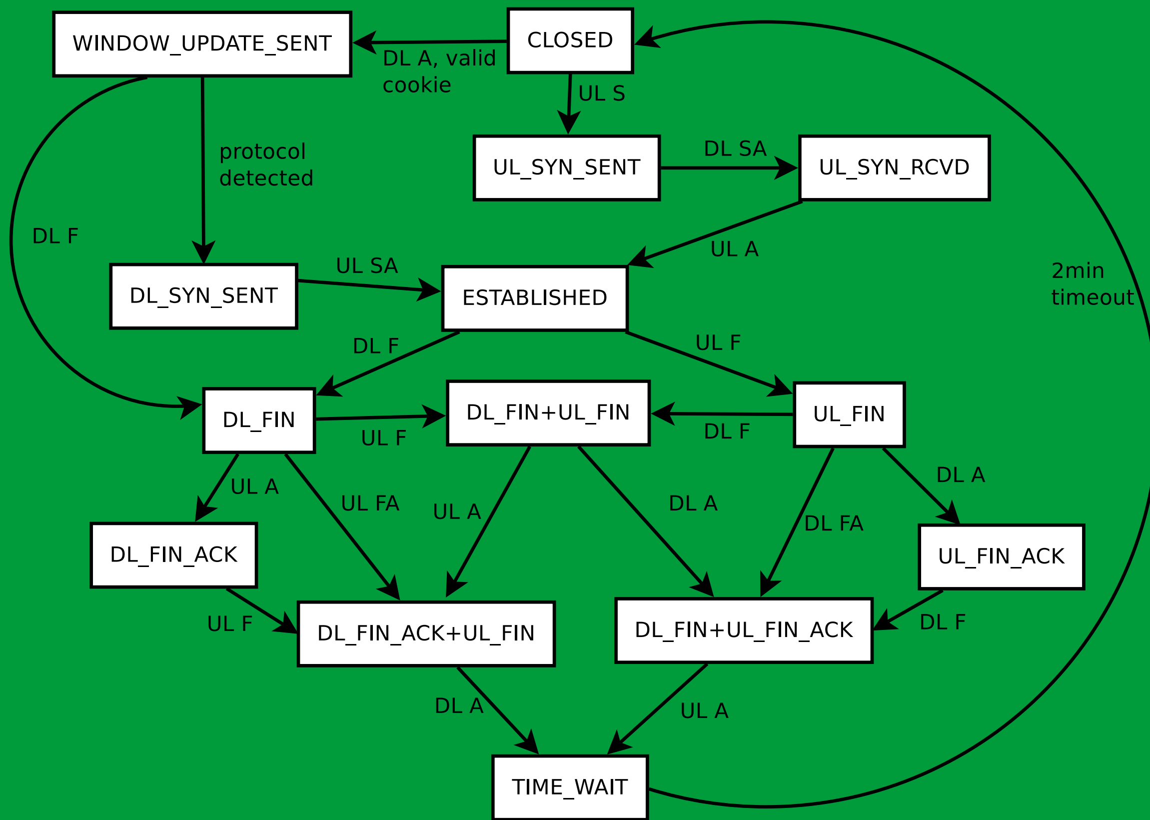
Department of Communications and Networking

Application Layer Network Address Translation

Juha-Matti Tilli



Aalto University

LICENTIATE THESIS

Aalto University
LICENTIATE THESIS 2022

Application Layer Network Address Translation

Juha-Matti Tilli

Thesis submitted for examination for the degree of Licentiate of Science in Technology.
Otaniemi, 08 Sep 2022

Supervisor: professor Raimo Kantola
Advisor: professor Raimo Kantola
Examiner: professor Jussi Kangasharju

Aalto University
School of Electrical Engineering
Department of Communications and Networking

Aalto University

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Abstract

Author
Juha-Matti Tilli

Title
Application Layer Network Address Translation

School School of Electrical Engineering

Department Department of Communications and Networking

Research field Networking Technology **Code** S041Z

Supervisor professor Raimo Kantola

Advisor professor Raimo Kantola

Examiner professor Jussi Kangasharju

Level Licentiate thesis **Date** 08 Sep 2022 **Pages** (0)+x+75 **Language** English

Abstract

The most important issues facing the Internet are lack of security and address space exhaustion. Flooding attacks are easy today due to for example the design of transmission control protocol (TCP). QUIC may offer some remedy for new applications but it is unlikely to replace TCP in all legacy applications. Most modern TCP endpoints have support for SYN cookies, but middle-points like firewalls may not be able to protect themselves.

Address space of Internet protocol version 4 (IPv4) is not sufficient to give even one address to every Internet user today. At the same time, Internet of Things (IoT) is gaining foothold, meaning the device count is expected to exceed Internet user count by orders of magnitude.

To counter these problems, network address translation (NAT) has emerged as a solution. However, NAT has poor characteristics when used on the server side.

In this thesis, SYN cookies and their middle-point implementations within SYN proxy are refined compared to the state of the art. One result of the thesis is a multithreaded user space TCP SYN proxy that is able to protect servers and legacy firewalls against SYN flooding attacks. It is also able to act as the protecting component of a novel firewall called Realm Gateway (RGW) that uses standard domain name system (DNS) queries for NAT traversal.

Additionally, an application layer NAT (AL-NAT) technique allowing NAT traversal for protocols where the client sends the first message containing host name of the server is defined and implemented. As a lightweight implementation of AL-NAT, a component supporting AL-NAT without security policy is provided.

Finally, for protocols such as secure shell (SSH) that are incompatible with AL-NAT, carrier grade TCP proxy (CG-TP) is implemented to work along with AL-NAT and verified to work with OpenSSH and other applications with similar characteristics.

Keywords transmission control protocol, network address translation, SYN proxy, application layer gateway

A! Aalto-yliopisto

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tiivistelmä

Tekijä
Juha-Matti Tilli

Työn nimi
Sovellustason verkko-osoitteiden muunnos

Korkeakoulu Sähkötekniikan korkeakoulu

Laitos Tietoliikenne- ja tietoverkkotekniikan laitos

Tutkimusala Tietoverkkotekniikka **Koodi** S041Z

Valvoja professori Raimo Kantola

Ohjaaja professori Raimo Kantola

Tarkastaja professori Jussi Kangasharju

Työn laji Lisensiaatintutkimus **Päiväys** 08.09.2022 **Sivuja** (0)+x+75 **Kieli** englanti

Tiivistelmä

Suurimmat Internetin ongelmat tänä päivänä ovat tietoturvan puute ja osoiteavaruuden ehtyminen. Palveunestohyökkäykset ovat helppoja tänä päivänä johtuen protokollan transmission control protocol (TCP) toteutuksesta. QUIC saattaa tarjota parannuksia uusille sovelluksille mutta ei korvanne TCP:tä kaikissa vanhoissa sovelluksissa. Useimmat modernit TCP-toteutukset sisältävät tuen SYN cookieille, mutta keskipisteiden toteutukset kuten palomuurit eivät välttämättä kykene puolustamaan itseään.

Protokollan Internet protocol versio 4 (IPv4) osoiteavaruus ei riitä antamaan edes yhtä osoitetta jokaiselle Internetin käyttäjälle tänään. Samanaikaisesti esineiden Internet saa jalansijaa, tarkoittaen että Internetin laitemäärän odotetaan kasvavan kertaluokkia suuremmaksi kuin Internetin käyttäjämäärän.

Näiden ongelmien ratkaisuun verkko-osoitteiden muunnos on ilmestynyt ratkaisuksi. Kuitenkin verkko-osoitteiden muunnoksella on huonoja ominaisuuksia, jos sitä käytetään palvelinpuolella.

Tässä lisensiaatintyössä SYN cookieita ja niiden keskipistetoteutuksia SYN proxyssä kehitetään eteenpäin. Yksi työn tulos on monisäikeinen käyttäjäosoiteavaruudessa pyörivä TCP SYN proxy, joka osaa suojata palvelimia ja palomuureja SYN flood -palvelunestohyökkäyksiltä. Se voi myös toimia suojaavana komponenttina realm gatewaylle (RGW), joka on uusi palomuuri joka käyttää tavallisia domain-nimikyselyitä verkko-osoitteiden muunnoksen läpäisyyn.

Lisäksi sovellustason verkko-osoitteiden muunnos (application layer network address translation, AL-NAT) määritellään ja toteutetaan. AL-NAT toimii protokollille joissa asiakasohjelma lähettää ensimmäisen viestin sisältäen palvelinkoneen nimen. Kevyenä AL-NAT toteutuksena tarjotaan komponentti, joka tukee AL-NAT:ia muttei minkäänlaista tietoturvasäännöstöä.

Lopuksi protokollille, kuten secure shell (SSH), jotka eivät toimi AL-NAT:n kanssa tarjotaan operaattoritason TCP proxy (carrier grade TCP proxy, CG-TP). CG-TP:n varmistetaan toimivan esimerkiksi OpenSSH-etäkäyttöohjelman kanssa.

Avainsanat transmission control protocol, verkko-osoitteiden muunnos, SYN proxy, sovellustason yhdyskäytävä

Preface

This thesis has been done in the Department of Communications and Networking of School of Electrical Engineering at Aalto University. I started doctoral studies officially in late 2015, although my part-time study leave started only in September 2016. In March 2017, I started to implement what eventually became PPTK and nmsynproxy. AL-NAT was invented in June 2018, and code implementation based on nmsynproxy quickly started.

I would like to thank the thesis advisor and supervisor Raimo Kantola for having belief in me despite the fact that I originally studied micro- and nanotechnology and not communications engineering in my Master's degree. Also big thanks to him for offering a part-time job at Aalto University and for Nokia Bell Labs for accepting a study leave. Alas, a second study leave was not accepted so I left Nokia to work at Foreca as my main job, where management was more tolerant of a part-time job at Aalto University.

Many interesting discussions with Jesus Llorente Santos are worthy of an acknowledgement too. Those discussions have genuinely challenged my approaches, and the fact that my approaches have stood unchanged can be probably considered a strength of the approaches.

Also big thanks for Maria Riaz for doing her Master's thesis enthusiastically. As an advisor, I had to answer various questions related to application layer gateways often, but this thesis would not be the same without those constant questions. Those questions have affected my thought process and hopefully have positively affected the clarity of this thesis.

I would also like to thank Microsoft for a generous award based on discovery of the FragmentSmack[40] IP fragmentation attack and Emil Aaltonen Foundation for a grant.

Espoo, September 08, 2022

Juha-Matti Tilli

Author's contribution

The author is the sole author of ldpairwall, nmsynproxy (apart from very minor control scripts by Jesus Llorente Santos), PPTK (apart from very minor contributions by Vladis Dronov and apart from some existing code that was used for testing comparisons with custom code), abce, stirmake, cghcpcli and YaLe. The author was also the advisor to the Master's thesis of Maria Riaz where a Python-based application layer gateway using YaLe was implemented to extend realm gateway and co-authored the conference paper about this work. Application layer gateway can achieve the same NAT traversal that AL-NAT achieves but terminates both sides of the connection at the middlebox, thus causing the endpoint application to see the middlebox as the originator of the connection, hiding the IP address of the true originator.

The author is the inventor of AL-NAT. However, the name AL-NAT was proposed by Raimo Kantola after being told of the invention. Originally the name was NG-AANAT (next generation application aware NAT), but the AL-NAT is significantly better name for such a sophisticated technology.

This manuscript has been fully written by the author.

List of acronyms

5G	Fifth generation of mobile telecommunications
ACK	ACKnowledge bit in TCP segment
ALG	Application layer gateway
AL-NAT	Application layer NAT
CG-TP	Carrier grade TCP proxy
CPU	Central processing unit
DDoS	Distributed DoS attack
DF	Don't fragment bit in IPv4 header
DHCP	Dynamic host configuration protocol
DIX	Digital equipment corporation (DEC), Intel, Xerox
DL	Downlink (from Internet to local network)
DNS	Domain name system
DoS	Denial of service attack
DS	Differentiated services field in IP header
FIN	FINalize bit in TCP segment
FTP	File transfer protocol
HTTP	Hypertext transfer protocol
HTTPS	Hypertext transfer protocol, secure
ICMP	Internet control message protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet engineering task force
IMAP	Internet mail access protocol
I/O	Input/output
IP	Internet protocol
IPv4	IP version 4
IPv6	IP version 6
LAN	Local area network
LLC	Logical link control
LRU	Least recently used (cache dropping policy)

MAC	Media access control (usually referring to address)
MPPS	Million packets per second
MTU	Maximum transmission unit
NAPT	Network address and port translation
NAT	Network address translation
NOOP	No operation
OpenSSH	The most popular open source implementation of SSH
PF	Packet filter (in OpenBSD)
PPTK	Packet processing toolkit (in Github/Aalto5G)
QUIC	A transport protocol designed to replace TCP
RFC	Request for comments (document series)
RGW	Realm gateway
RST	ReSeT bit in TCP segment
SACK	Selective acknowledgement option in TCP
SMTP	Simple mail transfer protocol
SNAP	Subnetwork access protocol
SNI	Server name indication (field in TLS)
SOCKS	The SOCKS proxy protocol
SSH	Secure shell
SSL	Secure sockets layer (referring to old versions of TLS)
SYN	SYNchronize bit in TCP segment
TCP	Transmission control protocol
TLS	Transport layer security
TTL	Time to live
UDP	User datagram protocol
UL	Uplink (from local network to Internet)
UNSAF	Unilateral self-address fixing
URI	Uniform resource identifier (roughly similar to URL)
URL	Uniform resource locator (roughly similar to URI)
VoIP	Voice over IP
WLAN	Wireless LAN
YaLe	Yet another lexer eliminator (in Github/Aalto5G)

Contents

Abstract ii

Tiivistelmä iii

Preface iv

Author's contribution v

List of acronyms vi

Contents viii

1. Introduction 1

2. Background theory and related work 4

2.1 Network address translation 4

2.1.1 NAT theory 4

2.1.2 NAT traversal 5

2.2 TCP window 8

2.3 SYN cookies 9

2.4 SYN proxy 10

2.5 Realm gateway 12

2.5.1 RGW firewall 13

2.5.2 RGW attacks 13

2.5.3 RGW attack mitigations 14

2.5.4 Application layer gateway 15

2.5.5 Shortcomings of RGW 16

3. Improved SYN cookies and nmsynproxy 18

3.1 Layer 2 SYN proxy 18

3.2 Header checksums 19

3.3 Hybrid SYN cookies 21
3.4 Hybrid SYN proxy 22
3.5 Fragment handling strategy 23
3.6 State machine 24
3.7 Improved hash limiting 26
3.8 Installation instructions 27

4. Application layer network address translation 29

4.1 Technical overview 29
4.1.1 Technical details 29
4.1.2 State machine 31
4.1.3 Packet loss and retransmissions 32
4.2 Protocol analysis 33
4.2.1 TCP 33
4.2.2 QUIC 34
4.2.3 Application-layer protocols 35
4.3 Carrier grade TCP proxy for unsupported TCP protocols . 38
4.3.1 Carrier grade TCP proxy in server-side middlebox 39
4.3.2 Carrier grade TCP client in client-side middlebox 40
4.3.3 New Internet architecture of cooperative firewalls 42
4.3.4 Carrier grade TCP client directly in client computer 42
4.4 Installation instructions 44

5. Theoretical analysis 45

5.1 nmsynproxy cryptography strength 45
5.2 nmsynproxy memory usage 46
5.3 ldpairwall memory usage 47
5.4 ldpairwall requirements analysis 48
5.4.1 RFC4787 48
5.4.2 RFC5382 52
5.4.3 RFC5508 54

6. Testing 59

6.1 nmsynproxy correctness tests 59
6.2 nmsynproxy performance tests 60
6.3 Protocol and hostname detection performance 61
6.4 HTTP and TLS header size study 62
6.5 AL-NAT traversal tests 65

6.5.1 Testing environment 65
6.5.2 Outgoing NAT connection 65
6.5.3 AL-NAT unencrypted connection 66
6.5.4 AL-NAT encrypted connection 66
6.5.5 HTTP CONNECT proxy 67
6.5.6 Port control protocol, TCP 67
6.5.7 Port control protocol, UDP 67

7. Conclusions **68**

7.1 Future work 69

Bibliography **71**

1. Introduction

Arguably the two most severe issues in today's Internet are shortage of IPv4[47] addresses along with very slowly progressing IPv6 transition during adoption of Internet of Things[2] and easiness of distributed denial of service (DDoS) attacks. In this thesis, solutions are proposed for both.

Internet is facing an IPv4 address shortage. To solve this, transition to IPv6[16] has been proposed starting from IPng[11] and the first specification of IPv6[15] but it is not without its drawbacks. For example, IPv6 multiplies IP header overhead by two, which is a genuine problem for VoIP[25] because low latency required necessitates frequent packets, and good lossy compression algorithms make those packets small. Furthermore, IPv6 is not backwards-compatible with IPv4, so devices having only IPv4 connectivity cannot reach IPv6-only servers. An already fully deployed partial solution for IPv4 address shortage is classless inter-domain routing (CIDR), but it cannot magically create a large address space, just allowing more efficient use of the already limited address space[23].

To solve the IPv4 address shortage, NAT[21] has been defined. Some variants of it allow masquerading multiple hosts behind a single IPv4 address, and are typically implemented as NAPT where both the source address and source port are translated. NAT solves some other issues in addition to the shortage of addresses. For example, NAT allows using static IP addresses in the local network even if the global address is dynamically distributed by e.g. DHCP[17]. Also, NAT increases security by making it impossible by design to connect to protected hosts because they are not addressable. Furthermore, NAT is an automatic form of routing, meaning one can set up an entire network behind a router without reconfiguring the routing tables of other routers either manually or with a routing protocol.

Where NAT falls short is when servers need to be run behind a NAT middlebox. Typically, the solution is manual port mapping, so that e.g.

port 80 of NAT middlebox is mapped to port 80 of a host behind the NAT middlebox. However, the NAT middlebox, typically having only 1 IP address, has only one TCP port 80. If one needs to run two web servers behind a NAT middlebox, the other has to use an alternative port such as 8080.

A solution to NAT traversal could be application layer gateway[53]. The incoming TCP connection is terminated at the NAT middlebox. The client sends its request, which at least for HTTP[22] and TLS[51] contains the host name of the server in plain text. This information is used to open another TCP connection between the NAT middlebox and the server in private address space. This termination of TCP connection at the NAT middlebox, however, requires lots of resources for large connection counts, and thus could make the NAT middlebox vulnerable to an attack. In particular, by terminating the TCP connection at the NAT middlebox, the data may take up lots of kernel memory used for retransmit buffers.

There is an advanced technique originally designed to protect vulnerable endpoint hosts from DDoS attacks that is called SYN proxy. This SYN proxy first temporarily terminates the opening of a TCP connection at the SYN proxy middlebox. When the connection is fully open, the TCP connection is handed off to the protected host. In this thesis, a fast netmap-based SYN proxy is implemented as a layer 2 inline element. The SYN proxy uses SYN cookies to protect itself from a DDoS attack. The author has reason to believe the SYN cookie and SYN proxy implementations in this SYN proxy are the most advanced and sophisticated implementations to date.

There is a SYN proxy in Linux netfilter subsystem, but the Linux implementation of SYN proxy is deficient in handling closed port RST responses. Also, it only works as a layer 3 device without advanced configuration tricks involving Open vSwitch that J. Llorente Santos discovered and that allow using layer 2 operation.

