC2285. Usually the packet forwarding rate measured in Packets /Second (PPS) is number of packets that leave the firewall to its correct destination per second. This measurement is observed as a function of packet-size, traffic load, etc.

In our case, we are interested in exposing the effects caused by deploying the NAT subsystem. We observe the packet forwarding time as a function of NAT table size. We argue that the size of the packet (IP datagram size) does not directly affect NAT translation time. It may affect the packet translation time, but then this time will be constant across all packets of the same size.

The effect of IP datagram size is easily canceled out by measuring packets that are of similar size. These packets are TCP/IP handshake packets. We measure the first two packets of the three way TCP handshake. The first handshake packet can be seen as a worst case packet forwarding time because , like the SIG implementation, the first handshake packet requires more processing time for allocating resources for tracking future packets related to the same flow.

The baseline performance is the performance of the static NAT system where DNS and Host services are exposed through a simple static NAT entry.

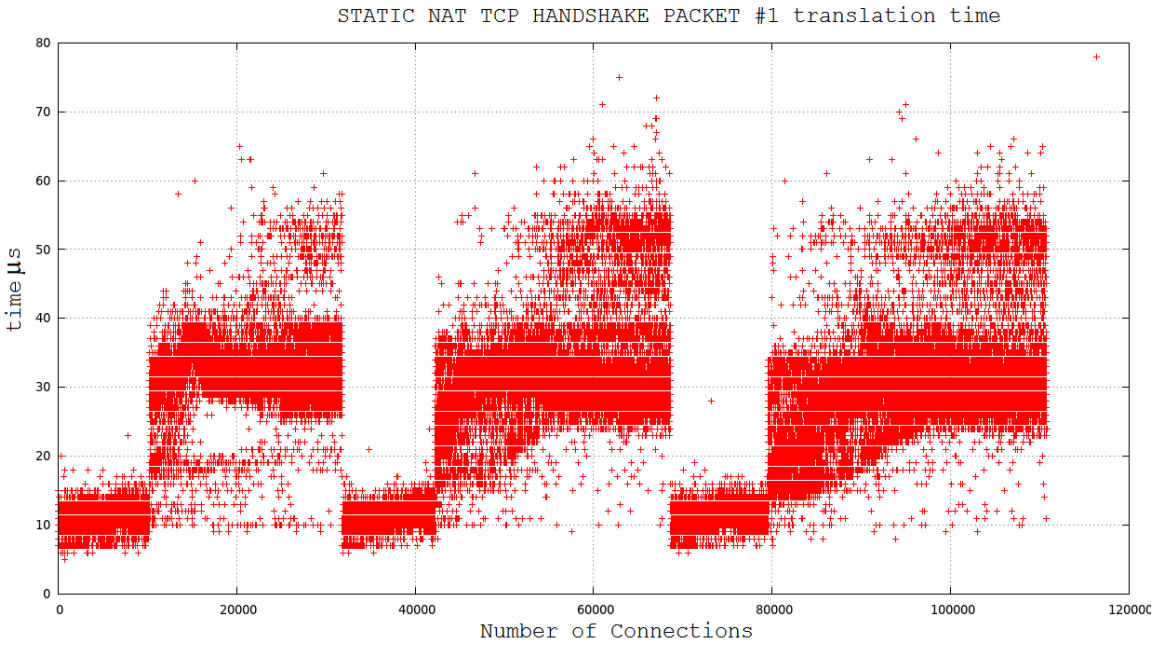

Figure 8A: Static NAT Packet forwarding time for TCP HANDSHAKE PACKET #1. Time taken by first handshake packet to traverse the Static NAT gateway.

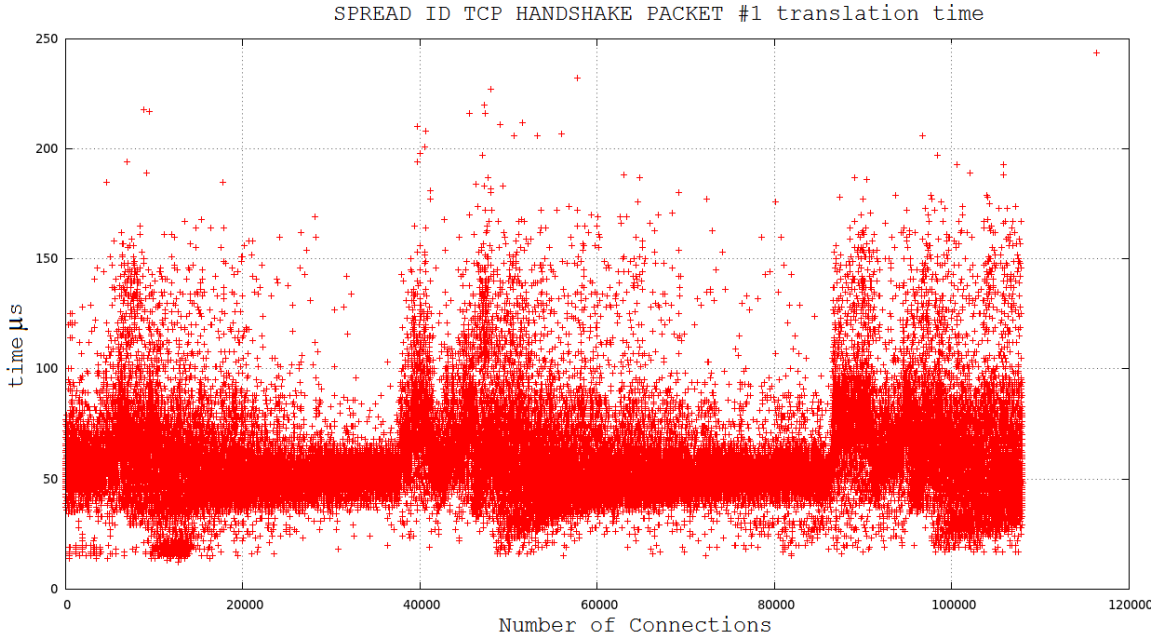

Figure 8B: Spread ID Packet forwarding time for TCP HANDSHAKE KET #1. Time required by first TCP handshake packet to traverse through the SIGateway.

