Weiwei Hu

Proxy for Host Identity Protocol

Aalto University
Department of Computer Science and Engineering
Data Communication Software Group

AALTO UNIVERSITY ABSTRACT OF THE MASTER'S THESIS

Author:	Weiwei Hu		
Name of the thesis:	Proxy for Host Identity Protocol		
Date:	April 22, 2010	**Number of pages:**	59
Department:	Department of Computer Science and Engineering		
Professorship:	T-110		
Supervisor:	Professor. Antti Ylä-Jääski		
Instructors:	M.Sc. Miika Komu		

In computer networks, a proxy acts as an intermediary computer system or an application program which serves the requests of its clients by forwarding their packets or data flows to some servers. Host Identity Protocol (HIP) provides security extension as well as mobility on top of TCP/IP stack. However, HIP requires some changes to the networking stacks which can be considered as a deployment obstacle. A HIP proxy provides a solution which can provide a transition path towards HIP-capable networks.

In the context of this thesis, a HIP proxy translates packet flows from legacy clients to HIP-capable servers without requiring any HIP-development at the client side. The goal of this master thesis is to design and implement a prototype of a HIP proxy which supports basic protocol translation from non-HIP based communications to HIP based communications.

In this thesis, We also present a performance analysis of the implemented prototype against communications without the proxy. We also compare the design of the proxy against another and describe future work items, such as server-side HIP proxy.

Keywords: HIP, Host, Identity, Protocol, proxy, translation, split, location, VPN

Acknowledgements

This Master's thesis has been done for a part of the InfraHIP project, which is a co-operation project of Helsinki Institute of Information Technology (HIIT), Helsinki University of Technology (Aalto University) and RWTH Aacher.

I would like to thanks my supervisor Antti Ylä-Jääski to offer the opportunity to work with the development team in HIIT. Special thanks to my instructor Miika Komu for expertise help with HIPL. His comments and corrections have helped to maintain steady progress with thesis.

Sincere thanks to all the people in the InfraHIP project and colleagues in the DSC group for providing such a nice working environment. It has been a pleasure to work with you all.

This work is dedicated to my wife Liu Wei for all her love and encouragement she has given to me. I want to give a big thank to my parents for their support in my study and life.

Espoo, April 22, 2010

Weiwei Hu

Contents

Abbreviations

AID Application Identifier

API Application Programming Interface

DSA Digital Signature Algorithm

DHT Distributed Hash Table

DNS Domain Name System

DoS Denial of Service

ED Endpoint Descriptor

FQDN Fully Qualified Domain Name

FTP File Transfer Protocol

HAA Host Assigning Authority

HIP Host Identity Protocol

HI Host Identifier

HIPL HIP for Linux

HIT Host Identity Tag

IETF Internet Engineering Task Force

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

IPsec Internet Protocol security

LSI Local Scope Identifier

PEM Privacy Enhanced Mail

PKI Public Key Infrastructure

POSIX Portable Operating System Interface

QoS Quality of Service

SA Security Association

SP Security Policy

SRV Service Record

SSH Secure Shell

TCP Transport Control Protocol

TLI Transport Layer Identifier

UDP User Datagram Protocol

UI User Interface

UID User ID

WLAN Wireless Local Area Network

XTI X/Open Transport Interface

Chapter 1

Introduction

Internet is widely used all around the world. The number of internet users is growing as well as the number of user equipments connected to those networks. The networks are also becoming increasingly more interconnected.

The current Internet architecture is based on IPv4 which has been invented 20 years ago. Since the development of the network technology, it cannot meet its new requirements any more. For example, mobility and security are not supported by the initial design of the internet architecture. IP addresses are used to route traffic from the source to the destination node as well as serving as the identifiers of the nodes. When a node moves to another location in the network topology, the IP address of the node usually changes and, hence, its identifier also changes. This means the same node has different identifiers on different locations. This causes many problems, including disrupting of TCP-based media streams, because the identity of transport layer connection is coupled with the network layer location. Fortunately, a number of solutions address this problem, for example Mobile IP. Many of the solutions are workarounds for the current network architecture and are therefore overly complicated or inefficient. Thus, more straightforward solutions are called for.

Several alternatives have been proposed to extend or redesign the current TCP/IP architecture to face the new challenges of the Internet. Host Identity Protocol (HIP) is OK and was also chosen as the experimentation framework for this thesis. HIP separates the coupled functionalities of the current TCP/IP suite: node identifier and its location. Host Identity (HI) provides the node identifier functionality in HIP and it is effectively a public key. The IP address acts as a locator that provides only routing information for the current node. In HIP, the transport layer is bound to HIs and they are translated dynamically to IP addresses by HIP layer. This introduces persistent and location-independent identifiers to the existing Internet architecture, hence allowing e.g. TCP streams to survive network changes.

To support the security, each HIP-capable host has an asymmetric key pair. The public part of the key acts as the HI of the host. HIP authenticates two end-hosts to each other using four-way Diffie-Hellman procedure called base exchange. This

also sets up the symmetric key material for IPSec.

The introduction of HIP layer, and in certain cases also IPSec, requires additional software installation at the client and server side. Deployment at both sides can introduce a high cost for HIP deployment. As there are more client hosts in the Internet, we have chosen to mitigate client HIP support to HIP based proxies in order to boost HIP deployment.