If the NAT middlebox is given multiple IPv4 addresses, it can be made into a component called RGW[34] that uses standard DNS[37, 38] queries to an integrated DNS server as indications of where to forward the connection. However, this system is vulnerable to SYN flooding[20] and DNS flooding attacks. The solution against DNS flooding is a reputation system. The solution against SYN flooding is a SYN proxy, such as the component implemented in this thesis. In theory, such a system can also work given only one IPv4 address, but in practice many simultaneous incoming

connections may end up being a troublesome situation.

As a further refinement on the SYN proxy technique, a NAT middlebox called AL-NAT is implemented. This middlebox temporarily terminates (proxies) the opening of a TCP connection at the middlebox. It then waits for the client to send its first data packets, extracts the hostname from the first data packets, and hands off the connection to the correct protected host.

This thesis is organized as follows. In Chaper 2, background theory and related work are discussed. Chapter 3 introduces nmsynproxy with its improved SYN cookie and SYN proxy support. Chapter 4 discusses AL-NAT. Chapter 5 includes various theoretical analyses of the implemented solutions. Chapter 6 describes what kind of testing has been performed for the components. Finally, Chapter 7 concludes this thesis.

2. Background theory and related work

This chapter first reviews the theory behind network address translation (NAT). It is an existing solution that builds the background for part of this work. Then SYN cookies and SYN proxy are discussed as denial of service (DoS) mitigation mechanisms. Finally, realm gateway (RGW) is analyzed in detail for its role in NAT traversal.

2.1 Network address translation

This section discusses network address translation. For more details, the reader can refer to e.g. [69] for an excellent historical description of NAT.

2.1.1 NAT theory

Network address translation (NAT) is a solution already widely deployed to mitigate IPv4 address exhaustion problem. In typical deployments, using NAT means that a network uses private addresses not visible to the Internet. Whenever a packet from this network is sent to the Internet, the source IP address is changed to be a publicly available address. Typically, NAT is implemented as network address and port translation (NAPT), which translates the TCP[48] or UDP[45] ports too. Protocols not having ports do not obviously work with NAPT. The reason for preferring NAPT is that it allows one to masquerade multiple private IPv4 addresses behind a single public IPv4 address, therefore allowing more efficient use of IPv4 addresses.

A good NAT implementation translates not only direct TCP/UDP packets but also ICMP[46] responses to TCP/UDP packets. However, there are many flawed implementations of firewalls and/or NAT that do not translate ICMP responses but rather drop them.

NAT may be tuned for security or for connectivity. A connectivity-tuned

NAT may prefer retaining port numbers and may open a port for all incoming connections after an outgoing connection has been established. For example, a connectivity-tuned NAT may open the connection 10.1.2.3:1500 (local internal address) → 192.0.2.5:80 (remote address). Then it translates the private IP address and port pair 10.1.2.3:1500 to public IP address and port pair 192.0.2.2:1500. If the port 1500 has not been already used, the port number is retained. Then, when somebody wants to connect 192.0.2.6:2000 → 192.0.2.2:1500, a connectivity-tuned NAT will forward the incoming connection to 10.1.2.3:1500. Essentially, after a NAT has established a mapping, it is remembered, it is not dependent on the other endpoint (*endpoint-independent mapping*) and also incoming connections are accepted for this mapping. This connectivity-tuned NAT is preferable for carrier grade NAT[54] because the job of the carrier is not to offer security but rather connectivity.

In contrast, a home router may very well have the property that once a mapping is established, it is used only for outgoing connections and never for new incoming connections. This allows better security than a connectivity-tuned NAT.

Several networks have been decided to not be allocated in the Internet: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/24[50]. There is also the network for carrier grade NAT[54] which is 100.64.0.0/10[63]. These networks are guaranteed not to be allocated in the Internet, so a NAT solution can choose to use any of these networks as private addresses. However, the carrier grade NAT network is meant to be used by the carrier, so end-users should not choose addresses in this range.

2.1.2 NAT traversal

Because NAT masquerades multiple private IPv4 addresses behind a single public IPv4 address, incoming connections cannot always be handled. Such incoming connections may be needed for e.g. peer-to-peer Internet telephony applications and peer-to-peer gaming. In such applications, both peers can be behind a NAT. Thus, it becomes very important to allow NAT traversal, for which there are many different solutions.

One of the simplest NAT traversal solution is static port mapping. One configures the router to forward all traffic to ports 80 and 443 to the web server. This allows having at most one such server behind a NAT having one IPv4 address. If multiple servers are required, the other server needs to use a non-standard port such as 8080 and 8443.

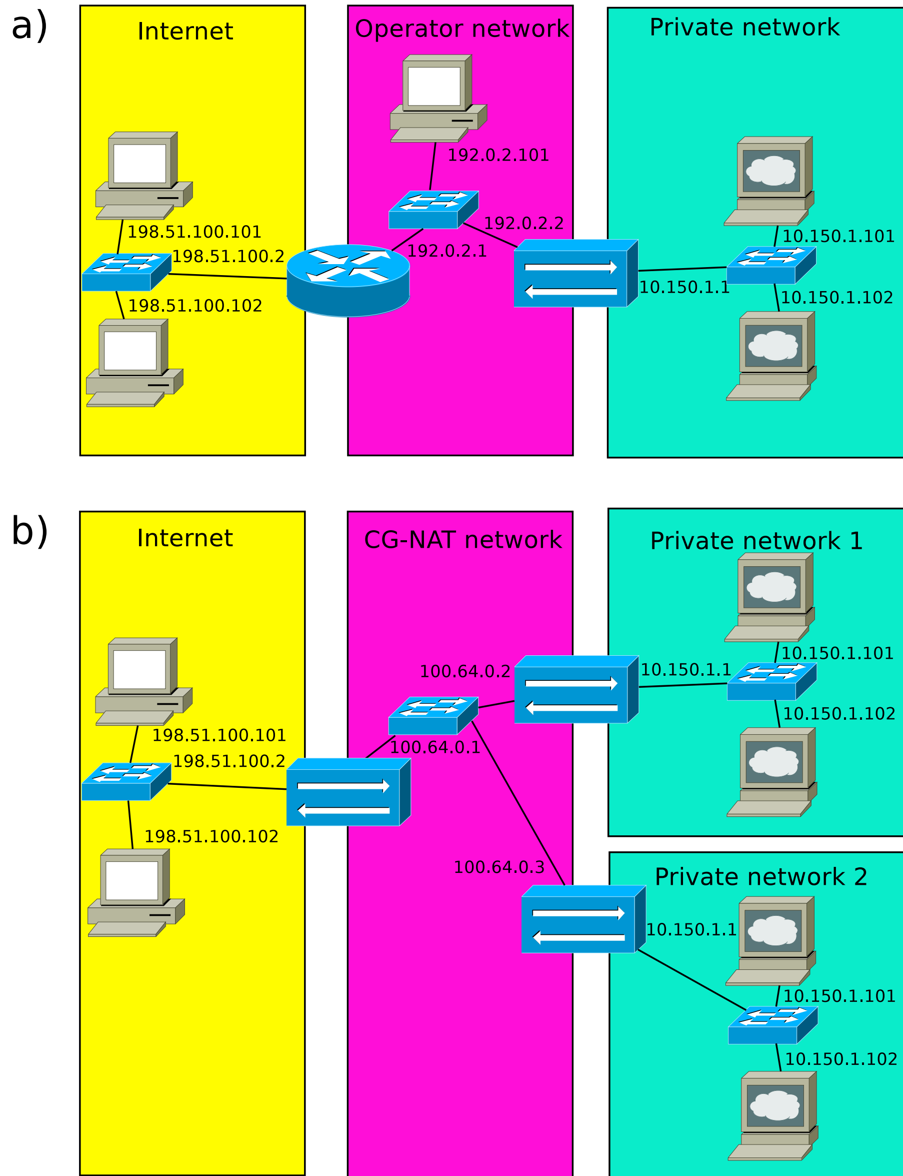


Figure 2.1. Deployment of NAT in (a) a non-nested configuration and (b) carrier grade NAT plus private NAT.

Version	IHL	DSCP	ECN	Total length
Identification			Flags	Fragment offset
Time to live		Protocol	Header checksum	
Source IP address				
Destination IP address				
Options (if IHL > 5)				
Source port			Destination port	
Sequence number				
Acknowledgement number				
Data off	Reserved	Flags	Window	
Checksum			Urgent pointer	
Options (if Data off > 5)				

Figure 2.2. IP and TCP headers and the fields a network address translator needs to touch. A network address translator primarily touches source and destination IP address and port fields. However, to make the header valid again, both checksums in the IP and TCP header need to be updated.

An enhancement to static port mapping is possible by using a protocol that is designed to control port mappings of a NAT middlebox. The first protocol to allow this was Internet Gateway Device (IGD) protocol of Universal Plug and Play (UPnP). This has not been standardized in any RFC, but is rather part of an ISO/IEC standard 29341. The UPnP+IGD is a very complex protocol, being a prime example of overengineering. Implementing it from the scratch is very hard, and picking an existing implementation done in an unsafe language is prone to risks, as the implementers have to incorporate a lot of complexity, thus opening doors for many potential attacks. Apple designed NAT Port Mapping Protocol (NAT-PMP) in 2005, and it has been published in a later RFC[12]. In 2013, IETF standardized Port Control Protocol (PCP)[64] based fortunately on NAT-PMP rather than UPnP+IGD.

By setting a SOCKS[32] proxy or a HTTP CONNECT proxy on the NAT middlebox, one can allow TCP connections through the NAT to the private address space. However, not all clients support such a proxy. It may be cumbersome to configure such a proxy. For example, a web browser has only global proxy configuration. The proxy cannot be configured separately for individual domains. Furthermore, the configuration needs to be manually entered, meaning one cannot in practice use this SOCKS / HTTP CONNECT proxy approach to host multiple web servers behind a NAT middlebox.

If NAT supports endpoing independent mappings and is tuned for connectivity rather than security, NAT hole punching is a viable NAT traversal mechanism. In NAT hole punching, there is a server the job of which is to determine the endpoint chosen by the NAT. If `peer2` wants to connect to

`peer1`, before that `peer1` connects to `portserver`. Then `portserver` tells `peer1` what the external port chosen by the NAT is. By opening a listening socket to the same internal port, `peer1` can accept connections to the external port. In particular, `peer2` can connect to `peer1` now that the mapping is set and ports are known. The job of `portserver` was only to be a globally addressable server that can tell the port mapping chosen by the NAT.

An application layer gateway (ALG) is also a possibility for NAT traversal. However, an ALG requires the TCP connection to be terminated at both sides to the NAT middlebox, meaning it requires lots of memory resources. Furthermore, an ALG can modify the protocol level traffic in some cases. An ALG tuned for connectivity rather than for protocol validation can avoid modifying the protocol level traffic, only using its protocol knowledge for determining the host name of the server the client wants to connect to.

A novel NAT traversal mechanism is realm gateway (RGW)[34]. In RGW, standard DNS[37, 38] queries are used for setting up a temporary mapping for the NAT middlebox. The mapping is expired in 2 seconds or after a connection arrives, whichever happens sooner. By having e.g. 3 public IPv4 addresses in the NAT middlebox, many simultaneous incoming connections can be accepted. RGW is analyzed in more detail in Section 2.5.

2.2 TCP window

TCP is a protocol that sends ordered reliable byte streams over an unreliable transport (IP). Because the underlying transport has a finite maximum packet size, a large chunk of data sent needs to be broken to multiple small packets. Also, there are two factors that can affect how much data can be sent at once: flow control and congestion control. Flow control ensures that a fast sender does not overflow a slow receiver, whereas congestion control ensures that a fast sender does not overflow a slow network.

The TCP window field is a 16-bit unsigned integer that can hold values between 0 and 65535, telling how many bytes of new data the sender is allowed to send without overflowing the receiver. The value 0 means that the receiver buffer is full and the receiver cannot receive more data. When the receiver has read some of the data, it sends a window update packet with a nonzero window. As this packet can be lost, a sender seeing zero window is supposed to periodically send zero window probes but some TCP stacks send keepalive packets instead that have the same function (see

Section 2.4). In response to a zero window probe or a keepalive packet, the receiver sends a new ACK window update packet with the new window value.

The TCP window field as originally defined supports windows only up to 65535 bytes. However, long links such as intercontinental links can alone due to speed of light have a latency of 0.2 seconds. Therefore, such a long link could achieve only throughput of approximately 328 kilobytes per second or approximately 2.6 megabits per second. Even the cheapest consumer Internet connections today are faster than that. This problem of limited window limiting throughput was realized and RFC1323[30] defines a window scale option that bit-shifts the window. Such a scaled window has a resolution coarser than 1 byte, but can represent larger values. For example, with window scale option 5, the coarse resolution is $2^5 = 32$ bytes so everything below and including 31 bytes has to be treated as zero, but maximal window is $2^5 \cdot 65535 = 2097120$ bytes allowing throughput of 84 megabits per second for an intercontinental link.

2.3 SYN cookies

SYN cookies are a mechanism to avoid allocating state for a half-open connection. A TCP SYN+ACK packet contains two 32-bit numbers that the other party has to echo back. Firstly, and most importantly, it has a 32-bit initial sequence number. Secondly, if timestamp option is supported, it has a 32-bit initial timestamp. These 32–64 bits of data can be used to store connection settings and cryptographical information that prevents random ACK packets from opening connections.

When the ACK packet arrives in response to the sent SYN+ACK, the connection is opened solely based on the information in the ACK packet. Firstly, the stored connection settings are extracted from the sequence number and the timestamp. Secondly, the cryptographical information is verified so that a randomly sent ACK packet does not open a connection.

Originally, SYN cookies were proposed by D.J. Bernstein. The original proposal included a 5-bit timestamp incrementing every 64 seconds that is an extremely poor way of protecting against replay attacks (because it wraps very soon). The original proposal also included only encoded maximum segment size information but not encoded window scaling information. Thus, when TCP connections were opened according to the original proposal, their performance suffered in high bandwidth-delay

product links, as noted by the author in the original proposal: “The biggest effect of the SYN flood is to disable large windows”[8]. The original proposal used MD5 as the hash function.

Fortunately, SYN cookies can be updated to be relevant in the modern times. MD5 is an insecure hash function[62], so SHA1[18] could be considered instead. However, SHA1 is even slower than MD5 which is already quite slow. Thus, in this thesis it was decided to follow FreeBSD’s example and use SipHash[4] as the hash function. Simpler hash functions such as MurmurHash having a secret seed are vulnerable to seed-independent multicollisions[5]. The poor 5-bit timestamp in the cookies can be replaced by two revolving secrets. The secrets can revolve e.g. every 32 seconds: every 32 seconds, the current secret index is flipped to become the old secret, which is at the same time regenerated using a cryptographically secure pseudorandom number generator. When checking a SYN cookie, both the current and the old secret are checked, giving cookies a validity time of 32–64 seconds. The window scaling information of the client can be encoded to the SYN cookie with coarse resolution. It is not an error to always treat the window as smaller than it actually is according to the client. In fact, the original SYN cookie implementation just ignores window scaling data, making a larger error than what is made here. These design details are somewhat similar to the FreeBSD SYN cookie implementation that has been explained in detail in a mail to the tcpm mailing list of IETF[1].

2.4 SYN proxy

SYN proxy is a technique used with or without SYN cookies that uses a proxy middlebox to ensure the connecting client genuinely wants to open a connection before handing off the opened connection to the true server. The functioning of SYN proxy is illustrated in Fig. 2.3. First, the client / remote host sends a SYN. The proxy responds with SYN+ACK but with zero window. The client responds with ACK. After the proxy has seen the ACK, it opens the other half of the connection: it sends a SYN to the protected server host, which responds with SYN+ACK containing the window of the server. The proxy then responds with ACK to the server and sends an ACK window update packet to the client. This ACK window update has a nonzero window, assuming the protected server host sent a

[1] https://www.ietf.org/mail-archive/web/tcpm/current/msg08071.html

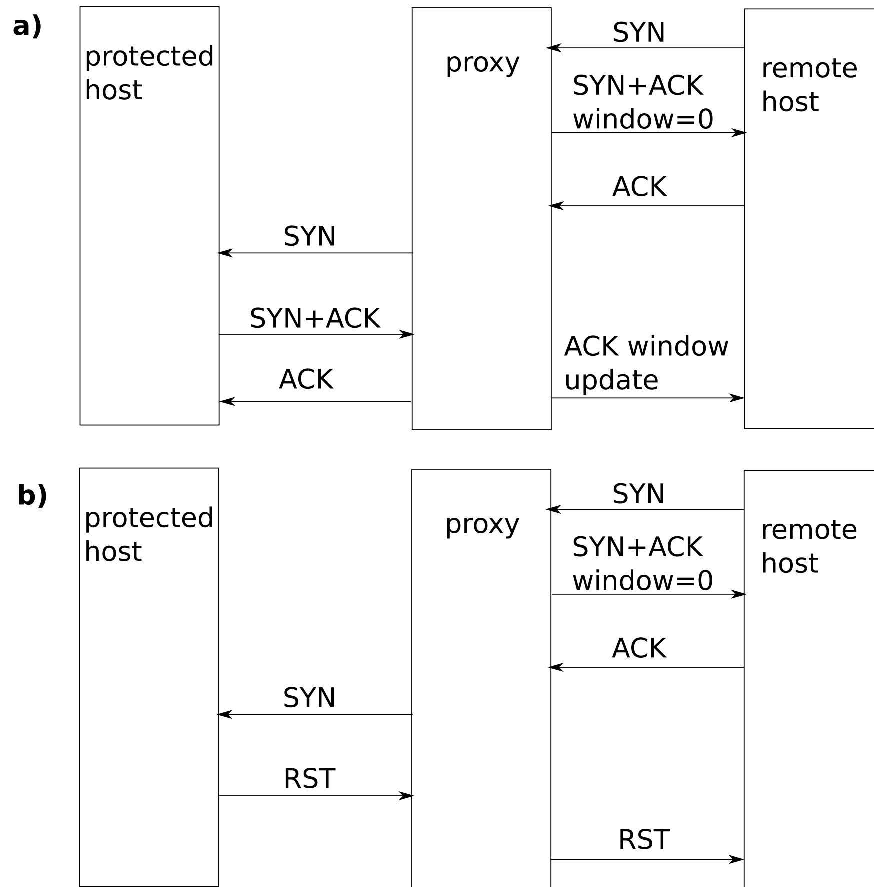


Figure 2.3. Establishment attempt of SYN proxied connection for (a) an open port and (b) a closed port.

nonzero window in the SYN+ACK.

SYN proxies can in theory be nested infinitely. When nesting them, however, latency adds up and the system sends many window update packets with zero window to the client before sending the final window update with nonzero window. In theory, the window update packets with zero window could be omitted for efficiency.

There are several things that need careful attention when developing a SYN proxy. For example, the port on the protected server host can be closed. The proxy will happily open the first half of any connection. However, when the first half of the connection is fully open, the protected server host can respond with RST to the SYN packet. An RST in response to a SYN is actually an RST+ACK packet, very different from an RST in response to any other packet. The SYN proxy needs to translate the RST+ACK response to SYN as an RST packet belonging to an existing already opened connection. Some SYN proxy implementations do not do this. For example, the Linux netfilter SYN proxy lacks this translation of the RST packet, and therefore, if one types `nc -v -v -v synproxyserver.example.com 12345` where

12345 is a closed port, one sees the connection remaing open forever in a stalled state. In contrast, a proper SYN proxy such as the one developed in this thesis will show for netcat that the connection was opened and then immediately closed.