Figures 8A and 8B show the packet translation time for the first packet of the TCP. This packet is generated on the subnet side when the POSIX connect() call is executed and forwarded through SIG to the internal host server.

In case of static NAT, we see a pattern. This pattern can be explained if we consider the corresponding DNS Name Resolution Graph (Figure 7A). Each area in the graph where the packet forwarding time is appreciably low, corresponds to the area in graph of Figure 7A where there was an increased name query response time. This must have lead in decreased number of connection handshakes. The range of packet forwarding time is anywhere from 5 to 60 micro seconds.

In case of SI, we compare it with corresponding DNS Name Resolution Graph (Figure 7B). Similar to the single NAT case, the areas where the packet forwarding time is appreciably low are areas where name resolution took appreciably more time. The range of packet forwarding time is anywhere from 15 to 150 microseconds.

Since these are first packets of the TCP handshake they encompass time required for setting up structures for connection tracking. The difference (5 to 60) and (15 to 150) microseconds indicates that the time required to set up connection tracking for Spread Identity is significantly higher than time required for Static NAT to set up connection tracking.

Figures 9A and 9B show the packet translation time for the second packet of the TCP 3 Way Handshake. These packets are generated by the host server and sent via SIG to the respective subnets. Note that since this is the second packet of the flow, the packet forwarding time excludes the connection setup time seen previously for first packet.

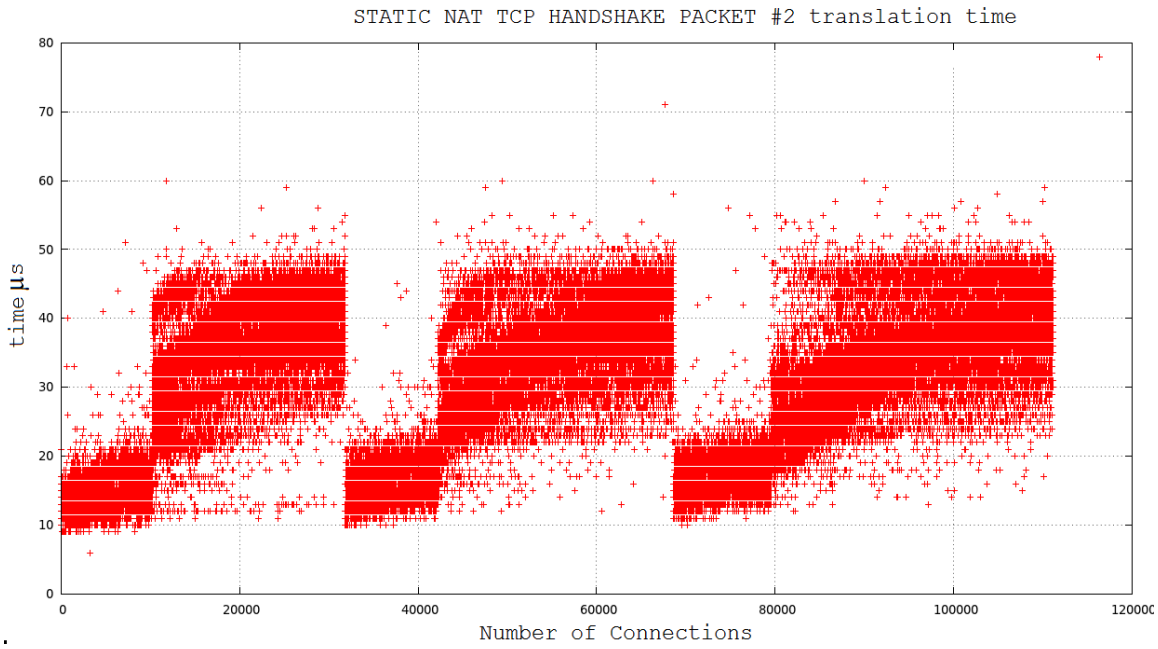

Figure 9A: Static NAT Packet forwarding time for TCP HANDSHAKE PACKET #2. Time taken by second handshake packet to traverse the Static NAT gateway.