1.1 Goals

Millions of client-side hosts exist in the current Internet. The deployment of HIP to all end-hosts can take several years. Upgrading all legacy end-hosts one-by-one can be expensive and some production environments running rather old OS versions may never be upgraded. Thus, a transition mechanism for legacy end-hosts seems useful.

The main goal of this thesis is to implement a HIP proxy which provides translation services between legacy end-clients and HIP-capable servers. A HIP proxy intercepts network traffic and tries to translate it from IP to HIP or vice versa:

- When the HIP proxy receives non-HIP traffic destined to a HIP-capable server, the proxy performs a base exchange with the HIP-capable server and translates the packet into an ESP tunnel.
- When the HIP proxy receives traffic from an ESP tunnel, it translates the packet to a non-HIP packet and forwards it to the correct client.
- The proxy translates only non-HIP connections originating from legacy clients. It does not translate connections originating from HIP-capable clients.

This thesis is organized as follows:

Chapter 2 presents some background for the work, the current and HIP-based Internet architecture. We also discuss some HIP protocol extensions.

Chapter 3 focuses on the design of HIP proxy. We illustrate the architecture and interface in detail.

Chapter 4 presents the details of the implementation of the HIP proxy. Different scenarios of HIP proxy are also introduced.

Chapter 5 presents the result of the performance analysis and compatibility with HIP extensions. The chapter also shows experimentation results with some application-layer protocols.

In Chapter 6 and Chapter 7, the future work and conclusions are presented.

Chapter 2

Background

This chapter presents an overview of the background topics. The focus of this chapter is on the current network stacks and their drawbacks. HIP and its extension are described in the end of this chapter.

2.1 TCP/IP

2.1.1 TCP/IP architecture

Today's TCP/IP architecture presents itself as a multi-layer architecture. Each layer has a set of roles related to the transmission and reception of data, and provides services and interfaces to the upper layer protocols. Upper layers are logically closer to the user and deal with more abstract data, relying on lower layer protocols to translate data into forms that can eventually be physically transmitted [32].

TCP and IP are the main protocols in the TCP/IP architecture, which is generally divided into four abstraction layers [4]. Figure 2.1 visualizes four layers: application layer, transport layer, internet layer and network interface layer.

Application layer basically consists of application programs and user interfaces. It refers to the higher-level protocols used by most applications for network communication. Examples of application layer protocols include the FTP and HTTP [30]. An application marshals its data structures into one or more transport layer protocols according to the application layer protocol. Then, transport layer passes the data to network layer.

Transport layer provides end-to-end connectivity between hosts. It defines the level of service and status of the connection used when transporting the data [5]. The main protocols that are used in this layer are TCP [23] and UDP [22].

The responsibility of the internet layer is to deliver IP packets between hosts in networks. Routing of packets is handled at the internet layer. With the advent of inter-networking, this layer provides functionalities to deliver data from the source

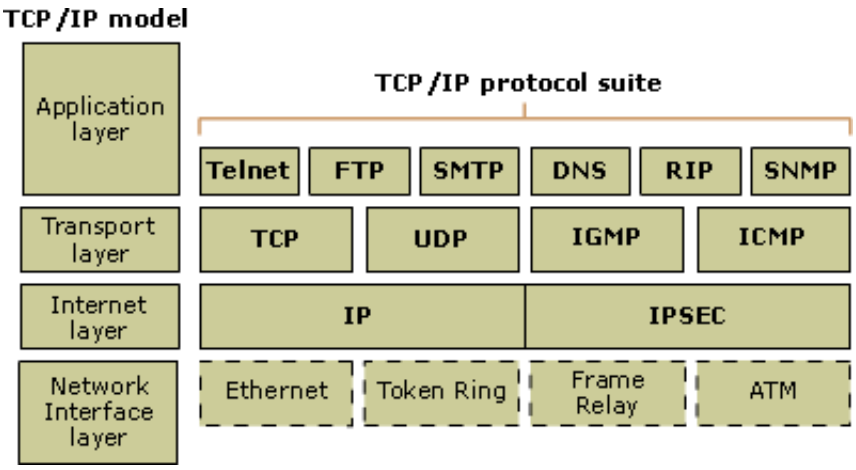

Figure 2.1: Network Stack
[5]

network to the destination network hop-by-hop. This is generally called packet routing. The protocols for the maintenance and boot strapping for Internet layer are ICMP ARP, RARP and IGMP.

Network interface layer is responsible for specifying how the data will be sent through the network physically. This layer handles how bits are electrically signaled by the hardware devices that interface directly with a network medium such as the coaxial cable or the twisted pair copper wire[18]. Protocols operating at this layer include the "Token Ring", "Ethernet", FDDI and IEEE 802x.

2.1.2 IPv4 vs IPv6

IPv4 has been widely deployed and used in the Internet today. With the rapid growth of the Internet, improvements to the internet protocol are needed to support the influx of new subscribers, Internet-enabled devices and applications. IPv6 is designed to enable the larger expansion of the Internet with over more IP-based devices.