Also, a SYN proxy needs to choose several important values in the TCP header. For example, the TCP initial sequence number of the server cannot be magically known by the SYN proxy so the SYN proxy needs to incorrectly guess some TCP initial sequence number and then forever for the lifetime of the connection translate the TCP sequence number (and do any TCP checksum translations required by the sequence number translation). Furthermore, the SYN proxy needs to guess a TCP window scaling value. The server may or may not choose the same TCP window scaling value guessed by the SYN proxy, so the SYN proxy needs to forever translate the window field value by bit-shifting for the lifetime of the connection. This translation requires the checksum translation, too.

The SYN proxy also needs to cope with packet loss. For example, if the transmission of SYN to the protected server host has not resulted in a seen SYN+ACK response, it means either the SYN or SYN+ACK was dropped. A timed retransmit of SYN ensures the SYN+ACK should be eventually seen. The ACK window update to the client also can be lost, meaning the connection may become stuck in a zero window state. To prevent this, most TCP stacks send zero window probes. However, some stacks send keepalive packets instead of zero window probes[2]. In comparison to zero window probe, a keepalive packet has the sequence number decremented by one. A proper SYN proxy functions with either type of TCP stack.

There are some already reported implementations of SYN proxy. One noteworthy implementation is the OpenBSD's packet filter (PF) that uses a SYN cache instead of SYN cookies. The Linux netfilter implementation uses SYN cookies and fails to properly translate RST response to SYN into an RST within an existing connection. These implementations have not been discussed in the literature. The literature contains at least one example of SYN proxy[14], but its source code is apparently unavailable.

2.5 Realm gateway

In this section, Realm gateway (RGW)[34] is introduced. It is a firewall that offers a novel NAT traversal mechanism that is based on standard

[2] https://www.spinics.net/lists/netfilter/msg57946.html

DNS[37, 38] queries. First, the firewall of RGW with its NAT traversal support is introduced, and then an application layer gateway extension to RGW that supports NAT traversal without needing address allocation is discussed. Finally, some shortcomings of RGW are underlined.

2.5.1 RGW firewall

The realm gateway (RGW) firewall is a component written in Python, using the Linux netfilter system for connection forwarding. A custom embedded DNS server is written in Python and included in RGW. The RGW is assigned a small number of IPv4 addresses, e.g. three addresses, in what is called a circular pool.

The operation occurs as follows: whenever a client requests IP address from the DNS server, RGW allocates state of the connection by dynamically setting up temporary port forwarding for the next 2 seconds. The IP address of this port forwarding is given to the client in the DNS reply with TTL=0. When the client connects to the IP address, the temporary port forwarding is automatically disabled, so the client can connect only once.

An immediate problem occurs if the client can connect multiple times to the same IP address. This problem occurs e.g. for HTTP[22] and HTTPS, and therefore, RGW requires an application layer gateway for these protocols.

If multiple users connect at the same time, the RGW can supply them with different IPv4 addresses up to the limit of the number of IPv4 addresses given to RGW. When all IPv4 addresses are already in use for new incoming connections, the RGW simply does not respond to the UDP DNS query. Fortunately, a DNS client tries multiple times to resolve the IP address if the first attempt was not successful.

It needs to be underlined that IPv4 address is reserved only between the period of sending the DNS response and seeing the TCP SYN packet. For an hour-long SSH session, the IPv4 address is not reserved for the entire hour but rather for the time between OpenSSH resolving the IP address and connecting to the IPv4 address.

2.5.2 RGW attacks

In this subsection, various attacks against RGW are hypothesized to see how RGW could be vulnerable. All of the attacks are DoS attacks for the simple reason that RGW has been designed to be secure, and in gen-

eral against secure systems the only possible attacks are variants of DoS attacks.

2.5.2.1 DNS flood

Each DNS query causes the RGW to allocate state for the next 2 seconds. Thus, by flooding the DNS server at a low packet rate, an attacker can cause the RGW to allocate all IPv4 addresses continuously for the attacker so that legitimate users cannot connect.

2.5.2.2 SYN flood

By flooding the RGW with a stream of TCP SYN packets for the served ports[20], the attacker could successfully steal allocated connection state of other users. Thus, the other users cannot connect and service is effectively denied for them.

2.5.2.3 Reflector use

By sending lots of TCP SYN packets to the RGW, with spoofed source IPv4 addresses belonging to the true target network, the attacker could use the RGW as a reflector[42] so that the RGW attacks the true target network with a high-rate sequence of SYN+ACK packets.

2.5.3 RGW attack mitigations

In this subsection, it is discussed how the various potential attacks against RGW can be mitigated.

2.5.3.1 DNS flood

Two strategies are used against DNS flooding attacks. The first is a reputation system. The second is a DNS truncated TCP challenge.

The reputation system assigns each client an initial reputation. If the client sends DNS queries that result in opening a TCP connection, its reputation rises. If the client does only DNS queries but does not connect via TCP after the DNS query, its reputation falls. When allocating state in situations of high traffic, high-reputation clients are preferred. There is a working implementation of this mechanism published in Github[33].

Furthermore, DNS truncated TCP challenge may be used. This sets the truncated bit in the DNS UDP response, indicating the client must connect via TCP. The original purpose of the truncated bit was for responses that are so large that they do not fit to a single UDP packet. Here, the truncated bit is (mis)used to cause the client to connect via TCP as a challenge.

2.5.3.2 SYN flood

The protection against SYN flood[20] is a SYN proxy sitting in front of the RGW. For details of how a SYN proxy works, see Section 2.4. The SYN proxy ensures the RGW sees the connection attempt only after the SYN proxy has caused the client already to demonstrate its true willingness to open the connection.

2.5.3.3 Reflector use

Against reflector use, a system to limit responses to incoming TCP SYN packets is installed. The system allows only a certain amount of SYN+ACK replies to SYN packets to be sent each second to a single /24 network. The network size is obviously configurable, but /24 seems like a good choice.

In the custom SYN proxy implemented in this thesis, a more sophisticated hash limiting is employed based on a token bucket algorithm where a token is added back to the bucket whenever a connection attempt is successful. Thus, somebody who repeatedly connects successfully, disconnects, connects successfully, disconnects, . . . will see all connection succeed without any kind of throttling.

2.5.4 Application layer gateway

Because RGW does not work with protocols such as HTTP[22] and HTTPS that may open multiple connections after single DNS address resolution, an application layer gateway is needed for RGW. Originally, the HTTPS gateway required installation of server certificates and private keys to the RGW. This is an intrusion of privacy. The firewall should not be able to decrypt traffic.

Later, in a Master's thesis[53], a custom application layer gateway was implemented where the main benefit is that server certificates and private keys are not needed in the RGW. It works by sniffing the server name indication from the TLS ClientHello message.

This implementation has been done in 2 processes per connection approach where each process handles one direction of the connection. Some alternative approaches could be: (a) 1 process per connection handling both directions of the connection in the same process using non-blocking I/O multiplexing, (b) handle both directions of all connections in single process in a non-blocking manner by using I/O multiplexing, (c) enhance the single-process approach to have a small number of processes or threads balancing the load between many CPU cores.

Because these days, in Master's programs the main programming language taught is Python, the ALG was implemented in Python. A more optimal implementation could be possible using C. However, the performance of the ALG was reasonably good, because Python can read a large block of data in a single line of code and send the same large block of data in another single line of code. Thus, the interpreted nature of Python will not be a bottleneck. The hostname extraction in the Master's thesis used YaLe, a parser generator written in C and targeting C by the author of this Licentiate thesis.

The characteristics of this ALG are mainly determined by its 2 processes per connection approach. At high data rates and small connection counts, this approach is superior to other approaches because it allows distributing the load to many CPU cores. However, at large connection counts, this approach uses lots of memory.

2.5.5 Shortcomings of RGW

One obvious shortcoming is that to operate in a high-volume environment, RGW requires multiple IPv4 addresses for optimal operation. This may be a problem as IPv4 addresses are a scarce resource. For example, if one uses RGW in a home environment, but wishes to host high-volume servers at home, the 1 IP address that an operator gives per device may not be enough. In Finland, many operators offer at most 5 IP addresses, one per device, so at most five devices can be operated without NAT. The addresses are usually served via DHCP[17], so if one requires many IP addresses for a single computer, some unusual configuration is required to send one DHCP request with the real MAC address and then send several DHCP requests with spoofed MAC addresses.

RGW also does not work with protocols where the client resolves IP address of a host name once but expects to be able to connect to this IP address multiple times. One such protocol where this happens is HTTP[22], either as plaintext or operated on top of TLS[51].

Furthermore, the TTL=0 header value of DNS response sent by RGW may not be fully supported in all environments. For example, a client could cache the request and then immediately recycle it during the next garbage collection event, seeing it has exceeded its TTL. If the garbage collection events are not continuously occurring but occur e.g. once per second, this could mean the IP address entry is cached for at most one second. Thus, two repeated connections in a very quick succession could cause just one

DNS query.

Originally RGW required not only ≈3 IPv4 addresses for itself but also routing configuration and two IPv4 addresses for both interfaces of the SYN proxy that is necessary to protect the RGW from state stealing DoS attacks. The routing changes needed could mean running the RGW in a home environment is impossible. The author of this thesis later implemented a SYN proxy as a layer 2 inline element on top of netmap. This SYN proxy is fully compatible with RGW and frees up two valuable IPv4 addresses and eliminates the need to configure routing.

The application layer gateway (ALG), as implemented for RGW, is user space implementation where the TCP connection at both sides is terminated to the ALG. Such termination of the TCP connection may use valuable memory resources on the RGW middlebox. A sophisticated attacker could open lots of connections and cause memory to be quickly used up. However, this attack on application layer requires working two-way communications so it cannot be done with a spoofed address. Therefore, this attack would be traceable, although the originator of the attack could be a compromised computer offering no information about the person who compromised it.

3. Improved SYN cookies and nmsynproxy

In this chapter, SYN cookies and especially their implementation in a SYN proxy are improved compared to the state of the art. The resulting component is called nmsynproxy where nm refers to netmap[57, 58, 56], although currently it can also run without netmap. The software is available at `https://github.com/Aalto5G/nmsynproxy`.

3.1 Layer 2 SYN proxy

Both the Linux kernel and the OpenBSD implementations of SYN proxy operate as a layer 3 network element. This type of SYN proxy requires routing changes if used as a standalone component. However, there is nothing that could prevent the implementation of a SYN proxy operating on layer 2, like there is nothing preventing the implementation of layer 2 firewalls.

SYN proxy operating as a layer 2 network element gives more freedom for efficient use of IPv4 addresses and requires less time configuring manual routing or routing protocols. A layer 2 network element does not preclude layer 3 operation: a layer 2 SYN proxy in front of a layer 3 router looks to the external network like a layer 3 router, and a layer 2 SYN proxy alone looks to the external network like an Ethernet bridge.

Layer 2 operation of course makes the SYN proxy implementation dependent on the network encapsulation. For example, there could be networks alternative to Ethernet and there could be alternative encapsulations used in Ethernet networks. In particular, the encapsulations for Ethernet can include:

1. DIX Ethernet / Ethernet II (the most common)

2. Raw IEEE 802.3 (length field instead of EtherType)

3. IEEE 802.2 LLC

4. IEEE 802.2 LLC + SNAP,

but however, in practice, only Ethernet II is used for IP networks.

Also, the alternatives to Ethernet can include at least IEEE 802.11 WLAN. However, in practice, the network drivers translate the headers so that the operating system sees them as Ethernet headers. Furthermore, virtual network interfaces such as veth in Linux and the virtual machine network access mechanisms also mimic Ethernet.

Thus, it is feasible to implement a layer 2 SYN proxy supporting Ethernet with Ethernet II as the only encapsulation. All other important networks mimic Ethernet, and Ethernet II is the only used encapsulation in practice. The SYN proxy of this thesis is for the abovementioned reasons a layer 2 SYN proxy supporting Ethernet II frames.

The layer 2 SYN proxy only specially processes TCP packets. Packets other than IP or IP packets other than TCP are silently passed through. Therefore, for example ARP[44] works. However, to allow operation, the network interfaces need to be placed to promiscuous mode.

3.2 Header checksums

Calculating the IP header checksums can take significant amounts of CPU time especially if packets are large. However, there is a possibility to update the checksum headers reflecting modifications to other header fields in a manner that the header checksum will be valid if and only if it originally was valid. Thus, for ordinary data segments, the entire header checksum needs no validation. The SYN proxy only validates it for "important" segments such as those opening or closing a connection, and for other segments the fast update mechanism is used.

The checksum calculation formula is:

$$C' = f(\sum_k V_k) \tag{3.1}$$

$$C = \sim C', \tag{3.2}$$

where V_k is a 16-bit word of the packet and the function $f(x)$ loops as long

as there are high-order 16 bits, and the high-order 16 bits are added to the low-order 16 bits. The operator $\sim$ is the bitwise NOT operator. The arithmetic is 32-bit two's complement integer arithmetic. The checksum calculation is detailed in RFC1071[10].

The fast update works as follows:

$$C_1' = \sim C_1 \quad (3.3)$$

$$C_2' = f(C_1' + \sim F_1 + F_2) \quad (3.4)$$

$$C_2 = \sim C_2' \quad (3.5)$$

where the function $f(x)$ is defined in the way explained previously, C_1 is the old checksum, C_2 is the new checksum, F_1 is the old 16-bit header field value (aligned on a 16-bit boundary) and the F_2 is the new 16-bit header field value.

An example: if C_1 = 0x752F (hexadecimal number), then C_1' = 0x8AD0. If an aligned 32-bit field is changed from 10.1.2.3 (two words 0x0A01 and 0x0203) to 192.0.2.2 (two words 0xC000 and 0x0202), then we calculate:

$$C_{2a}' = f(0x8AD0 + \sim 0x0A01 + 0xC000) \quad (3.6)$$

$$C_{2a}' = f(0x8AD0 + 0xF5FE + 0xC000) \quad (3.7)$$

$$C_{2a}' = f(0x240CE) \quad (3.8)$$

$$C_{2a}' = 0x40D0 \quad (3.9)$$

$$C_{2b}' = f(0x40D0 + \sim 0x0203 + 0x0202) \quad (3.10)$$

$$C_{2b}' = f(0x40D0 + 0xFDFC + 0x0202) \quad (3.11)$$

$$C_{2b}' = f(0x140CE) \quad (3.12)$$

$$C_{2b}' = 0x40CF \quad (3.13)$$

$$C_2 = \sim C_{2b}' \quad (3.14)$$

$$C_2 = 0xBF30 \quad (3.15)$$

The same formula works for any device updating TCP and also IP checksums. However, in practice only network address translators and firewalls update an already calculated TCP checksum. However, IP checksums need to be updated by routers that decrement the time-to-live header field.

The idea of the fast update is very similar to that of [35, 55] although the exact details of the equations can differ in their implementation but not in their results.

For unaligned 16-bit values, the two aligned 16-bit values that contain the unaligned value are enterred into the formula. 32-bit values and 64-bit values are broken to 16-bit chunks. 8-bit values are treated as 16-bit values with the other 8 bits set from the header. Thus, every 8-bit, 16-bit, 32-bit and 64-bit value, either aligned or unaligned, can be handled.

3.3 Hybrid SYN cookies

Some SYN cookies embed TCP settings and cryptographic information to TCP initial sequence number. This has the advantage that the 32-bit long initial sequence number must always be echoed by the client. Therefore, by merely using the initial sequence number, the SYN cookies do not rely on options that may not be supported by the TCP stack of the remote client. However, the initial sequence number has only 32 bits of space for settings and cryptography.

Other SYN cookies may embed the information to the TCP timestamp. This kind of approach has the drawback that not all TCP stacks support TCP timestamps. Thus, the 32-bit timestamp may not be properly echoed, and therefore, important information is lost.

In this thesis, hybrid SYN cookies are used. The word “hybrid” means making use of at least two options or features. In this case, both the TCP timestamp and the TCP initial sequence number are used for cryptographic data and TCP settings.

The scheme works as follows: there are two lists for window scaling and two lists for maximum segment size. The main list for window scaling can be for example (0, 2, 4, 7) that encodes to 2 bits and the additional list can be for example (0, 1, 3, 5, 6, 8, 9, 10) that encodes to 3 bits. Then, if the remote client has a window scaling value of 5, the main value is 4 that is encoded to two bits and the additional value is 5 that is encoded to three bits. When the SYN+ACK response is constructed, the main value 4 is encoded to the TCP initial sequence number and the additional value 5 is encoded to the TCP timestamp.

If the remote client supports TCP timestamps, the maximum value of the two possible TCP window scaling values is used. In this case, the main value was 4 and the additional value was 5. Thus, window scaling value of 5 would be correctly selected.

If on the other hand the remote client does not support TCP timestamps, the only information there is about the window scaling is the main value

4. Thus, the TCP connection works but not in an optimal manner as the SYN proxy thinks window scaling is 4 but in reality window scaling value 5 could be used for even better performance.

The present SYN cookie implementation also contains an additional measure for proper validation of initial sequence numbers. The sequence number of the remote endpoint is included to the hashed data. Thus, it is possible to verify that the remote endpoint did not change its initial sequence number. It is also possible to distinguish between zero window probes and keepalive packets belonging to a connection where packet loss occurred and thus the client saw a zero window that did not go away. This distinguishing is important because some TCP stacks send zero window probes and others send keepalive packets[1].

Also, this inclusion of the other endpoint initial sequence number allows quick port reuse. Quick port reuse means an old connection is closed and remains in the TIME_WAIT state, and then a new connection is opened with both endpoints using the same port number. The other endpoint may increase its initial sequence number value slowly, so the new increased initial sequence number can be a valid value for the old connection. Therefore, the ACK packet for which the SYN cookie is verified matches the existing state, and thus, it is treated as belonging to the existing connection in the TIME_WAIT state. By encoding the initial sequence number to the SYN cookie, even slightly changed other endpoint initial sequence number should result in major changes to the local SYN cookie that is used as the local initial sequence number.

3.4 Hybrid SYN proxy

Some SYN proxies always rely on SYN cookies. An example of such as SYN proxy is the Linux kernel netfilter SYN proxy module. However, SYN cookies have limited resolution for TCP settings, so for example window scaling value or maximum segment size can be chosen suboptimally. Thus, for best possible performance, SYN cookies should not be used.

Some other SYN proxies always rely on SYN cache. The SYN cache is a table of small data structures optimized for opening new connections. Only the information that needs to be stored in this stage is stored. When a connection is fully opened, the small SYN cache entry is promoted to a large main connection table entry. In practice, this promotion involves

[1] `https://www.spinics.net/lists/netfilter/msg57946.html`

freeing the small data block and allocating a new large data block. A benefit is that memory is saved when under a DoS attack. A drawback is that each connection requires two allocations: a small one initially and a large one later, thus increasing overheads. An example of such a SYN proxy is the OpenBSD packet filter (PF) SYN proxy feature.

In the SYN proxy of this thesis, hybrid use of SYN cache and SYN cookies is implemented. The SYN cache has a maximum size. When a new connection attempt arrives, it is put to SYN cache but its initial sequence number at the same time is a SYN cookie. If the client fully confirms its intention to open the connection, SYN cache is first consulted and if the data is not available in the SYN cache, SYN cookies are used as the fallback mechanism. If the SYN cache overflows, a new connection attempt overwrites the oldest attempt still in the cache.