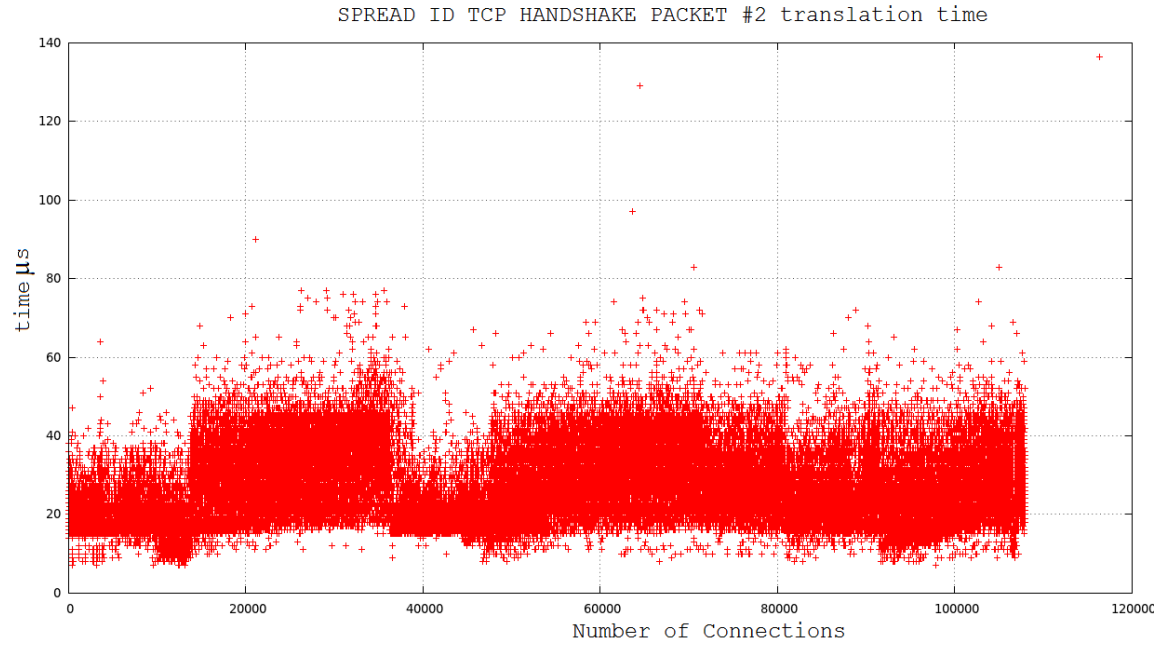

Figure 9B: Spread ID Packet forwarding time for TCP HANDSHAKE PACKET #2. Time taken by second handshake packet to traverse the SIGateway.

In case of Single NAT(ref. Figure 8A and 9A), the pattern is similar to that of the first graph, except in this case we are seeing reduced upper bound and slightly increased lower bound time. Packet forwarding time ranges between 10 and 50 microseconds.

In case of SI, the upper bound reduces more sharply (ref. Figure 8B and 9B). The range of packet forwarding time is now between 15 to 60 microseconds. This sharper reduction is simply an indication that in case of SI the time required to setup connection tracking is far more than time required for forwarding the second packet when connection setup is already over.

Since this packet represents a normal flow of a packet after connection establishment, we can consider these graphs to be a more realistic representative of the performance of the Spread identity and Static NAT systems, than Figure 8B which is a graph of forwarding time for the first packet of the handshake.

We observe a time difference of 10microseconds in worst case, and 5 microseconds in best case between SI and baseline Single NAT systems. We can hence, infer from this data, that there are no significant packet translation time difference between static NAT and Spread Identity System.

4.4.5 Memory requirements Single-NAT vs. Spread Identity

As seen in Figure 10 the Spread Identity maps a unique (client-host) address pair from the private network to a unique (client-host) address pair on the public network. However, when viewed simply as private host to public client or vice-a-versa, the mapping appears many-to-many. Shown in figure are two cases of spread identity deployment, with different blindfolding and pooled IP range.

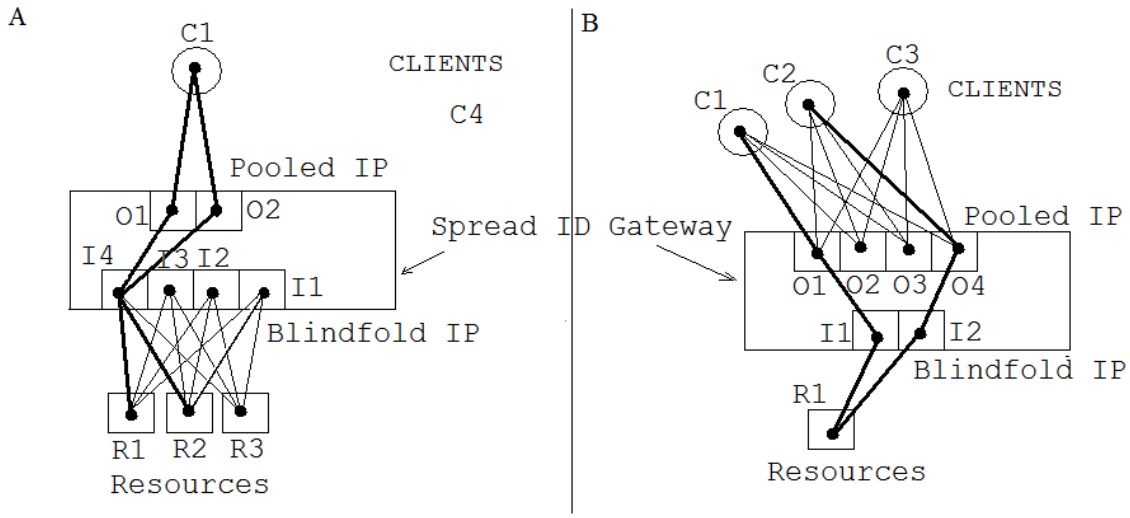

Figure 10: Effects of limiting blindfold IP and Pooled IP range. I1, I2...In: Internal

IP/interfaces. O1, O2...On: Outside IP/interfaces. C1, C2...Cn: clients. R1, R2...Rn: resources or hosts. The dark lines represent valid mappings, while the faint lines represent all possible communication pairs.