The larger address space affected by IPv6 is the most important feature compared to IPv4. An IPv6 address is 128 bits long, while IPv4 address size is 32 bits. The IPv6 address space provides 2^{128} addresses which is 2^{96} times larger than what IPv4 can provide. With IPv6, each people in the world could have roughly $5x10^{28}$ IP addresses.

IPv6 also brings also major changes to the IP header. The header of IPv6 is more flexible and contains fewer fixed fields. Option fields in the header have been replaced by a set of optional extensions in IPv6.

The header of IPv6 is efficient, which can be seen by comparing the address to the size of header. Even though an IPv6 address is four times as large as an IPv4 address,

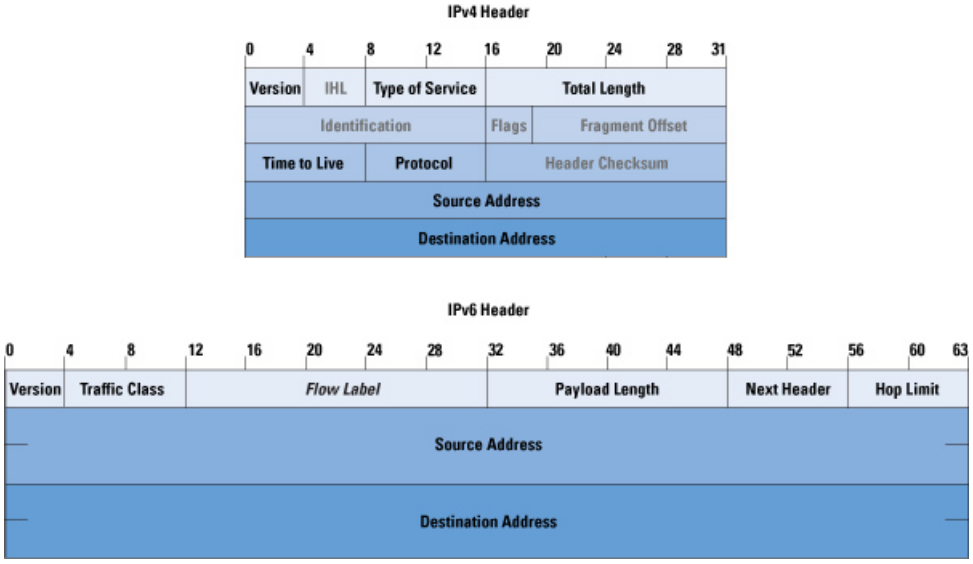

Figure 2.2: IPv4 vs IPv6

the header is only twice as large. Figure 2.2 presents the different structures in the IPv4 header and IPv6 header.

Options can increase the size of the IPv4 header which are absent from the IPv6 header. Instead, Next Header field in the IPv6 header indicates the existence of options in the header. For routers, they would just check the Next Header field to identify the existence of options instead of parsing the whole IPv6 header. Thus, the simpler structure of the IPv6 facilitates faster packets processing time in routers when options are present.

Security for IPv6 is a first class citizen rather than an add-on feature. For example, IPSec is optionally available for IPv4 implementations while it is an integral part of the security model of IPv6. IPv6 also includes an end-to-end security model that is designed to protect DHCP, DNS [14] and IPv6 mobility [6].

2.2 Host Identity Protocol

In this section, we give an overview of HIP and its related extensions.

2.2.1 A New Namespace

HIP introduces a new namespace for the Internet. The namespace provides location independent identification of endpoints. This decouples the network-layer identifiers from the upper-layer identifiers and provides mobility and multi-homing capabilities at the network layer [19]. In HIP, a host has a fixed endpoint identifier but multiple changing network layer addresses. The decoupling allows more flexible mobility

support since the location can be dynamically associated to the identity [33].

HI

In HIP, a HI presents an endpoint. A HI is basically the public key of an asymmetric key pair. Using a public key based HI, the security of the communication and end-host authentication are improved. HIP supports Rivest Shamir Adelman (RSA/SHA1) [1] public key algorithm and Digital Signature Algorithm (DSA) [7] algorithms. By using cryptographic end-point identifiers, end-host impersonation becomes difficult in HIP.

HIT

A hashed encoding of the HI, the Host Identity Tag (HIT), is used in upper-layer protocols to represent the Host Identity. A Host Identity Tag is a 128-bit representation for a Host Identity. Its fixed length suits better in protocol encoding and decreases control packet size. In addition, it presents the identity in a consistent format to the upper-layer protocol independent of the cryptographic algorithms used. HIT is the same length as an IPv6 address which can be used directly in IPv6 capable applications and APIs. On the other hand, it is designed to be self-certifying and the probability of a HIT collision between two hosts is very low.

In HIP control packets, the HITs identify the sender and recipient of a packet. Consequently, a HIT should be unique in the whole IP universe as long as it is being used. In the extremely rare case of a single HIT mapping to more than one Host Identity, the Host Identifiers (public keys) will make the final difference. If there is more than one public key for a given node, the HIT acts as a hint for the correct public key to use.

LSI

The Local Scope Identifier (LSI) is a 32-bit localized representation for a HI. It facilitates using HIs in existing protocols and APIs. An LSI has the same length as an IPv4 address so that it can support IPv4-only legacy applications. LSIs are only valid in the context of the local host. The disadvantage of LSI is its local scope [16].