As a matter of fact, it is very rare in the present implementation to rely on SYN cookies. A connection setup time is usually about 0.2 seconds if it is ever going to succeed. This setup time is mainly governed by speed of light latency. At maximum 40Gbps wire speed using 64 byte packets, i.e. at 59.92 million packets per second, 11.984 million SYN packets are seen between SYN and ACK. So, less than 12 million states need to be maintained in the SYN cache. At 300 bytes per entry (slightly overestimated), about 3.35 gigabytes of memory is required for the SYN cache. A server having 40Gbps network interface card surely has more memory than that.

3.5 Fragment handling strategy

The SYN proxy cleverly avoids handling fragmented IP packets. For IPv4, the design choice was made that the first fragment needs to be at least 60 bytes long (60 bytes being the maximum TCP header size). In theory, IPv4 specification does not explicitly forbid segments smaller than this, and IPv4 specification only requires minimum MTU of 68 bytes (maximal header size being 60 bytes and minimal payload being 8 bytes). In practice, it is rare to see such small MTUs. For IPv6, the first fragment needs to have the entire header chain, a requirement set by RFC7112[24].

An endpoint host reassembles IPv4 fragments based on source IP address, destination IP address, protocol number and packet identifier. Thus, an attacker cannot spoof the protocol number field. If a single fragment claims the protocol to be UDP, a subsequent fragment cannot overwrite it to be TCP.

All fragments except the first fragments are quickly passed through the SYN proxy. Non-TCP first fragments are also quickly passed through. For IPv6, first fragments not having the entire header chain[24] and fragment offset values between 1–511 (inclusive) are forbidden, as are packets where the end of a maximum sized TCP header would be at least 512 bytes within the fragmentable part.

These rules ensure that the first fragment always has complete information about the TCP header, so that subsequent fragments cannot overwrite it.

Note that if some or all of the subsequent fragments are passed through quickly, and the first fragment contains something that causes it to be dropped, the endpoint hosts never see the full packet.

This fragment handling strategy is not perfect. For example, if a FIN segment contains data and is fragmented at the same time, the length of the data cannot be known (nowhere in the IP or TCP header is the full length of the packet specified; the IP length is the length of the fragment, not the length of the whole packet). Thus, the end offset can be incorrectly calculated. However, most TCP implementations use path MTU discovery, which means they send only packets with the don't fragment (DF) bit on. Therefore, having a FIN segment that is both fragmented and contains data in addition to the FIN is extremely rare. Also, the same problem can occur for ordinary data segments, if they are fragmented. If a RST packet is sent after such an ordinary data segment, the SYN proxy may not accept it as RST sequence number verification is more strict than normal packet sequence number verification because RST packets do not contain a secondary acknowledgement number, only the primary sequence number, thus having less numbers to verify.

3.6 State machine

Fig. 3.1 shows the state machine of the SYN proxy. In this figure, DL refers to downlink (from Internet to protected network) and UL refers to uplink (vice versa). S refers to SYN, SA refers to SYN+ACK, F refers to FIN, A refers to ACK and FA refers to FIN+ACK acknowledging a previous FIN packet. Note that FIN segments always have the ACK bit set, but F refers to a FIN that does not acknowledge a previous FIN packet whereas FA refers to a FIN that acknowledges a previous FIN packet.

The state machine in the figure is missing one state, RESETED. It has

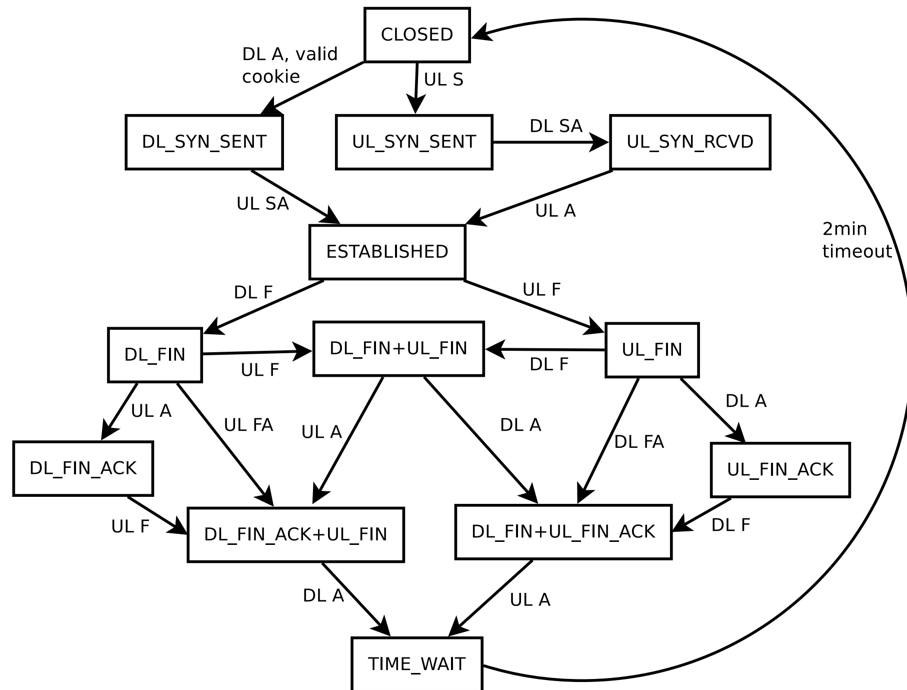


Figure 3.1. State machine for life cycle of SYN proxied connections.

a timeout of 45 seconds and upon reception of a valid RST packet this state is selected. The validity of RST packets is verified such that its sequence number must be the expected sequence number plus or minus at most three. This makes the RST handing secure against blind in-window attacks[49].

This state machine is not the same as in RFC793[48], because the RFC793 state machine is an endpoint state machine and this state machine is a middle-point state machine. This state machine was carefully constructed based on visualizing how a middle-point state machine between two RFC793 state machines should operate. This state machine is original work: for example, the OpenBSD's packet filter (PF) state machine has the state split into two, src.state and dst.state. In this state machine, some of the states may be simultaneously set which are denoted with the plus sign, so this state machine has some similarity to the PF state machine where the state is split into two. Also, Linux netfilter system has a state machine which contains the states NONE, SYN_SENT, SYN_RECV, ESTABLISHED, FIN_WAIT, CLOSE_WAIT, LAST_ACK, TIME_WAIT and CLOSE, so it is not at all similar to this state machine.

In addition to the state machine, there is strict sequence number handling for TCP packets. Sequence numbers are allowed between these two values:

- last sent sequence number minus maximum seen window,
- what other side has acknowledged plus current window.

These limits were not invented by the author of this thesis. Instead, credit is given to Guido van Rooij[61].

These limits should be understood in the circular sense. For example, 3221225472 is smaller than 1 in the circular sense (mod 2^{32}) because the circular path backwards from 1 to 3221225472 is shorter than the circular path forwards from 1 to 3221225472.

3.7 Improved hash limiting

SYN cookies are not without their drawbacks. One rarely considered drawback is that SYN cookies allow line rate responses to SYN packets. Therefore, if an attacker uses a massive botnet as DDoS source to send a flood of SYN packets to a SYN proxy, claiming to be from IP address 192.0.2.20, the SYN proxy will happily respond to every one of the SYN packets. Thus, the SYN proxy will effectively flood the 192.0.2.20 host in the 192.0.2.0/24 network with a SYN+ACK flood. Therefore, the SYN proxy can be used as a reflector in a DDoS attack.

To prevent the reflector use of SYN proxy, a hash limiting strategy can be used. The Linux netfilter system has a hashlimit module which does this. However, it uses dynamic memory allocation and does not add a token back to the hash bucket whenever a connection is successful, so it will rate-limit connections even if every connection succeeds.

In this thesis, a custom hash limiter has been implemented. It is a hash table of e.g. 131072 buckets. Each bucket contains e.g. 2000 tokens initially and is replenished at a rate of e.g. 400 tokens each second up to the maximum of e.g. 2000 tokens. Whenever a SYN+ACK response to a SYN packet is sent, one token is taken from the hash bucket of the target /24 network (the network size is configurable; for IPv6 it is /64 and configurable too). The hash function used is SipHash[4]. Whenever an ACK packet successfully establishes a connection, one token is added back to the hash bucket, which is a crucial feature where this hash limiter differs from the Linux netfilter hash limiter.

The timers to add back tokens to the bucket are batched. Each timer updates a batch of e.g. 16384 adjacent buckets, which takes approximately

20 microseconds. The timer firing intervals are evenly distributed, so a second timer does not expire immediately after the first timer expires. As the buckets are adjacent, the cache behavior of the timer function is well-defined and easily predictable. For a hash table size of 131072 buckets, eight such timers are used. Having one global timer would be approximately 160 microseconds, which some might consider as too high packet processing delay.

Note that because there are 16777216 /24 networks but only 131072 token buckets in the hash table, some networks are aliased to other networks. Which networks are aliases are however impossible to be predicted by the attacker due to use of SipHash with a secret seed.

3.8 Installation instructions

Currently, nmsynproxy uses the pptk git submodule and stirmake as the build system. The build system stirmake in turn uses the abce git submodule. To install stirmake:

```
git clone https://github.com/Aalto5G/stirmake↵
cd stirmake↵
git submodule init↵
git submodule update↵
cd stirc↵
make↵
sh install.sh↵.
```

These instructions install stirmake to `~/.local` where the binaries should be executable on modern Linux distributions and where manual pages should be viewable. If `install.sh` complains about the directory `~/.local`, then you must create this directory with `mkdir` and re-run `install.sh`. At this point, it may also be necessary to re-log-in on the Linux machine (for graphical sessions just reopening a terminal may not be enough) so that the local binary directory will be in path.

To build nmsynproxy without netmap support:

```
git clone https://github.com/Aalto5G/nmsynproxy↵
cd nmsynproxy↵
git submodule init↵
git submodule update↵
smka↵.
```

To build it with netmap support, create the file `opts.smk` in the `nmsynproxy` directory with these lines:

```
@subfile↵
@strict↵
↵
$WITH_NETMAP = @true↵
$NETMAP_INCDIR = "/home/yourname/netmap/sys"↵,
```

where netmap from `https://github.com/luigirizzo/netmap` has been cloned to `/home/yourname/netmap`. Then re-run `smka`.

4. Application layer network address translation

In this chapter, AL-NAT is proposed as a NAT traversal mechanism for protocols mainly based on HTTP[22] and TLS[51], although it is shown that the recently standardized transport protocol called QUIC[29] is also capable to operate with AL-NAT. As a prototype of AL-NAT, a component ldpairwall has been released at `https://github.com/Aalto5G/ldpairwall`. The name is "airwall" as opposed to "firewall", because the component is a prototype of AL-NAT without any kind of security policy.

4.1 Technical overview

4.1.1 Technical details

Whenever AL-NAT middlebox receives a connection attempt, it first SYN proxies the connection using either SYN cookies, SYN cache or making hybrid use of both. When the client has verified its true intention to fully open the connection, the AL-NAT middlebox sends an ACK window update packet but does not open the other half of the connection. Thus, the connection is not yet open at the server but the client sees it fully open.

AL-NAT works only for protocols where the client begins the exchange of messages without needing any data from the server. Such protocols include HTTP and TLS at least.

Whenever the client sends data for the semi-open connection, the middlebox buffers and acknowledges the data. As an alternative, it may only buffer without acknowledging the data. The middlebox attempts to detect the protocol automatically and to find the host name of the server within the data. HTTP/1.1 requires users to send host name in the `Host:` header, and TLS supports server name indication (SNI)[19]. In practice, all web browers support SNI and some popular web sites require the client to send

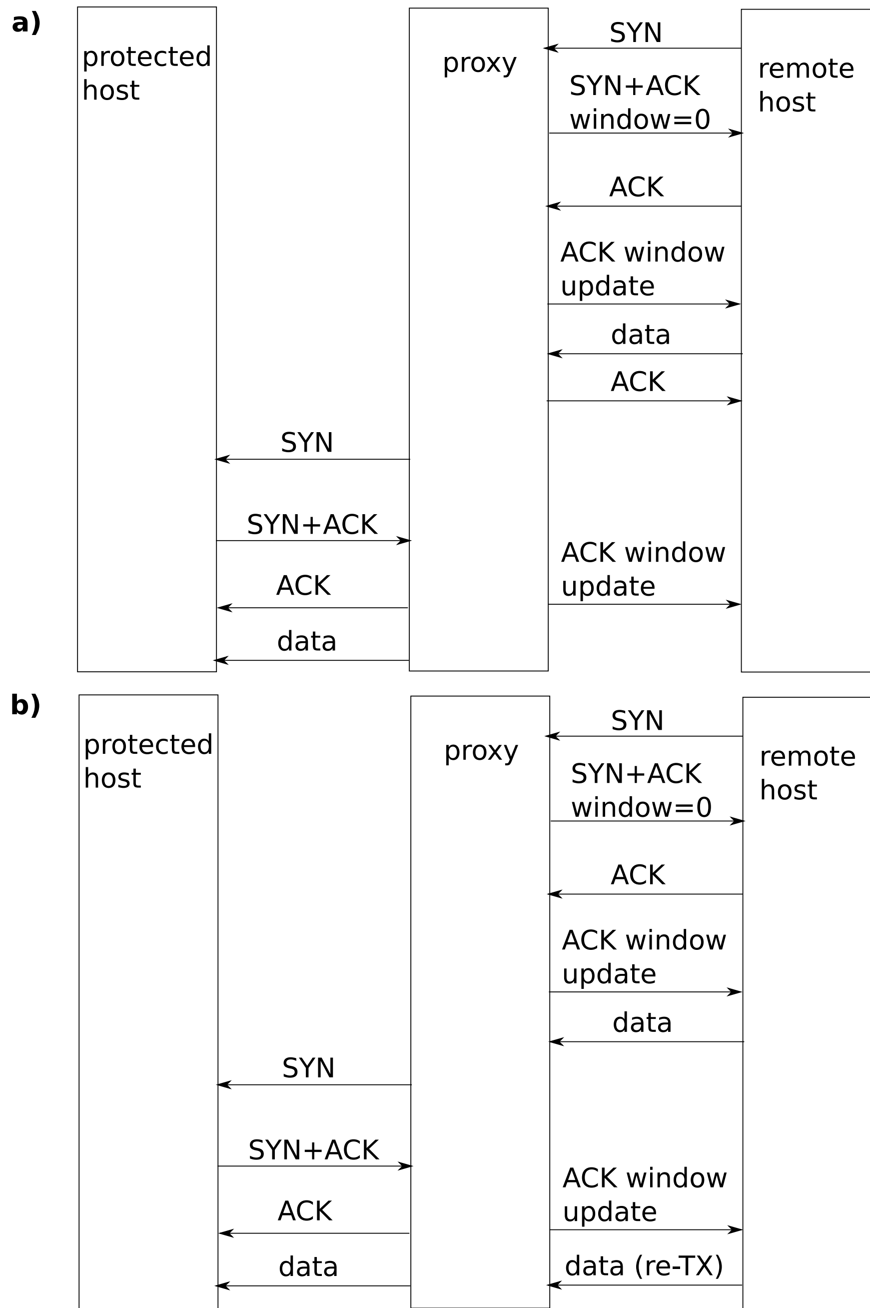


Figure 4.1. Establishment of AL-NAT connection in (a) the standard variant and (b) the alternative variant.

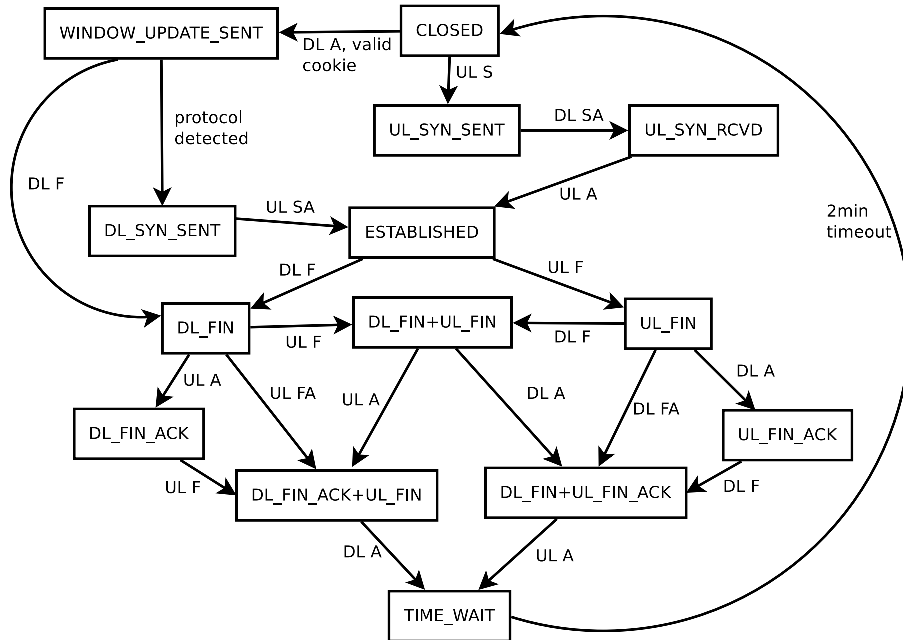


Figure 4.2. State machine for life cycle of AL-NAT connections.

the SNI; without an SNI, a TLS connection to any of these popular web sites is not possible.

When the middlebox has detected the host name of the server, it knows which private IP address to forward the connection to. In the standard variant, a SYN / SYN+ACK / ACK exchange is done and all acknowledged client data is sent with possible retransmissions if some packets are lost. Furthermore, the remote client is sent another ACK window update, this time with real data from the true server. In the alternative variant, the ACK window update frame is sent without acknowledging the initial client packets, causing the remote client to retransmit all of its initial data.

After the connection has been fully established at each end, the connection is handled like any SYN proxied connection is handled: NAT is performed, sequence and acknowledgement number modification is performed and window may be adjusted based on what window scaling difference the true server and the middlebox had.

4.1.2 State machine

The state machine of AL-NAT is heavily based on the state machine of SYN proxy, as described in Section 3.6. Most of the state machine is shown in Fig. 4.2 with the exception of special RESETED state which is caused by seeing a valid RST packet with valid sequence number. The reasoning for the RESETED state missing from the diagram is that it would be impossible to draw it in a non-overlapping manner, as every state has a

Lost	Response
SYN by client	client retransmits
SYN+ACK to client	client retransmits SYN
ACK response to SYN+ACK by client	client sends 0-window probe
First ACK window update to client	client sends 0-window probe
Data segment during detection	client timer
SYN to server by middlebox	client timer (see text)
SYN+ACK to middlebox	client timer (see text)
ACK response to SYN+ACK to server	server timer
Second ACK window update to client	non-fatal, merely an update

Table 4.1. Packet loss handling in AL-NAT.

transition to the RESETED state.

The timeout of each state is configurable and different. For example, the timeout in ESTABLISHED state is one day, the RESETED state has 45 second timeout and the TIME_WAIT has a timeout of 2 minutes. The state CLOSED is a special state that occupies no memory and lacks a timeout.

RFC7857[43] suggests a simpler rudimentary state machine that is valid only for middleboxes that do not track and verify TCP sequence numbers. In the present AL-NAT implementation, there is full sequence number handling, so the RFC7857 state machine is not applicable.