As seen in Figure 10 the number of mappings at any given time is limited by the minimum number of (client-host) IP address pairs. For example, in case (A) the number of NAT entries is limited by the number of unique pairs of (client IPs × pooledIPs). However, in case B it is limited by (resource IPs × BlindfoldIPs).

The number of 'Clients' is fixed and beyond control of an organization. The number of clients is a number bounded above by the number of publicly routable IPs. The number of 'Resources' is limited by the organization's hosting requirements, and as such independent from spread identity deployment's point of view. However, the pooled public IPs' range and Blindfold IPs range are in control of the Spread Identity, and they can be used to tune the performance of Spread identity.

To calculate an upper bound on number of entries, we need to know the number of pairs that can be formed on internal and external networks. The minimum of these numbers is the upper bound on number of NAT entries. It is easy to see that the number of (client-host) pairs on the public networks will be extremely large. E.g. In case of IP V4 this will be 4 billion times the number of pooled IPs. That means:

$$(\text{Public Clients} \times \text{pooled IPs}) >> (\text{internal Hosts} \times \text{blindfold IPs}).$$

We can conclude that the number of entries is bounded by number of hosts multiplied by number of blind folding addresses.

For example, if we allow internal blindfolding range of 250k IP addresses and we host 4 servers then the limit on the maximum number of NAT entries that this organization will need to handle is 1 million. Our platform takes 300 bytes of space per connection. Hence, roughly 300 MB memory will be required, assuming each client has one session only.

In case of single NAT, for 4 host servers, a straight forward view is to see that there will be only 4 NAT entries each of which exposes one of the 4 hosts from internal network to public network.

However, if we consider Connection Tracking [15], then these upper bounds do not hold. Instead, the number of connection tracking entries is simply equal to the number of connections established, for both SI and Static NAT cases.

Connection tracking [15] is a feature generally enabled for accounting and state full packet filtering; application level protocols many times deploy multiple connections. E.g. FTP uses one connection for control signaling and subsequent connections for data transfer. For complex applications such as FTP, connection tracking is required.

Memory consumption of connection-tracking NAT systems is dominated by connection tracking requirements rather than space required to store IP addresses.

4.5 Comparison Table.

We provide a tabular view that summarizes and compares measured values of all the parameters that we discussed above:

Table 1 Summary of experiments and results

Parameter Name	Single NAT Experiment	Spread ID Experiment	Observations
Input Load (Connections handled per second)	90 maximum connections handled per second	35 maximum connections handled per second	Single NAT can process more connection requests per second than Spread ID. This is because in Spread ID, name resolution process acts as a bottleneck.
Name Resolution Time	5 to 40 ms	90 to 2000 ms	Huge difference in response times. This parameter further exposes name resolution time difference between the two setups
Connection Establishment Time	1500 to 2500 us	1500 to 2500 us	After name resolution, TCP connection establishment time is almost similar.
TCP Handshake Packet #1 Translation time	6 to 60 us	10 to 150 us	Almost double worst case time difference between two setups. In both cases time varies more w.r.t input load, than on NAT table size.

TCP Handshake Packet #2 Translation time	10 to 50 us	10 to 60 us	Similar translation time observed in both setups. Again, time varies more w.r.t input load, than on NAT table size. This time is representative of the rest of the IP packet flow.
Memory Requirement	Equal to number of hosts/ number of connections	Min(private Host-blindfold IP pairs, pooled IP client pairs) OR Number of connections	In case of iptables, once the connection is established, NAT tables are not consulted, rather the connection tracking handles NAT data. As a result NAT table is a part of connection If connection tracking is used, each connection consumes 304 bytes of space per connection.

Chapter 5

DISCUSSION AND CONCLUSION

In this section we comment on the result of experiments explained in earlier chapter in relation to the work done by Phatak et al [3]. We compare implemented approaches, and discuss some open problems mentioned by Phatak *et al* [3]. We start by commenting on the SIR (I/O)-SIG name resolution process and its performance. Then we discuss the results of the NAT subsystem. Finally we discuss problems caused by spreading identities by harnessing DNS system followed by infrastructural changes that will be needed to implement SI. We end this chapter by providing conclusions of our work.

5.1 Performance of the SIR (I/O)/SIG Subsystem.

As described earlier, SI modifies the name resolution process to perform address remapping. Due to this change, the time it takes for name resolution increases to 1800 ms in worst case. In case of Single NAT, where there is no modification to the DNS name resolution process, we see a time times of 5 to 30ms.

However, we know that this problem has manifested due to inability of Net Filter (NAT) to install NAT entries in parallel, due to which response time of earlier queries stacks with response time of later queries, if later queries are received before earlier queries are processed. Moreover, in our implementation the first request from a particular IP takes extra time due to allocation of data structures in user space and selecting pooled and blindfolding IPs, which further aggravates the problem. To solve this problem we simply need to modify the net filter API to allow insertion of parallel entries, or use other alternative NAT systems.