2.2.2 A New Layer

In the current Internet model, IP addresses are used for routing. Each IP address represents a physical location in the internet, hence, acting as a locator. Meanwhile, the IP address also names an end-point. HIP separates the end point names and locators from each other. The Host Identifier takes the responsibility of naming endpoint and IP address still act as a routing locator. HIP requires a translation mechanism between host identifiers and IP addresses because host identifiers are not

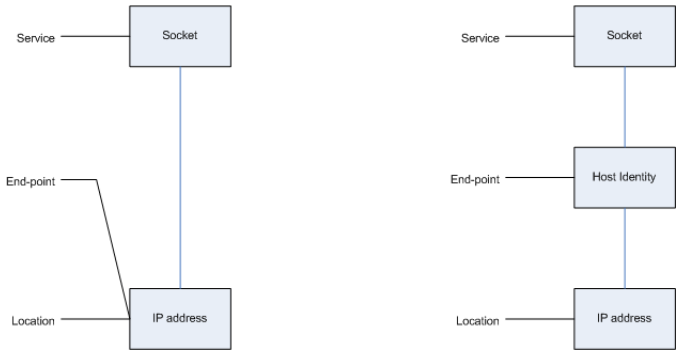

Figure 2.3: HIP Stack
[16]

Next header	Header Length	Packet type	VER	RES
Controls		Checksum		
Sender's Host Identity Tag (HIT)				
Receiver's Host Identity Tag (HIT)				
HIP parameters				

Figure 2.4: HIP Packet Structure
[17]

routable. Therefore, HIP architecture introduces a layer which is called Host Identity Layer between the transport and network layer. A comparison between the current network architecture and the HIP-based architecture is illustrated in Figure 2.3.

2.2.3 HIP packet structure

HIP header in all HIP packets contains fixed fields, including both sender's and receiver's HITs. In the first packet, I1, the receiver's HIT may also be zero, if it is unknown (opportunistic HIP). Figure 2.4 presents an overview of HIP packet structure.

2.2.4 HIP Base Exchange

HIP base exchange is a four-way handshake to authenticate the end-hosts to each other, i.e., to verify that possess the private keys corresponding to their host identifiers. In addition, the base exchange includes a computational puzzle to protect hosts against DoS attacks. As HIP base exchange is designed as an end-to-end authenti-

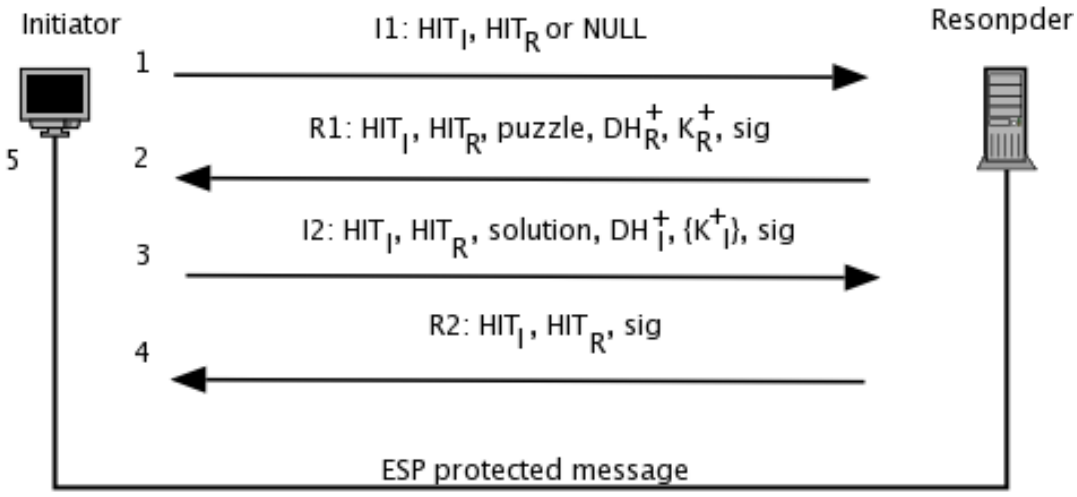

Figure 2.5: HIP Base Exchange
[2]

cation and key establishment protocol, it can be used with Encapsulated Security Payload (ESP) [24]. It is possible to extend HIP to use e.g. S-RTP.

In HIP, the system initiating a HIP base exchange (the client) is the Initiator, and the contacted host (the server) is the Responder.

The Initiator starts the handshake procedure by sending an I1 packet. The I1 packet is a "trigger" packet that contains the HIT of the Initiator and (optionally) the HIT of Responder. When the peer receives the I1 packet, it sends an R1 packet which contains a cryptographic challenge that the Initiator must solve before continuing the exchange. In addition, Diffie-Hellman [25] related parameters included. The initiator solves the puzzle and responds with an I2 packet. The I2 contains a Diffie-Hellman parameter and SPI number. If the solution for the challenge is not correct, the responder discards the I2 packet. Finally, the responder finalizes the base exchange by sending an R2 packet. All packets in the base exchange are signed with the public key of the originating host except the I1 packet.