4.1.3 Packet loss and retransmissions

The packet loss handling of AL-NAT is summarized in Table 4.1. The first two types of packet loss (SYN by client, SYN+ACK to client) are handled by client retransmitting SYN due to a timer. In AL-NAT, the SYN+ACK response to ACK creates a zero window. If the ACK sent by the client as a response to SYN+ACK or the response to the ACK (the ACK window update packet) is lost, the client sends timed zero-window probes or keepalive packets depending on the TCP stack variant the client is using. The AL-NAT proxy detects these packets and can treat the window update or zero window probe as the packet that opens the SYN proxied connection and retransmits the first ACK window update. If any initial data packet sent by client is lost, the client does not see an ACK response and retransmits the data using a timer. Purposefully, the last inital data packet that caused the protocol and server hostname to be detected is not acknowledged before the other half of the connection is established. Thus, if SYN to server by middlebox or SYN+ACK to middlebox by server is lost, the client retries sending the data segment that caused protocol and hostname to be detected, which causes the AL-NAT middlebox to re-send

SYN to server. If ACK response to SYN+ACK to protected server is lost, the protected server retries sending SYN+ACK. Furthermore, the last packet of the initial exchange, the second ACK window update to client, is merely an update the loss of which is not fatal.

4.2 Protocol analysis

4.2.1 TCP

TCP[48] is the protocol for which AL-NAT was originally intended. It is not an easy task to implement AL-NAT for TCP, however. Firstly, one needs a functioning SYN proxy codebase. This requires one to solve all corner cases of SYN proxy, including guessing a suitable window scaling and handling the eventual case where the server uses a different window scale value than what the SYN proxy chose. One also needs to have a strategy for loss of various types of packets. The mere loss of one packet should never cause the connection establishment process to stall.

Additionally, at least the TCP stack in Linux has the property that if it has not received an ACK packet for data it sent a second ago, it goes to a retransmit code path that only retransmits the first TCP segment. Thus, if one types:

`GET / HTTP/1.1`↵ (immediately)
`Host: server.com`↵ (second press 1 s later)
↵ (third press 1 s later than second press),

the Linux kernel sends first the GET line, and when the second line is completely typed, chances are 1 second has already elapsed. Thus, the Linux kernel has already entered the retransmit code path where it retransmits only the first TCP segment, waiting for an ACK. If the SYN proxy never acknowledges this first segment, instead waiting for the segment containing the host name, the connection stalls.

Thus, at least for Linux, if one desires that a slow human user entering the HTTP request manually to netcat should work, the SYN proxy must ACK and buffer the initial data segments received from the client instead of waiting for the host name segment to arrive and letting the client retransmit them to the protected host which would acknowledge them.

4.2.2 QUIC

QUIC[29] is a next-generation transport layer protocol designed to eventually replace TCP. QUIC is layered on top of UDP[45]. Unlike TCP/IP stacks that operate in kernel space, the QUIC has been designed to run entirely in user space with only the UDP part working in kernel space. It incorporates several enhancements that are missing from TCP:

- The connection establishment packet is large, thus increasing the costs of connection establishment and making DoS attacks less feasible
- The possibility of DoS attacks has been taken into account right from the start, making it possible for endpoints to send a challenge before trusting the other party
- All connections are encrypted[60]
- Connection and encryption setup requires less round trips than in TCP because encryption is deeply embedded into the protocol
- It supports several multiplexed streams for a single connection
- Head-of-line blocking is not a problem: blocking for one stream does not delay other streams
- Connections can be smoothly migrated to new IP addresses for example for a mobile phone that has simultaneous 5G and WLAN connectivity.

Some features are similar to TCP:

- Each stream is an ordered reliable byte stream akin to a TCP connection, although streams may be unidirectional instead of being bidirectional
- Flow control ensures a fast sender will not overload a slow receiver
- Congestion control[28] similar to that of TCP[1] is being used to avoid overloading the network

The important feature of QUIC is that the initial handshake message is similar to a TLS ClientHello message in that it contains the server name indication. Thus, AL-NAT is possible for every protocol operating on top of QUIC assuming the server name indication in set properly and that the server name indication packet is not encrypted. There is some progress towards encrypting even the server name[52] (still in Internet Draft stage so the QUIC RFC does not have this), so if this encryption is widely adopted, the encryption keys for the server name (that may differ from the encryption keys of the actual data) need to be shared with the AL-NAT middlebox. Since QUIC was a rapidly moving target when ldpairwall was implemented, QUIC support has not yet been implemented in the AL-NAT reference implementation, ldpairwall.

4.2.3 Application-layer protocols

An application-layer protocol must be defined such that the client sends the host name of the server before the server needs to send any data for AL-NAT to work. Not all application-layer protocols have this property.

4.2.3.1 HTTP

The first versions of HTTP, HTTP/0.9 and HTTP/1.0[7] did not support multihosting properly. In these versions, the client only connected to the server's IP address, and specified the path. It was assumed that the server's host name was implicitly known due to knowing which server IP address the client connected to.

However, when HTTP was being deployed, IPv4 address exhaustion was already forecasted to happen in the near future. Thus, HTTP/1.1[22] was defined, which makes the `Host:` header mandatory. The client has to specify the host name of the server always. By making the `Host:` header mandatory, multihosting, i.e. running many sites on the same IP address, became possible.

All major web browsers today support at least HTTP/1.1[22] if not HTTP/2[6]. However, HTTP/2, although defined also for plaintext connections, is in practice only supported in its encrypted form running over TLS. Therefore, an AL-NAT middlebox needs not support HTTP/2 separately. It needs to support only HTTP/1.1, with HTTP/2 support being handled by TLS support.

The newest version of HTTP, HTTP/3[9], is defined to work on top of QUIC[29]. As a protocol running on top of QUIC[29], it is also AL-NAT

compatible, although upcoming encrypted server name can provide challenges.

4.2.3.2 TLS

TLS[51] is a cryptography protocol running on top of TCP, that is in turn used for various different protocols such as mail protocols and HTTPS. A noteworthy exception is SSH[65, 66, 67, 68] which is a cryptographically protected protocol having its own cryptography layer.

TLS as a cryptography protocol sounds like it should not allow AL-NAT operation. After all, if TLS encrypts everything, the AL-NAT middlebox cannot operate properly by detecting the host name of the server. However, in TLS, one can run many sites having different certificates on the same server. Thus, the client must send the ClientHello message containing the unencrypted host name before expecting the server to send its ServerHello message with the certificate. Because the client must send the ClientHello before expecting the server to send any data, TLS has the property that AL-NAT works properly.

However, one caveat is that the server name indication (SNI) field was not initially defined for first versions of TLS. It is a later addition[19]. For an application to support SNI, the application has to tell the host name of the server to the TLS stack. Typically, before SNI support, applications were written as follows:

1. Resolve host name into IP address

2. Connect to the IP address

3. Create a TLS context

4. Give the connected socket to the TLS stack for the TLS context

The only location where this type of code handles the host name of the server is the name resolution. Afterwards, the code does not provide the host name to the TLS stack because the TLS stack did not originally need to know it. However, for SNI to work, the code needs to be structured like this:

1. Resolve host name into IP address

2. Connect to the IP address

3. Create a TLS context

4. Give the host name of the server to the TLS stack for the TLS context

5. Give the connected socket to the TLS stack for the TLS context

The additional step is what makes SNI work.

SNI is supported by all major web browsers. There are web sites such as the site of the large Finnish newspaper Helsingin Sanomat that do not work for browsers not having SNI support. The reason Helsingin Sanomat can do this is that all major web browsers support SNI.

4.2.3.3 SSH

SSH[65, 66, 67, 68] is a protocol where the client and the server send the version greeting simultaneously. Thus, by slightly altering the idea of the protocol, one could wait in a middlebox for the client to send its version greeting before choosing which server to hand off the connection to. Unfortunately, the version greeting has no indication of the host name the client wants to connect.

Fortunately, SSH is a protocol that is almost always implemented by using the OpenSSH implementation which is extremely configurable. For example, the OpenSSH implementation supports “jump hosts”. One can have a host running SSH server that is used for jumping to hosts behind the server. In particular, the OpenSSH implementation also supports “proxy command”. This proxy command allows the use of any binary implementing a proxy protocol. Thus, by being extensible, OpenSSH supports arbitrary proxy protocols, those already defined and also those not defined yet.

By using the SSH “proxy command” support on the client side, it is shown in Section 4.3 how the proxy can be implemented in the AL-NAT middlebox in such a lightweight manner that running proxied protocol is no more expensive than running directly supported protocols. This enables SSH to work correctly with AL-NAT.

4.2.3.4 Mail protocols

There are three important mail protocols that could benefit from AL-NAT when the server is in private address space behind NAT (if only the client

is in private address space, then AL-NAT works just like regular NAT, and every TCP connection works). The protocols are SMTP[31], IMAP[13] and POP3[39].

SMTP starts by the server responding to a connection opening event by a version greeting. Because the server is the first to communicate, and because there is no command for the client to send the host name, SMTP is not AL-NAT compatible. However, SMTP has the NOOP command which has an argument. So, the client could be modified to send `NOOP fqdn:smtp.example.com` before seeing the server's version greeting. This would be a minor modification to the protocol that would allow it to be AL-NAT compatible.

IMAP is similar to SMTP in that the server is the first party to communicate. However, IMAP does not have an argument for the NOOP command, only a tag, and some implementations may limit the tag length. Thus, it is not as easy to modify IMAP to support AL-NAT than it is to modify SMTP to support AL-NAT.

POP3 also has the server start communications with a version greeting, similar to SMTP and IMAP. However, unlike SMTP and IMAP, POP3 does not even have a NOOP command. Thus, it seems unlikely that POP3 could be enhanced to support AL-NAT.

If SMTP, IMAP or POP3 is operated on top of TLS, they may use the SNI of TLS. However, actually using the SNI requires that the TLS clients are programmed in a manner that tells the hostname to the TLS stack. This may not be the case for all client implementations.

4.3 Carrier grade TCP proxy for unsupported TCP protocols

In this section, we first discuss how carrier grade TCP proxy as a NAT traversal mechanism can be efficiently implemented in the server-side middlebox (this part is already in operation in ldpairwall, the reference implementation). Then details of implementing it in the client-side middlebox are discussed (this part has not yet been implemented in ldpairwall). Then a new architecture for Internet is proposed that works by doing switching between cooperative firewalls supporting AL-NAT and carrier grade TCP proxy. Finally, a client library is presented for applications such as OpenSSH and netcat that do not support carrier grade TCP proxy.

4.3.1 Carrier grade TCP proxy in server-side middlebox

AL-NAT has been designed to not modify the TCP stream. The initial request sent by the client is forwarded to the server after AL-NAT figures out the private address of the server based on its hostname. However, AL-NAT can be changed to perform certain modification operations in the TCP stream. In this case, the client sees a different TCP byte stream than the server.

Why would one want to modify the TCP stream? The reason is proxy protocols. For example, HTTP CONNECT proxy protocol begins by the client sending the following message to the proxy:

```
CONNECT www.example.com:80 HTTP/1.1↵
Host: www.example.com:80↵
↵,
```

and the proxy responding by the following response:

```
HTTP/1.1 200 OK↵
↵.
```

Then after the proxy negotiation has happened between the client and the proxy, normal protocol-dependent traffic continues. This protocol-dependent traffic may be encrypted. In the current implementation of HTTP CONNECT proxy protocol, the initial proxy traffic is not encrypted (although it could be encrypted with small modifications to the codebase).

It is not hard in AL-NAT to parse the HTTP `CONNECT` message and at the same time remove it from the TCP stream. It is not also hard to inject the `HTTP/1.1 200 OK` to the stream towards the client. These TCP stream modifications allow the AL-NAT middlebox to function as a carrier grade TCP proxy. The connecting client can specify any destination for the connection, and the AL-NAT middlebox then hands off the SYN proxied connection to the correct private host.

An alternative to HTTP CONNECT proxy protocol would be the SOCKS protocol that has several versions: SOCKS4, SOCKS4a and SOCKS5[32]. However, as HTTP support is anyway needed to support the most popular application-layer protocol in the Internet, using HTTP CONNECT protocol for the proxy seems like a natural fit. Furthermore, SOCKS5 has the problem of requiring quite many round trip times for connection establishment. In fact, there was SOCKS6 protocol being standardized to reduce the round trip time count for connection establishment[41], but the

work was stopped and the drafts have expired.

The client can for example decide to send the CONNECT message based on domain name system query that tells it has to do an additional CONNECT hop. Such domain name system modifications are explained starting from Section 4.3.2. In the case of SSH connections, it is also possible to configure the details to use a proxy on the client side. Since SSH is most secure when used with public and private keypair that already require some configuration, it is not too much effort to configure details for using proxy.

Nearly the same could obviously be achieved by having a real proxy running in the AL-NAT middlebox. However, this real proxy would terminate its TCP connections at both sides to the middlebox. It would not be thus a *carrier grade* proxy, because operating in a carrier environment could exhaust the kernel memory available in a powerful computer.

Carrier grade TCP proxy (CG-TP) differs from a real proxy program in that it has the same minimal memory consumption that AL-NAT has. The connection state fits in less than a kilobyte as will be shown in Section 5.3. In the initial state of a connection, more memory may be used temporarily for protocol and hostname detection. The carrier grade TCP proxy only needs to modify IP addresses, time to live header field, TCP ports, sequence and acknowledgement numbers, window field value and checksums. The actual data never needs any modification after the connection has been fully established.

4.3.2 Carrier grade TCP client in client-side middlebox

A client for carrier grade TCP proxy can be implemented in the client-side firewall middlebox. The middlebox intercepts `A?` DNS queries e.g. for `ssh.example.com` and translates them into two parts: unmodified `A?` query for `ssh.example.com` and modified `TXT?` query for `_cgtp.ssh.example.com`. Both of these translated queries are sent at the same time in a pipelined manner. The middlebox waits for both queries to return a result, positive or negative. If the `TXT?` query returned a positive result, e.g. `192.0.2.2!ssh.example.com`, the client-side middlebox reserves an IP address from a special private IP address space reserved for carrier grade TCP proxy client purposes. The middlebox then responds to the original intercepted `A?` DNS query by e.g. a response `A=10.240.5.6, TTL=1`. Note the `TTL=1` which makes it less likely for the client to cache the temporary address for long periods of time. It is known some DNS implementations do not properly support `TTL=0`, so there-

fore `TTL=1` is used instead of `TTL=0`. The middlebox also stores a bidirectional mapping `10.240.5.6=192.0.2.2!ssh.example.com` ⇔ `ssh.example.com`.

When the client computer connects to 10.240.5.6, the middlebox looks up the `10.240.5.6=192.0.2.2!ssh.example.com` ⇔ `ssh.example.com` entry and performs a destination NAT to 192.0.2.2 and a source NAT for the public IP address of the middlebox. It then inserts `CONNECT ssh.example.com:22 HTTP/1.1` method call into the TCP stream whenever the TCP connection is opened.

For example, the reserved IP address space could be 10.240.0.0/12 which contains million usable IP addresses. The client-side middlebox maintains bidirectional mappings between IP address and host names in a least recently used (LRU) cache. Whenever an IP address sees a TCP connection attempt or a DNS `A?` query requests an already known host name bound to its IP address, the IP address is moved to become the most recently used. Whenever all IP addresses in the reserved space are in use, and a modified `TXT?` query results in a response stating the requested server supports carrier grade TCP proxy, the least recently used IP address is dropped according to the LRU cache eviction policy, to be replaced by the newly discovered address that is added to the list as the most recently used address.

Here the `192.0.2.2!ssh.example.com` is a *bang path*. It means the carrier grade TCP proxy client should connect to `192.0.2.2` and request `CONNECT ssh.example.com:22 HTTP/1.1` where 22 is the port number. For nested firewalls, the bang path can contain more entries. For example, `192.0.2.2!bounce.example.com!ssh.example.com` means the client should first request `CONNECT bounce.example.com:22 HTTP/1.1` and only then `CONNECT ssh.example.com:22 HTTP/1.1`. The first bang path entry should always be an IP address.

It is perfectly fine for a client to support less than million addresses in order to limit memory consumption to much below 128 megabytes. For example, if the typical host name and bang path is at most 84 bytes (longer host names and bang paths can be stored as separately allocated memory blocks), the IP address is 4 bytes, a doubly linked list node is 8 bytes, and a red-black tree node is 32 bytes, a single entry takes 128 bytes. Thus, million entries take 128 megabytes. Most middlebox implementations are expected to have more memory than that, but if a low-end middlebox is extremely memory-limited, it can support e.g. only 65536 addresses that would at most use 8 megabytes of memory.

4.3.3 New Internet architecture of cooperative firewalls

By embedding a recursive DNS resolver server in the client-side middlebox, one can implement a new architecture for Internet that is based on switching between cooperative firewalls. No client application and no server application requires any changes for this kind of architecture. All real servers can be run behind NAT.

The client-side middlebox is a cooperative firewall that intercepts every TCP stream to the special reserved IP address space by inserting HTTP CONNECT method call as the first data packet:

```
CONNECT www.serverexample.com:80 HTTP/1.1↵
Host: www.serverexample.com:80↵
↵,
```

and the server-side middlebox then intercepts every TCP connection, thus detecting the CONNECT method call, and strips the CONNECT method call away so that the real server software sees only the protocol traffic.

This system allows every TCP-based protocol to work seamlessly, no matter whether the protocol works with regular AL-NAT or not. QUIC[29] based protocols would be handled directly by QUIC support, because the first QUIC packet sent by the client already has the server name indication.

The TCP-based switching approach would for example solve all MTU issues because every part of the connection would use TCP as its sole protocol. Any kind of approach based on tunneling would end up with MTU issues. The Internet is full of broken firewalls that do not properly translate or even drop ICMP[46] packet too big messages. Furthermore, tunneling would in practice need to run on top of UDP because TCP over TCP is a terrible idea due to two layers of retransmissions that can cause a practical collapse of connection throughput if there is heavy packet loss.

About the only drawback of this approach compared to tunneling is that tunneling can add encryption. This kind of proxying does not support encryption, so in practice encryption needs to be supported by the endpoint applications. However, typically many HTTP servers and web browsers use encryption.

4.3.4 Carrier grade TCP client directly in client computer

By modifying endpoint applications, it is possible to add support to CG-TP. There are three approaches in which applications may be modified:

1. Change the code of the application to use the new interfaces and recompile

2. Create a preloadable library that replaces some socket and name resolution functions

3. Offer a proxy command, a separate binary speaking a proxy protocol (works only for programs that support proxy commands such as OpenSSH)

The new programming interface for approach (1) is a function that creates a connected socket and returns it. It replaces (a) name resolution function, (b) socket creation function `socket()` and (c) socket connection function `connect()`. The interface is simply `int socket_ex(char *, uint16_t)`. Using the improved interface may be beneficial for applications because it not only adds CG-TP support but also adds transparent IPv6 support.

The approach (2) or the preloadable library uses 8192 revolving addresses in the forbidden network 0.0.0.0/8 in the range 0.0.0.0 – 0.0.31.255 as answers to DNS queries. It replaces the `gethostbyname()`, `getaddrinfo()`, `freeaddrinfo()` and `connect()` functions. The replaced functions store the bang path into a 8192-entry table and picks a revolving address from the range 0.0.0.0 – 0.0.31.255 which denotes the index to the table. Then, if the program wants to `connect()` to an IPv4 address within the range 0.0.0.0 – 0.0.31.255, the bang path is obtained from the 8192-entry table and CG-TP connection establishment is performed instead. Currently, the preloadable library is not thread-safe, so it is intended mainly for simple single-threaded applications like netcat.

The proxy command of approach (3) is a very simple two-file-descriptor non-blocking I/O multiplexer. It first creates the CG-TP socket, and then starts to copy data between standard input and the socket write side, and between the socket read side and standard output. It has been tested with OpenSSH and found to work perfectly.

All three approaches can be obtained from the git repository at `https://github.com/Aalto5G/cghcpcli`.