5.2 Performance of the NAT subsystem.

From the experiments discussed earlier we conclude that there is no appreciable difference between single NAT and SI gateway performance (translation speed). The worst case time difference is 10 microseconds. As explained before, stateful NAT makes use of Connection Tracking and as a result the memory requirements are also comparable, since both Single NAT and SI require same amount of memory for

connection tracking. This is not true for non-stateful NAT systems, in case of non-stateful firewalls, SI will consume (blindfold IP ✕ number of hosts) NAT entries in worst case. However these systems are not popular.

Link speeds of up to 40Gbps on ISP level are common today. We assume availability of some mechanism that can provide NAT translation at link speed or faster. Products that perform bidirectional NAT mechanism at link speed, at speeds of up to 20Gbps per unit are already available from network infrastructure vendors like Cisco. For example, their Firewall Module for Catalyst 6500 can perform 256k simultaneous NATting operations at 5Gbps [13].

We do not believe that for small to medium sized organizations, NATting will negatively affect performance of the Spread Identity System.

5.3 DNS Client / DNS Server caching

DNS system was originally designed to use limited replication to prevent overload of any single server. This is achieved by caching answer records along the path of the query, such that subsequent queries can be resolved via cached answers as long as the cache is fresh, i.e. TTL > 0.

Spread Identity dynamically returns the IP address and disables caching of answer records by setting a 0 value on TTL field. As a result the last name query will not be cached along the query path. For example, in case of www.sicompany.org, results of queries to root '.' and 'org.' domain will be cached, but an extra query will be needed to retrieve the IP of WWW host inside sicompany.org.

Phatak *et al* [3], propose studying effects of client side DNS caching on the size of NAT table. We believe TTL caching will prove detrimental to our SI implementation. TTL values greater than one will create local caches across the query path. Clients who get their name resolved from these local caches will then be detected as malicious users on the organization side. Phatak *et al* [3] describes an approach whereby all static server mappings are stored in SIG is proposed to alleviate the above mentioned caching problem. A better solution instead might be to setup a caching DNS server that stores only the entries for authoritative name records. Algorithms for using dynamic TTL have been explored in [5]; some of them might be modified for use in our case.

However, as mentioned in RFC2694, a DNS Application Layer Gateway should have its DNS configured to set up a TTL field of 1 or 0 if the IP address translation is dynamic. SIG performs exactly this operation and can readily follow information on TTL values provided by this RFC.

5.4 Exposed DNS

To work, the DNS parent requires at least one record of a DNS server corresponding to its child domain. *e.g.* 'Org.' DNS servers will need entries for name servers of, say, xyz.org child domain. These records are called glue records since they provide an IP for a domain name that is not directly under their authority and thus avoid circular dependencies.

Because of this behavior, the organization that implements SI needs to expose at least one 'static' IP corresponding to its name server. In our system this exposed static IP belongs to the SIGateway. Moreover, parent DNS Servers unlike SIG do not take into account the source of the query before providing an answer to the name request. As a result protecting SIG from attacks is difficult. Phatak [2] argues that handling attacks on DNS servers is easier due to symmetrical load on the client and the DNS. Furthermore, instead of exposing a single name server, a range of addresses can be exposed and selectively blocked in an event of server overload. We believe these solutions are partial at best.

Application of SI at the ISP level might remove this problem, since in this case, the ISP Gateways (SIGs at ISP level) will address remap all the records including name servers. The remapping algorithm takes into account the source of the request. Hence the same system by which we protect non-name server resources can now be applied to protect name servers.

5.4 Infrastructural change

Last hop routers that are in ISP's domain may not allow the end clients to install filters. However, as suggested by Laxminariyanan *et al* [6], one possibility to alleviate this problem is to install a proxy router between the last hop router and the SIG gateway. This router will be located on the premise of the last hop router such that the bottleneck bandwidth will lie between SIG and proxy-router. This proxy-router can then be used to

install filters. Hence, no major network infrastructural change will be required.

Concerning client side infrastructure (client operating system etc), we find that some operating systems such as Windows XP cache DNS replies, while some others such as Linux do not cache the results of a DNS reply. Applications may again cache this at the application level. The TTL field of records is not necessarily exposed to applications, and as such, applications do not honor the TTL field. E.g. Internet Explorer caches DNS replies from 2 to 30 minutes.

Furthermore, most browsers today employ techniques such as DNS precache, whereby a domain name is resolved even before the user accesses the resource. These techniques might prove incompatible with our system. As such, clients might require adjusting some of their DNS related applications before properly accessing a resource hosted using SI. These configuration-changes can be easily pushed through a notice from the server side, and are completely configurable on the client side without any major updates. More importantly, making these changes has no major effect on the usual applications and processes that the client system might access.

5.6 Further Work

We believe that there are two main areas where further work needs to be done. 1) Deploying this SI framework implementation in the real world, to find out problems it might cause to the end users. This process will be time consuming, since problems cannot be found out without having sufficient code coverage (code execution of all software programs that interact with SI). 2) Test against various DDOS attack templates using Deter Lab's Framework. Due to Deter Lab's structure, this will make our results readily comparable with other defense mechanisms that have used Deter Lab.

5.7 Conclusion

We have provided a novel implementation of Spread Identity framework using well published Ideas in RFC2663 and RFC2694. We test this new system for perceived problems caused by incorporating NAT system and modifying of the name resolution process.