2.2.5 HIP opportunistic mode

Initiator uses opportunistic HIP when it does not know the HIT of the responder. It sends an I1 packet to the responder by using NULL (all zeros) as the Responder's HIT and waits until it receives a R1 packet from the peer or times out. The Host Identity Protocol for Linux (HIPL) [9] implementation proceeds with non-HIP

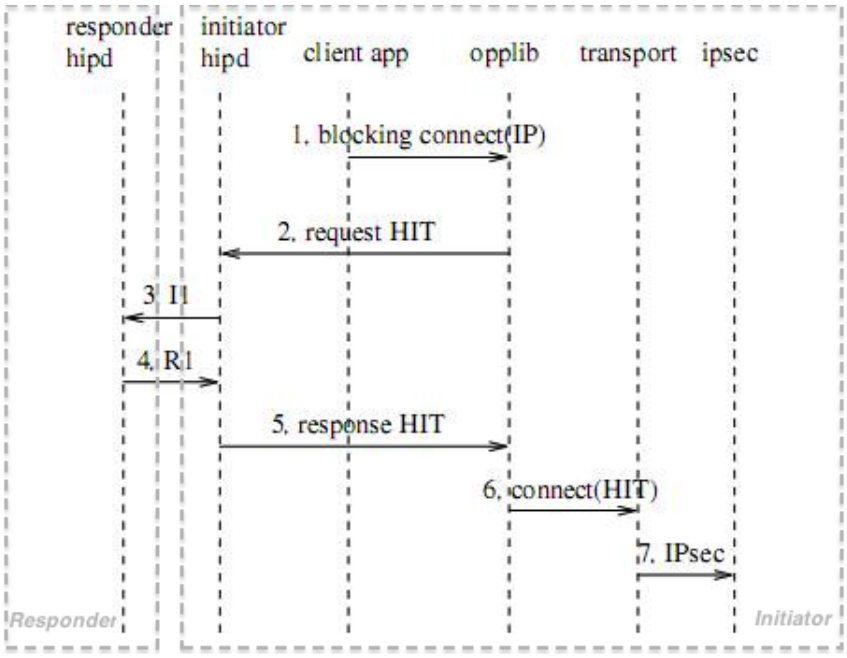

Figure 2.6: Flow Diagram of HIP Base Exchange in Opportunistic Mode

communications upon the time out.

Figure 2.6 illustrates the process of the opportunistic handshake in HIPL implementation:

1. The application calls a socket API [31] function that sends data using IP addresses. The opportunistic library then intercepts the function call.
2. The opportunistic library (opplib in Figure 2.6) queries the HIP daemon for the corresponding HIT. The query blocks until the HIP daemon responds.
3. The daemon triggers an opportunistic base exchange with the peer.
4. The HIP daemon at the initiator receives the R1 packet.
5. The HIP daemon at the initiator communicates the responder's HIT to the opportunistic library and proceeds with the base exchange.
6. The opportunistic library proceeds with the translation and connects to the HIT of the responder.
7. The control flow proceeds from transport to IPSec layer processing which finally transmits the data over ESP.

Finez describes a TCP-based optimization for HIP opportunistic mode [13]. The initiator sends an I1 packet together with a TCP SYN message with a special TCP

option. If the responder is a HIP-capable host, it processes the I1 packet and replies with a R1 packet. After the initiator receives the R1 packet from the responder, the base exchange continues. On the other hand, if the responder is not HIP-capable, it cannot understand the I1 packet but can process the TCP packet. The responder replies to the initiator with a TCP SYN ACK without the TCP option. At this point, the initiator detects that the peer does not support HIP and falls back to unprotected TCP connection.

Opportunistic mode is vulnerable to certain types of attacks. An attacker could guess the time of sending of an I1 and reply with his own R1 packet. Alternatively, the attacker could just sniff the I1 if it can read the packets or if it is on the path. Moreover, a man-in-middle attacker could just drop the I1, and force a fall-back to TCP/IP to inspect the traffic. In any case, opportunistic mode offers "better than nothing" security using the Leap of Faith model [12].

2.3 NAT Traversal

Network Address Translation is a method to provide transparent routing to hosts. Address translation allows hosts in a private network to transparently communicate with destinations on a public network by sharing a single public address and vice versa [29]. In this section, we attempt to give a brief introduction to the different NAT traversal techniques.

2.3.1 Types of NATs

Basically, we can divide NATs types into three types:

Cone NATs

Cone NATs stores a mapping between an internal address and port number, and an external address and port number. Once the NAT translation table entry is in place, the NAT translate the inbound traffic to the external address and port number from any source address and port number.

Restricted NATs

Restricted NAT stores a mapping between an internal address and port number, and an external address and port number, for either specific source addresses or specific source address and port numbers. The NAT discards the inbound packet that matches the NAT translation table entry for the external destination address and port number from an unknown external address or port number.

Symmetric NATs

Symmetric NAT maps the same internal address and port number to different external addresses and ports, depending on the external destination address of the outbound traffic. Only an external host that receives a packet from an internal host can send a packet back.

2.3.2 Teredo

Teredo is a tunneling protocol which grants IPv6 connectivity to nodes that are located behind one or multiple IPv4-based NAT devices. It encapsulates IPv6 packets into IPv4-based UDP tunnels that can be routed through NAT devices and in the IPv4 internet. However, Teredo is not compatible with all NAT devices. Full cone and restricted NAT devices are supported, while symmetric NATs are not [10].