4.4 Installation instructions

Currently, ldpairwall uses the pptk git submodule and stirmake as the build system. The build system stirmake in turn uses the abce git submodule. To install stirmake:

```
git clone https://github.com/Aalto5G/stirmake↵
cd stirmake↵
git submodule init↵
git submodule update↵
cd stirc↵
make↵
sh install.sh↵.
```

These instructions install stirmake to `~/.local` where the binaries should be executable on modern Linux distributions and where manual pages should be viewable. If `install.sh` complains about the directory `~/.local`, then you must create this directory with `mkdir` and re-run `install.sh`. At this point, it may also be necessary to re-log-in on the Linux machine (for graphical sessions just reopening a terminal may not be enough) so that the local binary directory will be in path.

To build ldpairwall without netmap support:

```
git clone https://github.com/Aalto5G/ldpairwall↵
cd ldpairwall↵
git submodule init↵
git submodule update↵
smka↵.
```

To build it with netmap support, create the file `opts.smk` in the `ldpairwall` directory with these lines:

```
@subfile↵
@strict↵
↵
$WITH_NETMAP = @true↵
$NETMAP_INCDIR = "/home/yourname/netmap/sys"↵,
```

where netmap from `https://github.com/luigirizzo/netmap` has been cloned to `/home/yourname/netmap`. Then re-run `smka`.

5. Theoretical analysis

In this chapter, several properties of the solutions are theoretically analyzed. For the SYN proxy, its strength of cryptography and its memory usage characteristics are only theoretically analyzed. There are no IETF requirements in RFCs for SYN proxies. In fact, IETF in most cases strongly opposes any kinds of middleboxes and this probably includes SYN proxies too. So, it is unlikely that IETF will ever put forth comprehensive requirements for SYN proxies. In contrast, NAT is a widely deployed solution, and although the preference of IETF would be for NAT to not exist and IPv6 to be deployed everywhere, they have had to realize the facts and define behavioral requirements for NAT. In this chapter, it is analyzed which of those behavioral requirements are met in ldpairwall, in addition to the memory usage of ldpairwall.

5.1 nmsynproxy cryptography strength

There is a default nmsynproxy configuration that uses 2 bits for window scale option, 2 bits for maximum segment size, 1 bit for SACK supported bit and 1 bit for revolving secret index, thus leaving 26 bits out of 32 for cryptographical protection. Therefore, one in 67108864 random ACK segments not belonging to an existing connection can cause opening of a new connection.

The user can configure the SYN proxy differently, however. There is a check that at least 19 bits of security must be enabled. 19 bits of security means one in 524288 random ACK segments not belonging to an existing connection can open a new connection.

One could ask whether the revolving secret index is necessary. It is, due to the following reasoning: without the secret index, the cryptographical protection would have 27 bits. Thus, one in 134217728 cookies actually

passes the incorrect secret check as well. If secrets are revolved once per 32 seconds, and average connection setup time is 0.2 seconds, one in 160 cookies is first checked against the incorrect secret. Thus, one in 134217728·160 or one in 21474836480 cookies sets up the connection with incorrect TCP options. At a rate of 1000 new connections per second (plausible for a highly loaded server), a server experiences one faulty connection per approximately 249 days. However, by encoding the secret index, the strength of cryptographic protection is not reduced, so it is clear that it should be encoded too. One might protest that with the secret index there is only 26 bits for cryptographic protection instead of 27, but then only one secret is used for checking, which gives an extra bit of security.

This level of security can be improved by using timestamp option in addition to the initial sequence number. The default timestamp option has 3 bits of maximum segment size information, 3 bits of window scale information, 5 bits of timestamp information and 1 bit of current secret index, leaving the rest for cryptographical protection. These settings leave 20 bits for cryptographical protection, meaning one in 1048576 segments passes this additional cryptographical check.

5.2 nmsynproxy memory usage

The SYN proxy only specially handles TCP connections. UDP and ICMP packets are silently passed through. IP fragments have a clever handling explained in Section 3.5 that requires no allocation of memory. Thus, the only way TCP SYN proxy can be attacked in a memory use attack is to simply create lots of TCP connections.

The size of SYN proxy TCP connection block is 280 bytes. Therefore, a gigabyte of memory can support over 3.8 million connections. Note that by SYN flooding, one cannot exhaust this memory because the SYN cache of the SYN proxy is limited in size, so only a small fraction of this memory can be consumed by the SYN cache. When the SYN cache is overflown, connections resort to using SYN cookies, thus meaning a connection state entry is created only after the remote client has truly demonstrated its willingness to fully open a connection.

5.3 ldpairwall memory usage

The per-connection memory usage of ldpairwall depends on the protocol and on the status of a connection. The following data structures are of interest:

- TCP connection block
- UDP connection block
- ICMP connection block
- Protocol detection context

The TCP connection block size is 304 bytes. Thus, a gigabyte of memory can support over 3.5 million TCP connections. It is therefore unlikely that ldpairwall will ever run out of memory.

UDP and ICMP connections require lighterweight tracking, so their size is only 144 bytes. Therefore, for example a far larger number of QUIC connections can be supported than TCP connections. Also, QUIC can have a number of streams within a single connection, so the need to establish multiple simultaneous connections to a QUIC server is nonexistent.

The most plausible way to exhaust ldpairwall memory is to create lots of connections that are stuck in the protocol and hostname detection stage. A connection in this stage requires an additional 5016 bytes of memory. This stage has 240 second timeout, but the timer is reset every time a segment belonging to the connection is seen. Thus, a skilled attacker could retain lots of connections in the protocol and hostname detection stage. This could be improved by never resetting the timer in the protocol and hostname detection stage so that the 240 second timeout is absolute.

Furthermore, by sending lots of IP fragments, the IP fragmentation code paths of ldpairwall could in theory be attacked. However, there is a global configurable reassembly memory limit of 32 megabytes (default), so a memory exhaustion DoS attack is not very efficient. Therefore, the main way the IP fragmentation code paths are vulnerable is their algorithmic complexity (the FragmentSmack attack successfully exploits this[40]).

5.4 ldpairwall requirements analysis

There are several requirements for NAT in various RFCs. In this section, we analyse whether they are met in ldpairwall.

5.4.1 RFC4787

RFC4787[3] contains NAT behavioral requirements for unicast UDP[45].

5.4.1.1 REQ 1

"A NAT MUST have an "Endpoint-Independent Mapping" behavior."

This requirement is fully met, provided that there are unused ports available. The NAT in ldpairwall has been designed with a port manager that supports endpoint-independent mapping behavior.

5.4.1.2 REQ 2

"It is RECOMMENDED that a NAT have an "IP address pooling" behavior of "Paired". Note that this requirement is not applicable to NATs that do not support IP address pooling."

This requirement is not applicable, as ldpairwall uses only one external IP address.

5.4.1.3 REQ 3

"A NAT MUST NOT have a "Port assignment" behavior of "Port overloading".

a) If the host's source port was in the range 0-1023, it is RECOMMENDED the NAT's source port be in the same range. If the host's source port was in the range 1024-65535, it is RECOMMENDED that the NAT's source port be in that range."

This requirement is met as long as there are unused ports available. Once all ports of the NAT are already in use, only then it starts to do port overloading. The additional recommendation is not met, as the NAT implementation considers low reserved ports unavailable for general purpose use.

5.4.1.4 REQ 4

"It is RECOMMENDED that a NAT have a "Port parity preservation" behavior of "Yes"."

This recommendation is not met, except in cases where port preservation is possible due to the same port being available externally. Even ports

may be mapped to odd ports and odd ports may be mapped to even ports if the same port is not available externally. This recommendation would not be terribly hard to support, yet it is only a recommendation and not requirement.

5.4.1.5 REQ 5

"A NAT UDP mapping timer MUST NOT expire in less than two minutes, unless REQ-5a applies.

a) For specific destination ports in the well-known port range (ports 0-1023), a NAT MAY have shorter UDP mapping timers that are specific to the IANA-registered application running over that specific destination port.

b) The value of the NAT UDP mapping timer MAY be configurable.

c) A default value of five minutes or more for the NAT UDP mapping timer is RECOMMENDED."

The recommended 5 minute timer is used, so this requirement is met.

5.4.1.6 REQ 6

"The NAT mapping Refresh Direction MUST have a "NAT Outbound refresh behavior" of "True".

a) The NAT mapping Refresh Direction MAY have a "NAT Inbound refresh behavior" of "True"."

The requirement is met, as both outbound and inbound refresh behavior is True. Traffic in either direction keeps the UDP mapping alive.

5.4.1.7 REQ 7

"A NAT device whose external IP interface can be configured dynamically MUST either (1) automatically ensure that its internal network uses IP addresses that do not conflict with its external network, or (2) be able to translate and forward traffic between all internal nodes and all external nodes whose IP addresses numerically conflict with the internal network."

This is not applicable, as the external IP interface is configured statically.

5.4.1.8 REQ 8

"If application transparency is most important, it is RECOMMENDED that a NAT have an "Endpoint-Independent Filtering" behavior. If a more stringent filtering behavior is most important, it is RECOMMENDED that a NAT have an "Address-Dependent Filtering" behavior.

a) The filtering behavior MAY be an option configurable by the administrator of the NAT."

This requirement is met, as the filtering behavior is endpoint-independent. A port is opened when first outgoing packets are sent, and later any arrival of packets to this port creates automatically a connection state entry. However, a small caveat is that when the NAT does not have enough free ports available, it may not be possible to determine which private IP address and port should receive the incoming packets from unknown destination. Thus, the NAT gracefully falls back to address and port-dependent filtering if there are not enough free ports.

5.4.1.9 REQ 9

"A NAT MUST support "Hairpinning".

a) A NAT Hairpinning behavior MUST be "External source IP address and port". "

Hairpinning is supported, so packets can be sent to other clients behind the NAT using the external IP address and port of the other client. The external IP-port pair is used for this purpose.

5.4.1.10 REQ 10

"To eliminate interference with UNSAF NAT traversal mechanisms and allow integrity protection of UDP communications, NAT ALGs for UDP-based protocols SHOULD be turned off. Future standards track specifications that define ALGs can update this to recommend the defaults for the ALGs that they define.

a) If a NAT includes ALGs, it is RECOMMENDED that the NAT allow the NAT administrator to enable or disable each ALG separately."

The current ldpairwall implementation does not have any ALGs. However, it is worth mentioning that IETF seems to have a rather restricted view of what ALGs can be. Not all ALGs modify traffic, and thus, the justification to allow integrity protection of UDP communications is not a true justification here.

5.4.1.11 REQ 11

(All section references in this requirement refer to the RFC.)

"A NAT MUST have deterministic behavior, i.e., it MUST NOT change the NAT translation (Section 4) or the Filtering (Section 5) Behavior at any point in time, or under any particular conditions."

Here IETF has clearly taken a wrong decision in the requirement. It is a worse crime to disallow communications due to lack of resources than it is to allow communications with changed semantics. As long as

there are enough resources such as open ports, the behavior of ldpairwall is deterministic. However, if ldpairwall runs out of open ports, it may simply have to change its behavior because the only other option would be disallowing all communications, which could lead to a denial of service attack.

5.4.1.12 REQ 12

"Receipt of any sort of ICMP message MUST NOT terminate the NAT mapping.

a) The NAT's default configuration SHOULD NOT filter ICMP messages based on their source IP address.

b) It is RECOMMENDED that a NAT support ICMP Destination Unreachable messages."

This requirement is fully met. NAT mappings are only timed out, not terminated by ICMP messages. ICMP messages are not filtered, and destination unreachable is supported.

5.4.1.13 REQ 13

"If the packet received on an internal IP address has DF=1, the NAT MUST send back an ICMP message "Fragmentation needed and DF set" to the host, as described in [RFC0792].

a) If the packet has DF=0, the NAT MUST fragment the packet and SHOULD send the fragments in order."

This is not supported, mainly because ldpairwall is intended for use cases where the uplink and downlink interfaces are Ethernet and thus have the same MTU.

5.4.1.14 REQ 14

"A NAT MUST support receiving in-order and out-of-order fragments, so it MUST have "Received Fragment Out of Order" behavior.

a) A NAT's out-of-order fragment processing mechanism MUST be designed so that fragmentation-based DoS attacks do not compromise the NAT's ability to process in-order and unfragmented IP packets."

IP fragmentation is fully supported, including out-of-order fragments. The additional requirement (a) may not be fully met, but then again most important operating systems (Windows, Linux) were vulnerable to the IP fragmentation DoS attack called FragmentSmack[40] that the author found, too. The reason for (a) not being fully met is that FragmentSmack was discovered after ldpairwall was created.

5.4.2 RFC5382

RFC5382[26] contains NAT behavioral requirements for TCP.

5.4.2.1 REQ 1

"A NAT MUST have an "Endpoint-Independent Mapping" behavior for TCP."

This requirement is fully met, provided that there are unused ports available. The NAT in ldpairwall has been designed with a port manager that supports endpoint-independent mapping behavior.

5.4.2.2 REQ 2

"A NAT MUST support all valid sequences of TCP packets (defined in [RFC0793]) for connections initiated both internally as well as externally when the connection is permitted by the NAT.

a) In addition to handling the TCP 3-way handshake mode of connection initiation, A NAT MUST handle the TCP simultaneous- open mode of connection initiation."

The state machine prefers security over functionality in all cases, so some extraordinarily rare corner cases may not be handled properly. The simultaneous opening of TCP connections has not been tested. This RFC however was written before TCP split handshake was known, and the proper way to handle TCP split handshake can very well be to deny it.

5.4.2.3 REQ 3

"If application transparency is most important, it is RECOMMENDED that a NAT have an "Endpoint-Independent Filtering" behavior for TCP. If a more stringent filtering behavior is most important, it is RECOMMENDED that a NAT have an "Address-Dependent Filtering" behavior.

a) The filtering behavior MAY be an option configurable by the administrator of the NAT.

b) The filtering behavior for TCP MAY be independent of the filtering behavior for UDP."

This requirement is met, as the filtering behavior is endpoint-independent. A port is opened when first outgoing packets are sent, and later any arrival of SYN packet to this port creates automatically a connection state entry. However, a small caveat is that when the NAT does not have enough free ports available, it may not be possible to determine which private IP address and port should receive the incoming packets

from unknown destination. Thus, the NAT gracefully falls back to address and port-dependent filtering if there are not enough free ports.

5.4.2.4 REQ 4

"A NAT MUST NOT respond to an unsolicited inbound SYN packet for at least 6 seconds after the packet is received. If during this interval the NAT receives and translates an outbound SYN for the connection the NAT MUST silently drop the original unsolicited inbound SYN packet. Otherwise, the NAT SHOULD send an ICMP Port Unreachable error (Type 3, Code 3) for the original SYN, unless REQ-4a applies.

a) The NAT MUST silently drop the original SYN packet if sending a response violates the security policy of the NAT."

This is related to TCP simultaneous open, and thus the intention is to not support it.

5.4.2.5 REQ 5

"If a NAT cannot determine whether the endpoints of a TCP connection are active, it MAY abandon the session if it has been idle for some time. In such cases, the value of the "established connection idle-timeout" MUST NOT be less than 2 hours 4 minutes. The value of the "transitory connection idle-timeout" MUST NOT be less than 4 minutes.

a) The value of the NAT idle-timeouts MAY be configurable."

This is supported, and the timeout is exactly 1 day for established connections. and 2 hours 4 minutes if one side has closed the connection.

5.4.2.6 REQ 6

"If a NAT includes ALGs that affect TCP, it is RECOMMENDED that all of those ALGs (except for FTP [RFC0959]) be disabled by default."

There are no ALGs in ldpairwall. It is here worth mentioning too that IETF seems to have a rather restricted view of what ALGs can be. Not all ALGs modify traffic.

5.4.2.7 REQ 7

"A NAT MUST NOT have a "Port assignment" behavior of "Port overloading" for TCP."

This requirement is met as long as there are unused ports available. Once all ports of the NAT are already in use, only then it starts to do port overloading.

5.4.2.8 REQ 8

“A NAT MUST support “hairpinning” for TCP.

a) A NAT’s hairpinning behavior MUST be of type “External source IP address and port”.”

This requirement is fully met, as hairpinning is supported, and external source IP address and port are used for it.

5.4.2.9 REQ 9

“If a NAT translates TCP, it SHOULD translate ICMP Destination Unreachable (Type 3) messages.”

This is a very important requirement, and ICMP packets are indeed translated.

5.4.2.10 REQ 10

“Receipt of any sort of ICMP message MUST NOT terminate the NAT mapping or TCP connection for which the ICMP was generated.”

Only RST packets or FIN packets can terminate the NAT mapping, so this requirement is fully met.

5.4.3 RFC5508

RFC5508[59] contains NAT behavioral requirements for ICMP[46].

5.4.3.1 REQ 1

“Unless explicitly overridden by local policy, a NAT device MUST permit ICMP Queries and their associated responses, when the Query is initiated from a private host to the external hosts.

a) NAT mapping of ICMP Query Identifiers SHOULD be external-host independent.”

ICMP queries are indeed supported, and mappings are endpoint-independent.

5.4.3.2 REQ 2

“An ICMP Query session timer MUST NOT expire in less than 60 seconds.

a) It is RECOMMENDED that the ICMP Query session timer be made configurable.”

ICMP query session expiration timer is configurable with default being 60 seconds.

5.4.3.3 REQ 3

“When an ICMP Error packet is received, if the ICMP checksum fails to validate, the NAT SHOULD silently drop the ICMP Error packet.

a) If the IP checksum of the embedded packet fails to validate, the NAT SHOULD silently drop the Error packet; and

b) If the embedded packet includes IP options, the NAT device MUST traverse past the IP options to locate the start of the transport header for the embedded packet; and

c) The NAT device SHOULD NOT validate the transport checksum of the embedded packet within an ICMP Error message, even when it is possible to do so; and

d) If the ICMP Error payload contains ICMP extensions [ICMP-EXT], the NAT device MUST exclude the optional zero- padding and the ICMP extensions when evaluating transport checksum for the embedded packet.”

A different strategy is used instead. The checksum is translated in a manner that it is valid if and only if it was valid. Thus, invalid ICMP checksums remain invalid and the endpoint host drops the packet.

5.4.3.4 REQ 4

“If a NAT device receives an ICMP Error packet from an external realm, and the NAT device does not have an active mapping for the embedded payload, the NAT SHOULD silently drop the ICMP Error packet. If the NAT has active mapping for the embedded payload, then the NAT MUST do the following prior to forwarding the packet, unless explicitly overridden by local policy:

a) Revert the IP and transport headers of the embedded IP packet to their original form, using the matching mapping; and

b) Leave the ICMP Error type and code unchanged; and

c) Modify the destination IP address of the outer IP header to be the same as the source IP address of the embedded packet after translation.”

This modification is done based on the mapping.

5.4.3.5 REQ 5

“If a NAT device receives an ICMP Error packet from the private realm, and the NAT does not have an active mapping for the embedded payload, the NAT SHOULD silently drop the ICMP Error packet. If the NAT has active mapping for the embedded payload, then the NAT MUST do the following prior to forwarding the packet, unless explicitly overridden by local policy:

a) Revert the IP and transport headers of the embedded IP packet to their original form, using the matching mapping; and

b) Leave the ICMP Error type and code unchanged; and

c) If the NAT enforces Basic NAT function ([NAT-TRAD]), and the NAT has active mapping for the IP address that sent the ICMP Error, translate the source IP address of the ICMP Error packet with the public IP address in the mapping. In all other cases, translate the source IP address of the ICMP Error packet with its own public IP address."

This modification is done based on the mapping.