We find that NAT system is unlikely to cause problems for small to medium (up to 100,000 simultaneous connections) organizations that might deploy SI. We also

conclude that memory requirements of a NAT system are dominated by Connection Tracking rather than by NAT rules. We also notice that Net Filter (the leading open source NAT solution) is not well suited for the purpose of SI due to lack of reentrant APIs, which cause abysmal DNS name resolution performance if DNS requests are generated at a rate higher than the rate in which they can be processed.

Based on individual conclusions above, we believe that we can implement the SI framework and deploy it successfully at an organizational level for small to medium sized organizations. We also believe that due to the strict packet classification performed by SI, we can avoid flood packets from reaching bottle neck bandwidth, by using filters on last hop routers or proxy routers.

REFERENCES

[1] Jelena Mirkovic and Peter Reiher. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.* 34, 2 (April 2004), 39-53. DOI=10.1145/997150.997156 http://doi.acm.org/10.1145/997150.997156

[2] Dhananjay S. Phatak, "Spread-Identity mechanisms for DOS resilience and Security.," securecomm, pp.23-34, First International Conference on Security and Privacy for Emerging Areas in Communications Networks (SECURECOMM'05), 2005

[3] Dhananjay Phatak, Alan T. Sherman, Bhushan Sonawane, Vivek G. Relan .2011. Spread Identity: A New Dynamic Address Remapping Mechanism for Anonymity and DDoS Defense. Submitted to Journal of Computer Security January 20, 2011
URL: N/A

[4] Abbass Asosheh, Dr. and Naghmeh Ramezani. 2008. A comprehensive taxonomy of DDOS attacks and defense mechanism applying in a smart classification. *W. Trans. on Comp.* 7, 4 (April 2008), 281-290.

[5] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. 1999. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing* 3, 3 (May 1999), 28-39. DOI=10.1109/4236.769420 http://dx.doi.org/10.1109/4236.769420

[6] Karthik Lakshminarayanan, Daniel Adkins, Adrian Perrig, and Ion Stoica. 2004. Taming IP packet flooding attacks. *SIGCOMM Comput. Commun. Rev.* 34, 1 (January 2004), 45-50. DOI=10.1145/972374.972383
URL : http://doi.acm.org/10.1145/972374.972383

[7] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. 2004. Internet indirection infrastructure. *IEEE/ACM Trans. Netw.* 12, 2 (April 2004), 205-218. DOI=10.1109/TNET.2004.826279 http://dx.doi.org/10.1109/TNET.2004.826279
Jelena Mirkovic, Erinc Arikan, Songjie Wei, Sonia Fahmy, Roshan Thomas, and Peter Reiher. 2006. Benchmarks for DDoS defense evaluation. In *Proceedings of the 2006 IEEE conference on Military communications* (MILCOM'06). IEEE Press, Piscataway, NJ, USA, 3527-3536.

[8] Jelena Mirkovic, Erinc Arikan, Songjie Wei, Sonia Fahmy, Roshan Thomas, and Peter

Reiher. 2006. Benchmarks for DDoS defense evaluation. In *Proceedings of the 2006 IEEE conference on Military communications* (MILCOM'06). IEEE Press, Piscataway, NJ, USA, 3527-3536.

[9] Berkeley Internet Name Domain , BIND9
URL : http://www.isc.org/software/bind

[10] Netfilter and Iptables: A Structural Examination
URL: http://www.sans.org/reading_room/whitepapers/firewalls/netfilter-iptables-structural-examination_1392

[11] REMOTELY TRIGGERED BLACK HOLE FILTERING—DESTINATION BASED AND SOURCE BASED
URL : http://www.cisco.com/web/about/security/intelligence/blackhole.pdf

[12] IP_CONNTRACK AND TOOLS
URL : http://conntrack-tools.netfilter.org/manual.html

[13] Cisco Firewall Services Module for Cisco Catalyst 6500 and Cisco 7600 Series
http://www.cisco.com/en/US/prod/collateral/modules/ps2706/ps4452/product_data_sheet0900aecd803e69c3.html

[14] Bind9 Deployment Report
URL : http://dns.measurement-factory.com/surveys/201010/

[15] Netfilter's connection tracking system
URL : http://people.netfilter.org/pablo/docs/login.pdf

[16] P. Srisuresh. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, Internet Engineering Task Force, August 1999.
URL : ftp://ftp.ietf.org/rfc/rfc2663.txt

[17] P. Srisuresh. DNS extensions to Network Address Translators (DNS_ALG). RFC 2694, Internet Engineering Task Force, September 1999.
URL : ftp://ftp.ietf.org/rfc/rfc2694.txt

[18] Bushan Sonawne. Spread Identity Paradigm : A Distributed Denial of Service (DDoS) resilient framework . May 2010. UMBC M.S Thesis Submission.
URL : http://contentdm.ad.umbc.edu/u?/ETD,24320

[19] Amol Dhawalbhakta. Demonstrate the application of the concept of Spread Identity to enhance security on internet against DDOS attacks

URL:

http://contentdm.ad.umbc.edu/cdm4/item_viewer.php?CISOROOT=/ETD&CISOPTR=222&CISOBOX=1&REC=1