The Teredo protocol diagnoses UDP over IPv4 connectivity and discovers the type of NAT present by using a simplified replacement to the STUN protocol. After that, it assigns a globally IPv6 address to the host and tunnels IPv6 packets over UDP for transmission over an IPv4 network. Teredo supports also connectivity between Teredo-based hosts and native IPv6 hosts.

Teredo communication is facilitated by Teredo servers and Teredo relays. Teredo servers are used by Teredo clients to autodetect the type of NAT behind which they are located. A Teredo client sends a UDP packet to its Teredo server at regular time intervals to maintain a binding on its NAT. This ensures that the server can be contacted by any of its clients, which is required for hole-punching to work properly. A Teredo relay is responsible for receiving traffic from the IPv6 hosts addressed to any Teredo client, and forwarding it over UDP/IPv4. Symmetrically, it also receives packets from Teredo clients addressed to native IPv6 hosts over UDP and IPv4 and detunnels packets to the native IPv6 network.

2.3.3 Interactive Connectivity Establishment

Interactive Connectivity Establishment (ICE) is another technology used to solve the NAT traversal issues for different Internet applications. In ICE philosophy, a node does not try to detect the type or presence of NATs. A node just tries with a brute-force approach to find a working address pair with its peer node [26].

In ICE, each node has a variety of candidate TRANSPORT ADDRESSES which is usually defined as a combination of IP address and port for a particular transport protocol. Those candidate transport addresses include [26]:

- A transport address on an attached network interface
- A translated transport address on the public side of a NAT
- The transport address allocated from a TURN server

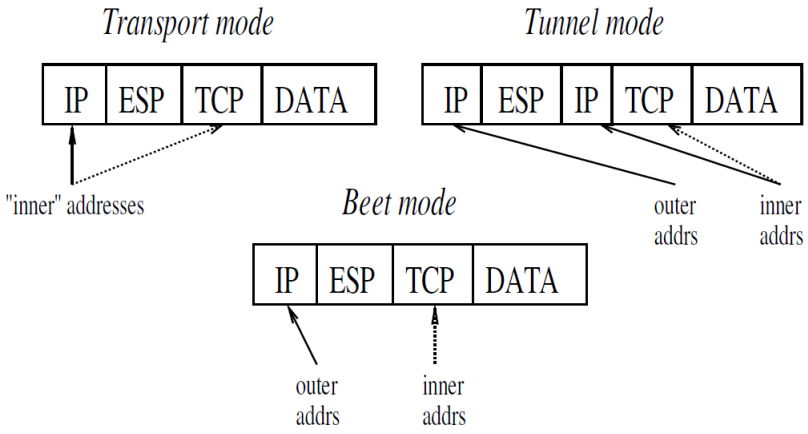

Figure 2.7: Three IPSec Mode

The purpose of ICE is to discover which pairs of addresses will work. In this way, an ICE implementation systematically tries all possible pairs until it finds one or more pair that works.

2.4 HIP and IPSec

2.4.1 Bound End-to-End Tunnel (BEET)

HIP base exchange is used to set up a HIP association between two hosts. The base exchange provides two-way host authentication and IPSec symmetric key generation. HIP uses a new IPSec mode called Bound End-to-End Tunnel for data transmission after the HIP base exchange. The new mode combines the transport mode and tunnel mode functionality. It uses transport mode format but tunnel mode semantics [21].

The difference between the BEET and the other two modes is in the IP header processing. The BEET mode introduces two different address concepts: "inner" address and "outer" address. The "inner" address of the packet is a HIT and is visible to transport and application layer protocols. The outer address is as locator, i.e., a routable IP address, and defines the topological location in a network. Figure 2.7 illustrates the difference between packet structures among three modes. In summary:

- In the transport mode, the IP header is kept intact.
- In the tunnel mode, the packet has two headers on the wire.
- In the BEET mode, the packet has a single header similarly as in transport mode but processing follows tunnel mode semantics.

The BEET mode spares bandwidth and increases MTU by excluding the inner IP header from the packet.

Figure 2.8: HIP data packet structure

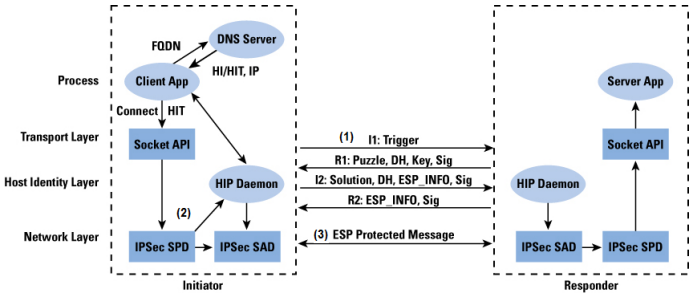

Figure 2.9: HIP with IPSec
[2]

2.4.2 Internet Protocol Security (IPSec) with HIP

ESP (Encapsulating Security Payload) protects application-layer data traffic after the base exchange is implemented. HIP supports BEET (Bound End-to-End Tunnel) mode. The BEET mode uses HITs as the inner addresses and routable IP addresses as the outer addresses. Figure 2.8 shows packet structure for BEET.