5.4.3.6 REQ 6

"While processing an ICMP Error packet pertaining to an ICMP Query or Query response message, a NAT device MUST NOT refresh or delete the NAT Session that pertains to the embedded payload within the ICMP Error packet."

No such deletion is done in the current implementation in ldpairwall.

5.4.3.7 REQ 7

"NAT devices enforcing Basic NAT [NAT-TRAD] MUST support the traversal of hairpinned ICMP Query sessions. All NAT devices (i.e., Basic NAT as well as NAPT devices) MUST support the traversal of hairpinned ICMP Error messages:

a) When forwarding a hairpinned ICMP Error message, the NAT device MUST translate the destination IP address of the outer IP header to be same as the source IP address of the embedded IP packet after the translation. "

Hairpinning should work for ICMP packets too, as hairpinning has been implemented in a protocol independent manner.

5.4.3.8 REQ 8

"When a NAT device is unable to establish a NAT Session for a new transport-layer (TCP, UDP, ICMP, etc.) flow due to resource constraints or administrative restrictions, the NAT device SHOULD send an ICMP destination unreachable message, with a code of 13 (Communication administratively prohibited) to the sender, and drop the original packet."

Extremely heavy care has been taken to make it implausible to exhaust the resources of a NAT. For example, the same port may be reused if free ports are not available. However, the communication administratively prohibited error message is not currently sent.

5.4.3.9 REQ 9

"A NAT device MAY implement a policy control that prevents ICMP messages being generated toward certain interface(s). Implementation of such a policy control overrides the MUSTs and SHOULDs in REQ-10."

Such policy control has not been implemented.

5.4.3.10 REQ 10

(All section references in this requirement refer to the RFC.)

"Unless overridden by REQ-9's policy, a NAT device needs to support ICMP messages as below, some conforming to Section 4.3 of [RFC1812] and some superseding the requirements of Section 4.3 of [RFC1812]:

a. MUST support:

1. Destination Unreachable Message, as described in Section 7.1 of this document.

2. Time Exceeded Message, as described in Section 7.2 of this document.

3. Echo Request/Reply Messages, as described in REQ-1.

b. MAY support:

1. Redirect Message, as described in Section 4.3.3.2 of [RFC1812].

2. Timestamp and Timestamp Reply Messages, as described in Section 4.3.3.8 of [RFC1812].

3. Source Route Options, as described in Section 7.3 of this document.

4. Address Mask Request/Reply Message, as described in Section 7.4 of this document.

5. Parameter Problem Message, as described in Section 7.5 of this document.

6. Router Advertisement and Solicitations, as described in Section 7.6 of this document.

c. SHOULD NOT support:

1. Source Quench Message, as described in Section 4.3.3.3 of [RFC1812].

2. Information Request/reply, as described in Section 4.3.3.7 of [RFC1812].

In addition, a NAT device is RECOMMENDED to conform to the following implementation considerations:

d. DS Field Usage, as described in Section 7.7 of this document.

e. When Not to Send ICMP Errors, as described in Section 4.3.2.7 of [RFC1812].

f. Rate Limiting, as described in Section 4.3.2.8 of [RFC1812]."

Various different kinds of ICMP messages are supported. In particular,

all of the MUST requirements are met.

5.4.3.11 REQ 11

(All section references in this requirement refer to the RFC.)

"A NAT MAY drop or appropriately handle Non-QueryError ICMP messages. The semantics of Non-QueryError ICMP messages is defined in Section 2."

This is not a requirement at all because according to it, a NAT MAY do anything. Thus, the non-requirement can be claimed to be met.

6. Testing

In this chapter, we discuss how the correctness of SYN cookie implementation in nmsynproxy was established. The performance of nmsynproxy is determined. The performance of protocol and hostname detection in ldpairwall is established. This protocol and hostname detection in ldpairwall along with the single-threaded nature of ldpairwall is the largest difference between ldpairwall and nmsynproxy performance related aspects. The required buffer size of AL-NAT is determined from analysis of real network traffic. Furthermore, it is shown that AL-NAT traversal works for both plain unmodified HTTP and TLS and when augmented with CG-TP also for SSH.

6.1 nmsynproxy correctness tests

The TCP SYN proxy nmsynproxy has comprehensive correctness tests that are automatically run by the build system. All of the tests are written in the C programming language, crafting the packets manually, and each test has an IPv4 and IPv6 version. The first test is a 3-way handshake + 4-way FIN test that tests setting up and closing a connection in the non SYN proxied direction. Then RST packets are tested for established connections in the uplink and downlink direction. Furthermore, a closed port test is performed in the non SYN proxied direction. Then it is tested that retransmissions of various packets belonging to a connection are correctly handled.

The remaining tests are tests of the SYN proxied direction. A normal SYN proxy handshake is tested. Then retransmissions of various packets belonging to the handshake are tested to be handled properly. This testing includes tests of both keepalive and zero window probe packets for a connection having zero window (some TCP stacks send keepalives even

though standards-compliant stacks should send zero window probes). After this, closed port handling in the SYN proxied direction is tested. This is a test that the Linux kernel SYN proxy in the netfilter system would not pass.

Then there are data transfer and RST tests for SYN proxied connections. Data transfer is tested both for uplink direction, downlink direction and both directions. RST packets are tested in both uplink and downlink directions.

The SYN proxy passes all of these correctness tests currently, and this has been the case for a long amount of time. The SYN proxy codebase is extremely stable and modifications are currently performed very rarely.

It is also worth mentioning that the SYN proxy was used for everything in Master's thesis of Maria Riaz[53]. It contained a very thorough analysis of the custom application layer gateway (ALG). Every single connection that went past the ALG went through the SYN proxy as well. The very careful testing of the ALG did not reveal a single bug in the SYN proxy, although one minor code improvement to SYN proxy was made (resending SYN packet in a timed manner) and one helpful documentation update was made to ensure Linux kernel does not drop packets that are supposed to go through the SYN proxy.

6.2 nmsynproxy performance tests

There are several performance tests that can be executed. One is the performance of worker-only packet processing. In this test, the packets to be processed are generated internally within the computer program, and they are not sent anywhere. The test measures pure packet processing overheads without including any packet I/O overheads. The performance of this test is illustrated in Table 6.1 on various different CPU models.

This worker-only test uses a non-ending sequence of two full-sized packets and one small minimum-sized packet. Thus, it is representative of large file transfers. Average packet size is approximately kilobyte, so even the slowest of the CPUs should be able to saturate a 40 Gbps network interface card using a single thread, if packet processing overhead is the major overhead.

However, it turns out that packet I/O is an extremely large overhead that cannot be ignored. This overhead is particularly large in virtualized environments. Thus, on the dual E5-2630L v2 machine, it took three

test	CPU	performance
worker-only	i5-8250U	5.29947 MPPS
worker-only	dual E5-2630L v2	5.99728 MPPS
worker-only	dual E5-2630 0	5.31238 MPPS

Table 6.1. Performance of worker-only packet processing on different machines.

protocol	time
SSL	0.21 μs
HTTP, host header as first	0.33 μs
HTTP, host header as last	1.18 μs

Table 6.2. Performance of protocol and hostname detection. The performance was measured on an Intel Core i5-8250U laptop.

threads to saturate an Intel 40Gbps network card, even though with no packet I/O overheads single thread could saturate such a link. The CPU is capable of executing 24 simultaneous threads of execution (although half of those threads are merely simultaneous multithreads and not physical CPU cores). Thus, the saturation of the 40Gbps network card does not even need to fully use all available CPU resources.

This level of performance (three threads to saturate a 40Gbps network card) cannot unfortunately be matched currently in virtualized environments. The netmap driver for veth network interface is designed for cases where both ends of the veth driver run a netmap application. Standard unmodified veth driver works with only one thread in the case of netmap, and the performance of that single thread is not equivalent to a performance of a single thread in the case of a physical supported network interface card such as the Intel 40Gbps one.

6.3 Protocol and hostname detection performance

To test the performance of protocol and hostname detection, two HTTP requests and one TLS ClientHello message were created. The TLS ClientHello message was 194 bytes long. The HTTP requests were 648 bytes long, with a 27 bytes long URL. Then protocol and hostname detection was performed a million times in a loop to estimate the cost of processing one packet. The setup consist of only part of ldpairwall code in a unit test, so that for example packet processing performance of previous section was not included.

The results are shown in Table 6.2. It can be observed that protocol detection for most typical cases happens in a fraction of a microsecond, so more than million hostnames can be detected per second. This is not a

major limitation of performance in ldpairwall. Practically all HTTP clients send the host header as the first, but it was also tested what happens if the host header is the last header. As expected, performance drops in a major way, but not so much that the drop would make ldpairwall vulnerable to a protocol and hostname detection algorithmic complexity attack.

This protocol and hostname detection is one of the two major performance-related differences between ldpairwall and nmsynproxy. The other is that ldpairwall as a quickly written prototype was created to be single-threaded.

6.4 HTTP and TLS header size study

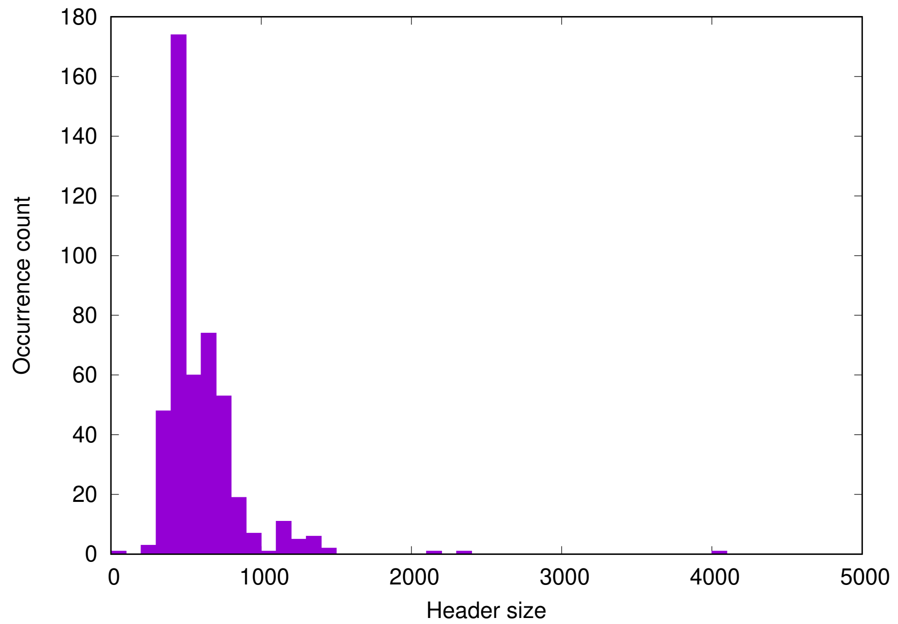
The AL-NAT middlebox needs to store a certain amount of first data packets for each TCP connection in the proxied state. In practice, this memory required cannot be allocated fully dynamically, because otherwise it opens a door for a denial of service attack where the first data packets are deliberately made very large. Partially dynamic allocation with a cap on the memory size is of course possible.

Thus, some investigation is needed to determine how many bytes of initial data is required for determination of the hostname. The current implementation in ldpairwall stores at most 4096 bytes of initial data. This is the same as the maximum supported header block size in nginx, a popular web server implementation, that uses the system page size as the header size limit (on most important systems, page size is 4096 bytes).

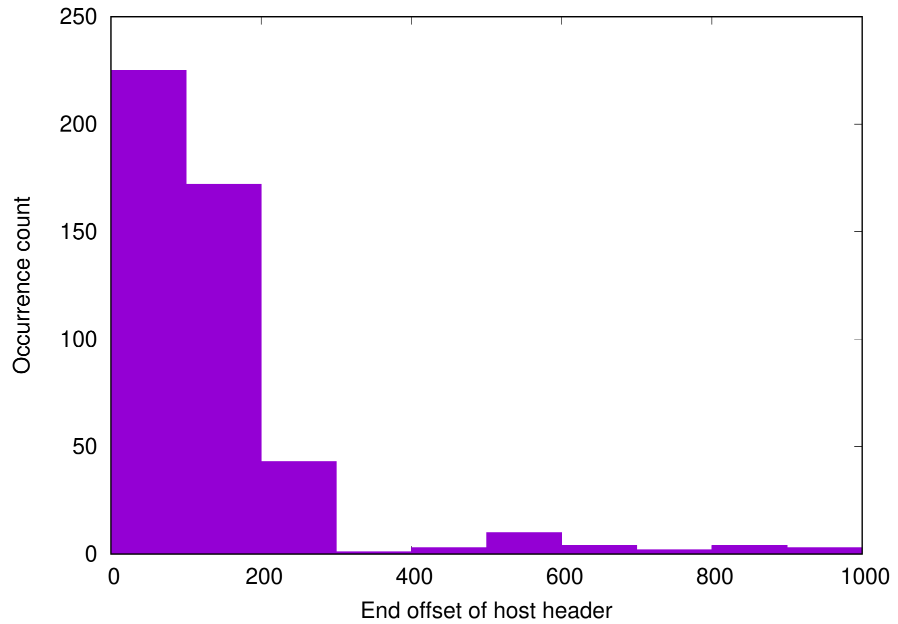
In practice, all noteworthy implementations of HTTP client send the `Host:` header as the first header after the request line. Thus, even if the total HTTP headers may be longer than 4096 bytes, the `Host:` header should occur immediately after the request line. Some implementations of HTTP limit the request URI to a small number of bytes. For example, Internet Explorer supports at most 2083 characters[36].

Therefore, 4096 bytes for initial data may be enough. To understand the distribution of end offset of host name and the initial message size, information from 942 TLS ClientHello messages and 467 plaintext HTTP requests was obtained. For the ClientHello messages, a mixture of browsers (Edge, Chrome, Firefox) was used.

The results in Fig. 6.1 and Fig. 6.2 clearly illustrate that caching only 4096 bytes of data is enough. The whole header block can be nearly 4 kilobytes (there was one such outlier), but even then, the host header value ended within the first kilobyte.

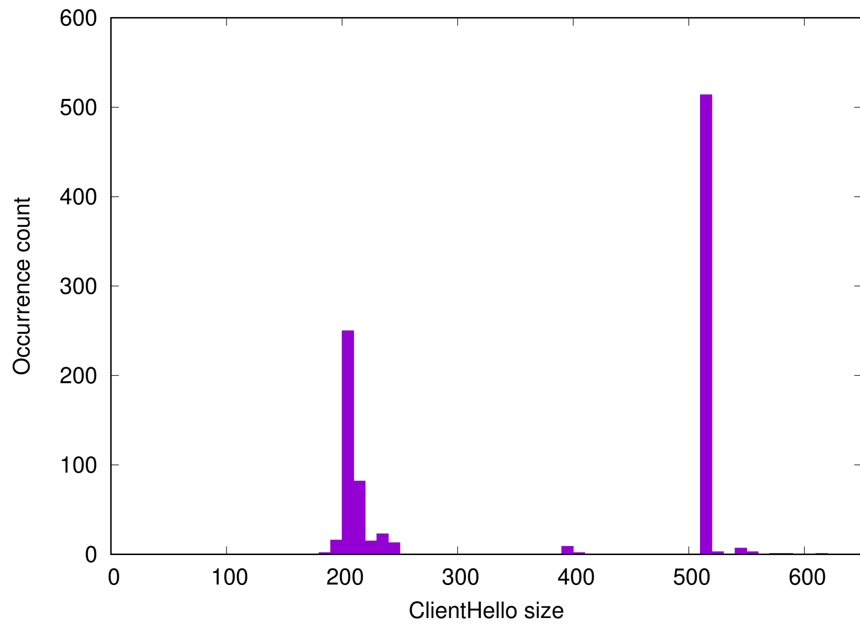


(a) HTTP request block size

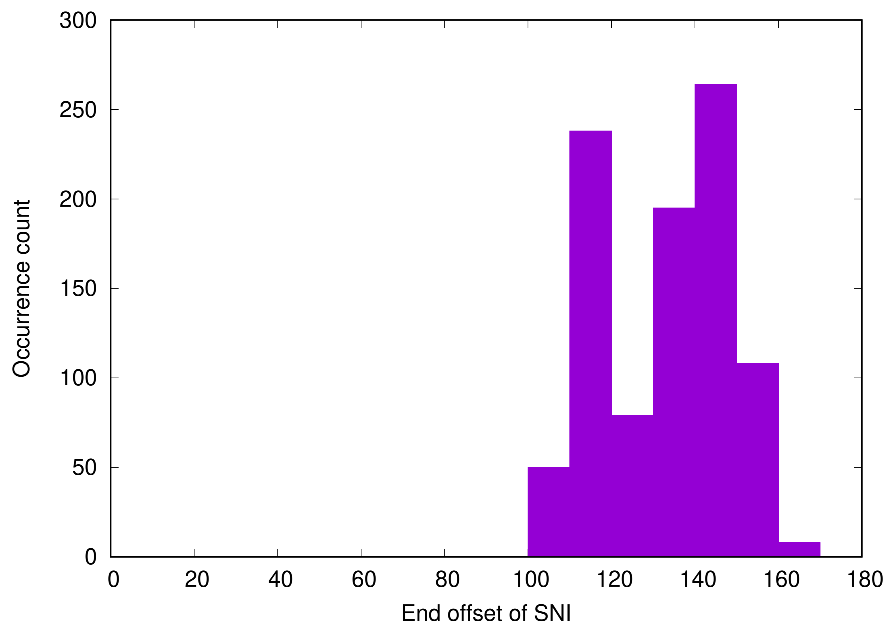


(b) Host header end position

Figure 6.1. A histogram of (a) HTTP request block total size and (b) position of the host header end. One outlier request was 4013 bytes, but even then the host header ended within the firts kilobyte.



(a) TLS ClientHello message size



(b) Position of server name indication end

Figure 6.2. A histogram of (a) TLS ClientHello message size and (b) position of server name indication end. To create these ClientHello messages, a mixture of three web browsers was used. The browsers were Firefox, Chrome and Edge. Google Chrome improves security by padding the ClientHello message to make its size invariant.

6.5 AL-NAT traversal tests

6.5.1 Testing environment

The following configuration changes were performed as the superuser (root):

```
mkdir -p /etc/netns/ns1↵
echo "nameserver 10.150.2.100" > /etc/netns/ns1/resolv.conf↵
ip link add veth0 type veth peer name veth1↵
ip link add veth2 type veth peer name veth3↵
ifconfig veth0 up↵
ifconfig veth1 up↵
ifconfig veth2 up↵
ifconfig veth3 up↵
ethtool -K veth0 rx off tx off tso off gso off gro off lro off↵
ethtool -K veth1 rx off tx off tso off gso off gro off lro off↵
ethtool -K veth2 rx off tx off tso off gso off gro off lro off↵
ethtool -K veth3 rx off tx off tso off gso off gro off lro off↵
ip netns add ns1↵
ip netns add ns2↵
ip link set veth0 netns ns1↵
ip link set veth3 netns ns2↵
ip netns exec ns1 ip addr add 10.150.2.1/24 dev veth0↵
ip netns exec ns2 ip addr add 10.150.1.101/24 dev veth3↵
ip netns exec ns1 ip link set veth0 up↵
ip netns exec ns2 ip link set veth3 up↵
ip netns exec ns1 ip link set lo up↵
ip netns exec ns2 ip link set lo up↵
ip netns exec ns2 ip route add default via 10.150.1.1↵.
```

The ldpairwall was started in the main namespace:

```
./airwall/ldpairwall veth2 veth1↵.
```

All of the subsequent tests assume the configuration changes have been performed and that ldpairwall is up and running.