APPENDIX A Abbreviations and Acronyms

ALG	Application Layer Gateway
API	Application Programming Interface
DDoS	Distributed Denial of Service
DMZ	De militarized Zone
DNS	Domain Name System
FOSS	Free Open Source Software
FTP	File Transfer Protocol
I3	Internet Indirection Infrastructure
IETF	Internet Engineering Task Force
IP	Internet Protocol
IRC	Internet Relay Chat
ISP	Internet Service Provider
JVM	Java Virtual Machine
NAT	Network Address Translation
RFC	Request For Comment
RSIP	Realm Specific IP
SI	Spread Identity
SIG	Spread Identity Gateway
SIR	Spread Identity Resolver
SIR(I/O)	Spread Identity Resolver (Input / Output)
SIRI	Spread Identity Resolver Input
SIRO	Spread Identity Resolver Output
SNAT	Single Network Address Translation
TCP	Transmission Control Protocl
TTL	Time To Live

APPENDIX B Source Code Snippets

```
//Implementation of the address remapping algorithm. This is the core
function that runs on the SIG proxy //and handles the DNS replies and
responses.

      public void addressRemmpper() {

            DatagramPacket pck2=null;
            short[] randExIP=null;
            short[] randInIP=null;
            Message m;
            while(true){

            //0.try to get a packet from blocking queue.wait if theres
nothing to produce
            try{pck2=produce.take();}catch(Exception
e){System.out.print("\nProblem in take() "+e);continue;}

            //1.save original ip:port
            InetAddress pck1ipAddr=pck2.getAddress();
            int pck1port=pck2.getPort();
            long
sourceIP=IPHelpers.otLong(IPHelpers.inetToOctets(pck1ipAddr));
            boolean
considerSourceInternal=IPHelpers.isIpInternal(sourceIP);


            //3.Query the auth server first.
            //dont bother doing anything if theres an error in packet
send/recv.
            byte b[]=pck2.getData();
            byte pck2clone[]=new byte[256];System.arraycopy(b,
0,pck2clone,0,b.length);
            if(!queryDNSServers(IPHelpers.intDnsIP,
IPHelpers.intDNSPort, pck2)){
                  System.out.print("\nSystem cannot work without
internal dns srvr!");continue;
            }




            //4.modification has for combinations
            // a. external query--->internalIP. : this is normal, auth
type , no recursion
            // b. internal query--->internalIP. : this is normal,
recursive server.
            // c. internal query--->externalIP. : this is nomal ,
recursive server.
            // d. external query--->externalIP. : not allowed , will
return empty record. probably resource abuse.
            try{
                        if(considerSourceInternal){
                              //natting for internal intries.
                              m=new Message(pck2.getData());
```

```
                                Record[] answerRecords=
m.getSectionArray(Section.ANSWER);
                                //System.out.print("\nRecSiz
"+answerRecords.length);

                                //need to do si stuff only if the dest ip
is not internal
                                //ie. internal dns answer = nothing = 0.
                                if(answerRecords.length==0){
                                      pck2=new DatagramPacket(pck2clone,
pck2clone.length);

      if(!queryDNSServers(IPHelpers.extDnsIP, IPHelpers.extDNSPort,
pck2))continue;

                                                m=new
Message(pck2.getData());
                                                answerRecords=
m.getSectionArray(Section.ANSWER);
                                                //DEBUG
System.out.print("\nNew record SIze:"+answerRecords.length);

                                                int
i=0;while(i<answerRecords.length &&
answerRecords[i].getType()!=Type.A)i++;
                                                //System.out.print("\ndata in
answer record ext query resutl: "+answerRecords[i].rdataToString());

                                                //there was a valid reply. so
we have an external ip.
                                                if(i<answerRecords.length &&
answerRecords.length!=0){
                                                      InetAddress
origDNSRespIP=((ARecord)answerRecords[i]).getAddress();

                                                      //TODO: DELETE
COMMUNICATION ENTRIES>

      /*if(origDNSRespIP.equals(IPRanges.communcationAvailable.get(pck1
ipAddr)))//duplicate request.

      {System.out.println("Dropping packet, path already
established!");continue;}//duplicate request.

      IPRanges.communcationAvailable.put(pck1ipAddr,origDNSRespIP);
                                                       */
                                                      //get random external
ip to use in place of origdnsip.

      randInIP=IPHelpers.internal.getRandomIP(pck1ipAddr);//get random
ip

      if(randInIP==null){System.out.print("\nYou have reached limit of
web access");continue;}

      randExIP=IPHelpers.external.getRandomIP(origDNSRespIP);
```

```
      if(randExIP==null){System.out.println(externalRangeFullError+orig
DNSRespIP.getHostAddress());continue;}


      //System.out.println(pck1ipAddr.getHostAddress()+ "xdata in
answerrecord ext query
resutl:"+origDNSRespIP.getHostAddress()+"randIn"+IPHelpers.valueOf(rand
InIP)+"randOut"+IPHelpers.valueOf(randExIP));

                                                  //.PREROUTING
(inCli_IP,RandIntIP)--->(inCli,origDNSIP)
                                                  //.POSTROUTE (inCli_IP
to origDNSIP)--change-->(randExtIp to orDNSIP).

      sendCommand(true,null,null,"iptables -t nat -A PREROUTING -s
"+pck1ipAddr.getHostAddress()+" -d "+IPHelpers.valueOf(randInIP)+" -j