Figure 2.9 shows how IPSec interacts with HIP. To start communicating using HIP, two hosts first establish a HIP association using a base exchange (step 1). After the base exchange is successfully completed, both hosts create a pair of IPSec ESP SAs, one for each direction. HIP uses IPSec to provide data encryption and integrity protection for network applications.

For outgoing traffic, applications use HITs as source and destination addresses instead of IP addresses. IPSec layer intercepts the packet, translates the HITs into SPI numbers and encrypts the packets. IPSec reconstructs a new IP header to replace the original one. The new IP header contains the outer source and destination addresses, as defined in the SA (step 2). Finally, the packet is sent to the IPSec tunnel (step 3).

For incoming traffic, the incoming HIT can be found from the SPI number in the ESP packet. IPSec verifies and decrypts the incoming packet based on the SPI in the ESP packet. The original IP header is discarded and IPSec constructs a new IP header which uses HITs as the sender and receiver address (step 2). In this way, client and server applications manages HITs instead of routable addresses, thus leaving mobility management as a task for the HIP layer.

2.5 Sockets API

Sockets API is the most important part in all network communication. This section introduces the sockets API.

In computer networking, a network socket is the endpoint of a communication flow. Sockets form the basis for networking APIs in Unix-based operating systems. The sockets deliver incoming data packets to the recipient application processes or threads and deliver outgoing data. Each socket is mapped by the operating system to a communicating application process or thread based on Transport Layer Interface (TLI). A network socket is bound to a protocol, local socket address, and remote socket address structure.

A socket address structure contains the IP address and transport layer port number. IPv4 addresses are encapsulated in `sockaddr_in` structures with `AF_INET` family while IPv6 addresses are stored in `sockaddr_in6` structures with the `AF_INET6` family. Figure 2.10 illustrates the structures.

```
/* IPv4 socket address structure for 4.4BSD based systems */
struct sockaddr_in {
   uint8_t          sin_len;     /* length of structure (16) */
   sa_family_t      sin_family;
   in_port_t        sin_port;
   struct in_addr   sin_addr;
}

/* IPv6 socket address structure for 4.4BSD based systems */
struct sockaddr_in6 {
    uint8_t         sin6_len;      /* length of this struct */
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* transport layer port # */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* set of interfaces for scope */
};
```

Figure 2.10: The socket address structures used for passing IPv4 and IPv6 addresses to the sockets functions in 4.4BSD format.

A socket provides a communication point between the application and networking stack. A socket is categorized according to three different types. Datagram sockets are known as connectionless sockets which are datagram oriented, connectionless and unreliable. Stream sockets are used for creating sequenced, reliable, two-way, connection-based byte streams. In practice, a stream socket refers to Transport Control Protocol (TCP) and a datagram socket refers to User Datagram Protocol

(UDP) based communication. The third socket type is a raw socket which receives also network packet headers, while non-raw sockets strip the header and receive only the payload. For outgoing packets, the IP header can be usually modified by using raw sockets.

In this section, we focus on discussion of raw sockets because they are essential in the HIP Proxy implementation. It should be noted that raw sockets require super-user privileges.

Socket Creation

A raw socket allows access to the network and transport layer headers. To create a socket of type `SOCK_RAW` , we use the `socket()` function call. It takes three arguments. The first argument defines the address family such as `PF_INET` for IPv4 or `PF_INET6` for IPv6. The second argument sets the type of socket. Value `SOCK_RAW` refers to a raw socket. The third argument sets the protocol number and it defines the protocol value in the IP header (HIP is 139, for example). Prototype of the socket function is shown in Figure 2.11.

```
int socket(int domain, int type, int protocol);
```

Figure 2.11: The `socket` function

A raw socket allows the application to specify the whole IP header by using the `IP_HDRINCL` socket option. If `IP_HDRINCL` option is set, the whole IP packet including payload and IP header can be accessed and modified by the application. Only IPv4 packets can use the `IP_HDRINCL` option. It is not allowed to modify the source address field in IPv6 header.

Socket Options

Socket options can be queried and set using the function calls shown in Figure 2.12. The first argument is the socket descriptor. The second argument defines the socket handler for which the socket option is targeted. The third argument is the name of the option such as `IP_HDRINCL`. The fourth argument is a pointer to the socket option parameter. The last argument indicates the length of the parameter. In Figure 2.13, the function call `setsockopt()` can be used to set the `IP_HDRINCL` options associated with a socket. If the application should manipulate packet headers, the fourth parameter should be set to one.

```
int setsockopt(int s, int level, int optname, const void* optval,
                                socklen_t optlen);
int getsockopt(int s, int level, int optname, void* optval,
                                socklen_t* optlen);
```

Figure 2.12: The Socket option function

```
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

Figure 2.13: The Socket option function

Socket Binding

Bind() function is illustrated in Figure 2.14. The `bind()` system call assigns a local address to a socket. However, a connection-oriented application can call the `bind()` function only once for any given socket. The socket descriptor must be deleted when it is no longer needed.