6.5.2 Outgoing NAT connection

The functionality of outgoing connections through ldpairwall was verified by using netcat. The commands were:

```
ip netns exec ns1 nc -v -v -v -l -p 1234↵
ip netns exec ns2 nc -v -v -v 10.150.2.1 1234↵.
```

The connection to 10.150.2.1:1234 succeeded and typing anything into one netcat instance appeared in the other netcat instance. It was verified the connection was working bidirectionally.

6.5.3 AL-NAT unencrypted connection

The functionality of incoming connections through ldpairwall was verified by using netcat, too. The commands were:

```
ip netns exec ns2 nc -v -v -v -l -p 1234↵
ip netns exec ns1 nc -v -v -v 10.150.2.100 1234↵.
```

In this case, the client saw its connection half open immediately, but the listening server did not immediately see an open connection. However, the following text was typed to the client:

```
GET / HTTP/1.1↵
Host: www1.example.com↵
↵,
```

and now the request typed to the client appeared on the server side, and the connection worked bidirectionally afterwards.

6.5.4 AL-NAT encrypted connection

The previous section tested HTTP protocol connection support. However, AL-NAT supports TLS too and it must also be verified to work. Setting up TLS requires generating a keypair. Executing the following commands tests TLS:

```
openssl req -x509 -newkey rsa:2048 -keyout k.pem -out c.pem -nodes↵
ip netns exec ns2 openssl s_server -key k.pem -cert c.pem -accept 1234↵
alias nns1='ip netns exec ns1'↵
nns1 openssl s_client -servername www1.example.com 10.150.2.100:1234↵.
```

Then when typing anything into the client, it arrives at the server side. It can also be tested that by omitting the servername option, the connection does not work.

6.5.5 HTTP CONNECT proxy

The integrated HTTP CONNECT proxy was verified by using the commands:

```
ip netns exec ns2 nc -v -v -v -l -p 1234↵
ip netns exec ns1 nc -v -v -v 10.150.2.100 4321↵.
```

In this case as well, the client saw its connection half open immediately, but the server did not see any incoming connection yet. Note the server's port number is different. A HTTP CONNECT request was written to the client with the correct port number:

```
CONNECT www1.example.com:1234 HTTP/1.1↵
↵,
```

and now the client showed `HTTP/1.1 200 OK` response. After this, the server saw an incoming connection with no data in it. It was verified that data transfer worked bidirectionally after the connection was established. In particular, none of the CONNECT or OK lines were present on the server side.

6.5.6 Port control protocol, TCP

The opening of ports for TCP[48] using port control protocol[64] (PCP) was tested:

```
ip netns exec ns2 ./airwall/pcpclient tcp 40000 40000 86400↵
ip netns exec ns2 nc -v -v -v -l -p 40000↵
ip netns exec ns1 nc -v -v -v 10.150.2.100 40000↵.
```

Even though the connection was an incoming connection, data transfer worked bidirectionally due to the port control protocol client command.

6.5.7 Port control protocol, UDP

The opening of ports for UDP[45] using port control protocol[64] (PCP) was tested:

```
ip netns exec ns2 ./airwall/pcpclient udp 40000 40000 86400↵
ip netns exec ns2 nc -v -v -v -u -l -p 40000↵
ip netns exec ns1 nc -v -v -v -u 10.150.2.100 40000↵.
```

In this case, data transfer was first tested from the remote client to the protected server. Data transfer in this direction worked. The other direction was tested then and it worked too.

7. Conclusions

It is often said that TCP/IP needs replacing. IPv4 according to this claim needs to be replaced by IPv6 because of address space exhaustion, and TCP needs to be replaced by QUIC because of vulnerability to DoS attacks. In this thesis, it was shown that this is not necessarily the case.

Clever improvements to NAT traversal allow IPv4 to have more life as the de facto network layer protocol in the Internet. Although IPv6 transition is in progress, it is slow. It is best to let it happen at its natural pace, while at the same time deploying advanced NAT traversal solutions that work even in the server side.

The NAT traversal solution presented in this thesis is AL-NAT which works transparently for HTTP and TLS-based protocols using SNI such as HTTPS and QUIC. It also works for other protocols if the client-side middlebox or program is modified to support the HTTP CONNECT proxy protocol. For example, OpenSSH has support for a ProxyCommand configuration option, meaning no code changes are required for adding support to new proxy protocols.

TCP DoS attacks such as SYN flood are easy, but so is protecting against them. SYN proxy can be deployed transparently as a layer 2 inline element, requiring no routing changes in the network where it is placed. The implemented SYN proxy has good multithreaded performance due to being based on netmap.

Due to the improvements implemented in this thesis, TCP/IP has a very bright future as the basis of the Internet. It does not need to be replaced with QUIC/IPv6.

7.1 Future work

There is some work that was not included in this thesis but could be used to enhance the results of the thesis.

The ldpairwall uses a hand-coded parser which is very hard to understand and to maintain. Later, after ldpairwall implementation, a parser-lexer generator named YaLe was implemented by the author. It greatly reduces the workload of creating working parsers for various text-based or binary protocols. The ldpairwall should be changed to use YaLe instead of a hand-coded parser.

When ldpairwall was implemented, the author worked at Nokia that did not look upon implementation of an open source firewall favorably. It was argued that such an open source firewall could compete with Nokia that as a big company has research and development related to firewalls and sales of 3rd party firewalls. Thus, the author had to implement the ideas as a component that is not a firewall. The component chosen was an airwall, a new type of NAT middlebox which has been specifically designed to allow all connections as seamlessly as possible, by favoring connectivity over security. Now that the author no longer works at Nokia, ldpairwall could be enhanced to incorporate a security policy. However, this would be a large project that would add lots of code.

Recently QUIC was defined into an RFC. Thus, ldpairwall should be changed to support the UDP-based protocol QUIC. Doing it would require firstly adding UDP AL-NAT support and then adding QUIC support.

Furthermore, ldpairwall was designed as quickly as possible and thus multithreading was not taken into account. Therefore, ldpairwall is single-threaded and all locations that access global data lack locking. This is not expected to be as large problem as for nmsynproxy that is a SYN flood protection component that has to be fast given its job. In contrast, ldpairwall will be run as a NAT middlebox.

The hash tables of ldpairwall and nmsynproxy should be modified to use MurMurHash3 as the hash function and red-black trees[27] as collision resolution strategy. Currently attacker-created collisions are avoided by using SipHash as the hash function with a secret seed unknown to the attacker. The current collision resolution strategy is a linked list. This current approach has the drawback that SipHash is slow to calculate. Using MurMurHash3 and red-black trees would be faster.

The ldpairwall was implemented before FragmentSmack[40] was discov-

ered. Thus, the IP fragment reassembly methods of ldpairwall may be vulnerable to FragmentSmack. They should be rewritten to be based on red-black trees.

Bibliography

[1] M. Allman, V. Paxson, and W. Stevens. RFC2581: TCP congestion control. RFC 2581, RFC Editor, 1999. URL http://www.rfc-editor.org/rfc/rfc2581.txt.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] F. Audet and C. Jennings. RFC4787: Network address translation (NAT) behavioral requirements for unicast UDP. RFC 4787, RFC Editor, 2007. URL http://www.rfc-editor.org/rfc/rfc4787.txt.

[4] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[5] J.-P. Aumasson, D. J. Bernstein, and M. Boßlet. Hash-flooding DoS reloaded: attacks and defenses. In *29th Chaos Communications Congress*, 2012.

[6] M. Belshe, R. Peon, and M. Thomson. RFC7540: Hypertext transfer protocol version 2 (HTTP/2). RFC 7540, RFC Editor, 2015. URL http://www.rfc-editor.org/rfc/rfc7540.txt.

[7] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC1945: Hypertext transfer protocol — HTTP/1.0. RFC 1945, RFC Editor, 1996. URL http://www.rfc-editor.org/rfc/rfc1945.txt.

[8] D. J. Bernstein. SYN cookies. Personal website, 2013 (last updated), 26.6.2021 (retrieved). URL https://cr.yp.to/syncookies.html.

[9] M. Bishop. HTTP/3. RFC 9114, RFC Editor, 2022. URL http://www.rfc-editor.org/rfc/rfc9114.txt.

[10] R. Braden, D. Borman, and C. Partridge. RFC1071: Computing the Internet checksum. RFC 1071, RFC Editor, 1988. URL http://www.rfc-editor.org/rfc/rfc1071.txt.

[11] S. Bradner and A. Mankin. RFC1550: IP: Next generation (IPng) white paper solicitation. RFC 1550, RFC Editor, 1993. URL http://www.rfc-editor.org/rfc/rfc1550.txt.

[12] S. Cheshire and M. Krochmal. RFC6886: NAT port mapping protocol (NAT-PMP). RFC 6886, RFC Editor, 2013. URL http://www.rfc-editor.org/rfc/rfc6886.txt.

[13] M. Crispin. RFC3501: Internet message access protocol — version 4rev1. RFC 3501, RFC Editor, 2003. URL http://www.rfc-editor.org/rfc/rfc3501.txt.

[14] V. T. Dang, T. T. Huong, N. H. Thanh, P. N. Nam, N. N. Thanh, and A. Marshall. SDN-based SYN proxy — a solution to enhance performance of attack mitigation under TCP SYN flood. *The Computer Journal*, 62(4):518–534, 2018.

[15] S. Deering and R. Hinden. RFC1883: Internet protocol, version 6 (IPv6) specification. RFC 1883, RFC Editor, 1995. URL http://www.rfc-editor.org/rfc/rfc1883.txt.

[16] S. Deering and R. Hinden. RFC8200: Internet protocol, version 6 (IPv6) specification. RFC 8200, RFC Editor, 2017. URL http://www.rfc-editor.org/rfc/rfc8200.txt.

[17] R. Droms. RFC2131: Dynamic host configuration protocol. RFC 2131, RFC Editor, 1997. URL http://www.rfc-editor.org/rfc/rfc2131.txt.

[18] D. Eastlake and P. Jones. RFC3174: US secure hash algorithm 1 (SHA1). RFC 3174, RFC Editor, 2001. URL http://www.rfc-editor.org/rfc/rfc3174.txt.

[19] D. Eastlake 4rd. RFC6066: Transport layer security (TLS) extensions: Extension definitions. RFC 6066, RFC Editor, 2011. URL http://www.rfc-editor.org/rfc/rfc6066.txt.

[20] W. Eddy. RFC4987: TCP SYN flooding attacks and common mitigations. RFC 4987, RFC Editor, 2007. URL http://www.rfc-editor.org/rfc/rfc4987.txt.

[21] K. Egevang and P. Francis. RFC1631: The ip network address translator (NAT). RFC 1631, RFC Editor, 1994. URL http://www.rfc-editor.org/rfc/rfc1631.txt.

[22] R. Fielding and J. Reschke. RFC7230: Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. RFC 7230, RFC Editor, 2014. URL http://www.rfc-editor.org/rfc/rfc7230.txt.

[23] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. RFC 1519, RFC Editor, 1993. URL http://www.rfc-editor.org/rfc/rfc1519.txt.

[24] F. Gont, V. Manral, and R. Bonica. RFC7112: Implications of oversized IPv6 header chains. RFC 7112, RFC Editor, 2014. URL http://www.rfc-editor.org/rfc/rfc7112.txt.

[25] B. Goode. Voice over internet protocol (VoIP). *Proceedings of the IEEE*, 90(9): 1495–1517, 2002.

[26] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. RFC5382: NAT behavioral requirements for TCP. RFC 5382, RFC Editor, 2008. URL http://www.rfc-editor.org/rfc/rfc5382.txt.

[27] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on, IEEE*, 1978.

[28] J. Iyengar and I. Swett. RFC9002: QUIC loss detection and congestion control. RFC 9002, RFC Editor, 2021. URL http://www.rfc-editor.org/rfc/rfc9002.txt.

[29] J. Iyengar and M. Thomson. RFC9000: QUIC: A UDP-based multiplexed and secure transport. RFC 9000, RFC Editor, 2021. URL http://www.rfc-editor.org/rfc/rfc9000.txt.

[30] V. Jacobson, R. Braden, and D. Borman. RFC1323: TCP extensions for high performance. RFC 1323, RFC Editor, 1992. URL http://www.rfc-editor.org/rfc/rfc1323.txt.

[31] J. Klensin. RFC5321: Simple mail transfer protocol. RFC 5321, RFC Editor, 2008. URL http://www.rfc-editor.org/rfc/rfc5321.txt.

[32] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC1928: SOCKS protocol version 5. RFC 1928, RFC Editor, 1996. URL http://www.rfc-editor.org/rfc/rfc1928.txt.

[33] J. Llorente Santos. RealmGateway. Github, 2018 (last updated), 26.6.2021 (retrieved). URL https://github.com/Aalto5G/RealmGateway.

[34] J. Llorente Santos, R. Kantola, N. Beijar, and P. Leppaaho. Implementing NAT traversal with private realm gateway. In *Communications (ICC), 2013 IEEE International Conference on*, 2013.

[35] T. Mallory and A. Kullberg. RFC1141: Incremental updating of the Internet checksum. RFC 1141, RFC Editor, 1990. URL http://www.rfc-editor.org/rfc/rfc1141.txt.

[36] Microsoft. Maximum URL length is 2,083 characters in internet explorer, 2019 (last updated), 25.12.2019 (retrieved). URL https://support.microsoft.com/en-us/help/208427/maximum-url-length-is-2-083-characters-in-internet-explorer.

[37] P. Mockapetris. RFC1034: Domain names — concepts and facilities. RFC 1034, RFC Editor, 1987. URL http://www.rfc-editor.org/rfc/rfc1034.txt.

[38] P. Mockapetris. RFC1035: Domain names — implementation and specification. RFC 1035, RFC Editor, 1987. URL http://www.rfc-editor.org/rfc/rfc1035.txt.

[39] J. Myers and M. Rose. RFC1939: Post office protocol — version 3. RFC 1939, RFC Editor, 1996. URL http://www.rfc-editor.org/rfc/rfc1939.txt.

[40] T. Novelly (document writer) and J.-M. Tilli (reporter). VU#641765: Linux kernel IP fragment re-assembly vulnerable to denial of service. Vulnerability Note 641765, CERT, 2018. URL https://www.kb.cert.org/vuls/id/641765.

[41] V. Olteanu and D. Niculescu. SOCKS protocol version 6, historical. Internet-Draft draft-olteanu-intarea-socks-6-11, IETF Secretariat, 2020.

[42] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM SIGCOMM Computer Communication Review*, 31(3):38–47, 2001.

[43] R. Penno, S. Perreault, M. Boucadair, S. Sivakumar, and K. Naito. RFC7857: Updates to network address translation (NAT) behavioral requirements. RFC 7857, RFC Editor, 2016. URL http://www.rfc-editor.org/rfc/rfc7857.txt.

[44] D. C. Plummer. RFC826: An ethernet address resolution protocol — or — converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. RFC 826, RFC Editor, 1982. URL http://www.rfc-editor.org/rfc/rfc826.txt.

[45] J. Postel. RFC768: User datagram protocol. RFC 768, RFC Editor, 1980. URL http://www.rfc-editor.org/rfc/rfc768.txt.

[46] J. Postel. RFC792: Internet control message protocol. RFC 792, RFC Editor, 1981. URL http://www.rfc-editor.org/rfc/rfc792.txt.

[47] J. Postel. RFC791: Internet protocol. RFC 791, RFC Editor, 1981. URL http://www.rfc-editor.org/rfc/rfc791.txt.

[48] J. Postel. RFC793: Transmission control protocol. RFC 793, RFC Editor, 1981. URL http://www.rfc-editor.org/rfc/rfc793.txt.

[49] A. Ramaiah, R. Stewart, and M. Dalal. RFC5961: Improving TCP's robustness to blind in-window attacks. RFC 5961, RFC Editor, 2010. URL http://www.rfc-editor.org/rfc/rfc5961.txt.

[50] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. RFC1918: Address allocation for private internets. RFC 1918, RFC Editor, 1996. URL http://www.rfc-editor.org/rfc/rfc1918.txt.

[51] E. Rescorla. RFC8446: The transport layer security (TLS) protocol version 1.3. RFC 8446, RFC Editor, 2018. URL http://www.rfc-editor.org/rfc/rfc8446.txt.

[52] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood. TLS encrypted client hello, work in progress. Internet-Draft draft-ietf-tls-esni-14, IETF Secretariat, 2022.

[53] M. Riaz. Extending the Functionality of the Realm Gateway. Master's thesis, Aalto University, 2019.

[54] P. Richter, F. Wohlfart, N. Vallina-Rodriguez, M. Allman, R. Bush, A. Feldmann, C. Kreibich, N. Weaver, and V. Paxson. A multi-perspective analysis of carrier-grade NAT deployment. In *Proceedings of the 2016 Internet Measurement Conference*, pages 215–229, 2016.

[55] A. Rijsinghani. RFC1624: Computation of the Internet checksum via incremental update. RFC 1624, RFC Editor, 1994. URL http://www.rfc-editor.org/rfc/rfc1624.txt.

[56] L. Rizzo. Revisiting network I/O APIs: the netmap framework. *Communications of the ACM*, 55(3):45–51, 2012.

[57] L. Rizzo and M. Landi. netmap: memory mapped access to network devices. *ACM SIGCOMM Computer Communication Review - SIGCOMM '11*, 41(4): 422–423, 2001.

[58] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *INFOCOM, 2012 Proceedings IEEE*, 2012.

[59] P. Srisuresh, B. Ford, S. Sivakumar, and S. Guha. RFC5508: NAT behavioral requirements for ICMP. RFC 5508, RFC Editor, 2009. URL `http://www.rfc-editor.org/rfc/rfc5508.txt`.

[60] M. Thomson and S. Turner. RFC9001: Using TLS to secure QUIC. RFC 9001, RFC Editor, 2021. URL `http://www.rfc-editor.org/rfc/rfc9001.txt`.

[61] G. van Rooij. Real stateful TCP packet filtering in IP filter. In *10th USENIX Security Symposium*, 2001.

[62] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 19–35, 2005.

[63] J. Weil, V. Kuarsingh, C. Donley, C. Liljenstolpe, and M. Azinger. RFC6598: IANA-reserved IPv4 prefix for shared address space. RFC 6598, RFC Editor, 2012. URL `http://www.rfc-editor.org/rfc/rfc6598.txt`.

[64] D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk. RFC6887: Port control protocol (PCP). RFC 6887, RFC Editor, 2013. URL `http://www.rfc-editor.org/rfc/rfc6887.txt`.

[65] T. Ylonen and C. Lonvick. RFC4251: The secure shell (SSH) protocol architecture. RFC 4251, RFC Editor, 2006. URL `http://www.rfc-editor.org/rfc/rfc4251.txt`.

[66] T. Ylonen and C. Lonvick. RFC4252: The secure shell (SSH) authentication protocol. RFC 4252, RFC Editor, 2006. URL `http://www.rfc-editor.org/rfc/rfc4252.txt`.

[67] T. Ylonen and C. Lonvick. RFC4253: The secure shell (SSH) transport layer protocol. RFC 4253, RFC Editor, 2006. URL `http://www.rfc-editor.org/rfc/rfc4253.txt`.

[68] T. Ylonen and C. Lonvick. RFC4254: The secure shell (SSH) connection protocol. RFC 4254, RFC Editor, 2006. URL `http://www.rfc-editor.org/rfc/rfc4254.txt`.

[69] L. Zhang. A retrospective view of network address translation. *IEEE network*, 22(5), 2008.