DNAT --to "+origDNSRespIP.getHostAddress(),true);

      sendCommand(false,null,null,"iptables -t nat -A POSTROUTING -s
"+pck1ipAddr.getHostAddress()+" -d "+origDNSRespIP.getHostAddress()+" -
j SNAT --to "+IPHelpers.valueOf(randExIP),true);


                                                  //.PREROUTING
(inCli_IP,RandIntIP)--->(inCli,origDNSIP)
                                                  //.POSTROUTE (inCli_IP
to origDNSIP)--change-->(randExtIp to orDNSIP).

      sendCommand(true,pck1ipAddr,randInIP,"iptables -t nat -D
PREROUTING -s "+pck1ipAddr.getHostAddress()+" -d
"+IPHelpers.valueOf(randInIP)+" -j DNAT --to
"+origDNSRespIP.getHostAddress(),false);

      sendCommand(false,origDNSRespIP,randExIP,"iptables -t nat -D
POSTROUTING -s "+pck1ipAddr.getHostAddress()+" -d
"+origDNSRespIP.getHostAddress()+" -j SNAT --to
"+IPHelpers.valueOf(randExIP),false);

                                                  //add randInternalIP as
the result ip, so that client is blindfolded and has
                                                  //no clue about the
external /original DNS IP.

      m.removeAllRecords(Section.ANSWER);
                                                  m.addRecord(new
ARecord(answerRecords[0].getName(),
answerRecords[0].getDClass(),answerRecords[0].getTTL(),
InetAddress.getByName(IPHelpers.valueOf(randInIP))),Section.ANSWER);

                                             }
                                             //else there are no records:
then we dont need to modify anything.
                               }
                               //else host was internal and we dont need
to modify orig reply.
                         }
```

```
                        else{//query source is external
                              m=new Message(pck2.getData());
                              Record[] answerRecords=
m.getSectionArray(Section.ANSWER);
                              int i=0;while(i<answerRecords.length &&
answerRecords[i].getType()!=Type.A)i++;

                              //there is an internal ip.
                              if(i<answerRecords.length){
                                    InetAddress
origDNSRespIP=((ARecord)answerRecords[i]).getAddress();
                                    //. source ip: pck1sourceIP is
external

                                    //TODO:CHECK IF PATH aVAILABLEW.

      /*if(pck1ipAddr.equals(IPRanges.communcationAvailable.get(origDNS
RespIP)))//duplicate request.
                                    {System.out.println("Dropping
packet, path already established!");continue;}//duplicate request.

      IPRanges.communcationAvailable.put(origDNSRespIP,pck1ipAddr);
                                    */



                                    //get random external ip to use in
place of origdnsip.

      randExIP=IPHelpers.external.getRandomIP(pck1ipAddr);//get random
ip

      IPHelpers.external.releaseIP(pck1ipAddr,randExIP);

      if(randExIP==null){System.out.print("\nYou have reached limit of
web access");continue;}
                                    //TODO: Track array capacity
overlimit exception here.
                                    //TODO:Release placed for debug
purposes.

      randInIP=IPHelpers.internal.getRandomIP(origDNSRespIP);

      IPHelpers.internal.releaseIP(origDNSRespIP, randInIP);

      if(randInIP==null){System.out.println(internalRangeFullError+orig
DNSRespIP.getHostAddress());continue;}



                                    //.PREROUTE (extCli IP to randExtIP
) --change--> (extCli_IP to origInternalIP).
                                    //.POSTROUTE (extCli_IP to
origInternalIP)--change-->(randIntIP to origInternalIP).
                                    sendCommand(false,null,null,"sudo
iptables -t nat -A PREROUTING -s "+pck1ipAddr.getHostAddress()+" -d
"+IPHelpers.valueOf(randExIP)+" -j DNAT --to
```

```
"+origDNSRespIP.getHostAddress(),true);
                                    sendCommand(true,null,null,"sudo
iptables -t nat -A POSTROUTING -s "+pck1ipAddr.getHostAddress()+" -d
"+origDNSRespIP.getHostAddress()+" -j SNAT --to
"+IPHelpers.valueOf(randInIP),true);


                                    //PREROUTE (origInternalIP to
randIntIP) --change--> (origInternalIP to extCli_IP)
                                    //.POSTROUTE (extCli_IP to
origInternalIP)--change-->(randIntIP to origInternalIP).

      //sendCommand(false,pck1ipAddr,randExIP,"sudo iptables -t nat -D
PREROUTING -s "+pck1ipAddr.getHostAddress()+" -d
"+IPHelpers.valueOf(randExIP)+" -j DNAT --to
"+origDNSRespIP.getHostAddress(),false);

      //sendCommand(true,origDNSRespIP,randInIP,"sudo iptables -t nat -
D POSTROUTING -s "+pck1ipAddr.getHostAddress()+" -d
"+origDNSRespIP.getHostAddress()+" -j SNAT --to
"+IPHelpers.valueOf(randInIP),false);

                                    //add modify packet etc.
                                    m.removeAllRecords(Section.ANSWER);
                                    m.addRecord(new
ARecord(answerRecords[0].getName(),
answerRecords[0].getDClass(),answerRecords[0].getTTL(),
InetAddress.getByName(IPHelpers.valueOf(randExIP))),Section.ANSWER);
                                }
                                   //else external cli query external
host, we dont care.
                            }

                byte nb[]=m.toWire();
                  pck2=new DatagramPacket(nb, nb.length);

            }catch(Exception e){e.printStackTrace();}

            //6.send back reply
            pck2.setAddress(pck1ipAddr);
            pck2.setPort(pck1port);
            try{dsockProxy.send(pck2);}catch(Exception
e){e.printStackTrace();}
        }
      }
```