```
int bind(int sockfd, struct sockaddr* my_addr, socklen_t addrlen);
```

Figure 2.14: The `bind` function

Packet Handling

For outgoing packets, the kernel uses the `sendto()` function call as shown in Figure 2.15 to deliver transport layer packets using a raw socket. It should be noted that the application must calculate the header checksum of the transport layer packet by itself before sending it when the specific transport protocol employs checksums.

For incoming packets, the kernel passes packets to a raw socket only if the protocol and destination of the incoming packet matches the local IP address and protocol of the raw socket. If the raw socket is bound to `INADDR_ANY` for IPv4 or `IN6ADDR_ANY_INIT` for IPv6, the kernel passes a copy of all incoming IP packets associated with the corresponding protocol to the raw socket.

2.6 IPQ Library

Packets are usually handled inside kernel space in Linux [3]. The kernel is ultimately responsible for the sending and receiving of packets. However, the implementation of

```
size_t sendto(int sockfd, const void* buff, size_t bytes, int flags,
                const struct sockaddr* to, socklen_t* addrlen);
```

Figure 2.15: The `sendto` function

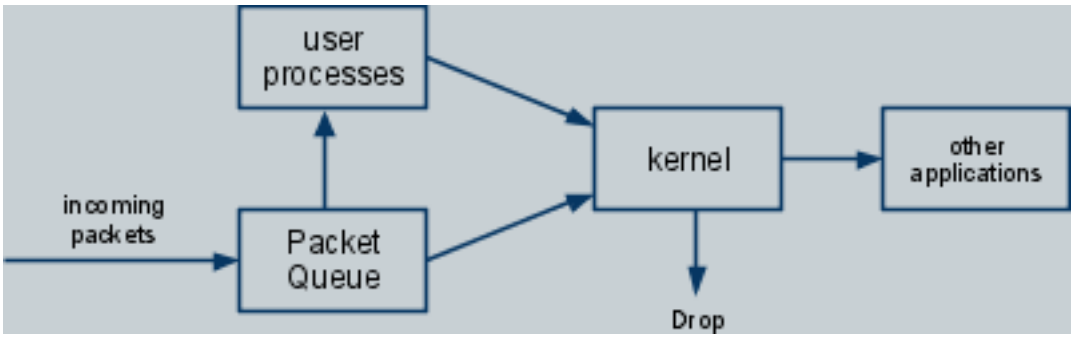

Figure 2.16: Libipq Process

HIP Proxy requires packet handling inside the user space so that the proxy can run without the modification to the kernel. This reduces development time and eases deployment. Iptables userspace packet queuing library (Libipq) redirects traffic from the kernel to userland and was used for the development of the proxy.

Libipq is a library for iptables userspace packet queuing. It provides mechanisms for intercepting packets to userspace. The application can assign verdicts to drop or accept the intercepted packets [8]. Figure 2.16 illustrates the libipq packet handling mechanism.

For each supported protocol, a kernel module called the queue handler is used to perform the mechanics of passing packets to and from userspace. Once the kernel loads the module, IP packets are selected using iptables and queued for userspace processing via the QUEUE target. For example, running the following commands on the command line enables queuing of ICMP packets:

```
iptables -A OUTPUT -p icmp -j QUEUE
ip6tables -A OUTPUT -p icmp -j QUEUE
```

Figure 2.17: The `iptables` function used by Libipq in IPv4 and IPv6

In Figure 2.17, kernel broadcasts all ICMP messages to all listening applications. If no userspace application is waiting, the packets will be dropped.

Libipq provides an API to communicate with kernel modules. It supports the following operations [8]:

1. Initialization.
   The function `ipq_create_handle` creates a Netlink socket [28] and returns an opaque context handle.

2. Setting the Queue mode.
   Libipq specifies the type of userspace packet data by using Function call `ipq_set_mode`. The queued packets has two modes: metadata(`IPQ_COPY_META`) and payload(`IPQ_COPY_PACKET`).

3. Receiving packets from the Queue.
   When messages arrive, function `ipq_read` returns the value `IPQM_PACKET` if it is a network packet. The type of packet is determinated with function `ipq_message_type`. Function `iqp_get_apcket` is responsible for retrieving the metadata and optional payload.

4. Issuing verdicts on packets.
   Function `ipq_set_verdict` issues a verdict on a packet before the packet returns to the kernel.

5. Cleaning up.
   Function `ipq_destroy_handle` frees the resources associated with the context handle.

Chapter 3

Design

In this chapter, we describe the architecture of the HIP proxy.

3.1 HIP Proxy

A HIP-capable host requires that the IP-stack of the end-host supports HIP. From a deployment perspective, this can be an obstacle and some proprietary legacy stacks may never be HIP capable. A HIP proxy that translates non-HIP traffic to HIP-based traffic on behalf of a client host can facilitate HIP deployment.

The proxy uses different mechanisms to process the packets originating from the client vs. from the server, we chose to organize the discussion according to the packet processing directions at the proxy. At the proxy, the outbound direction refers to the direction from the client to the server while the inbound direction indicates the direction from the server to the client.

3.1.1 Databases in HIP Proxy

In our design, a HIP Proxy acts an on-path middlebox (e.g. router or WLAN access point) between HIP-capable servers and legacy clients. All network traffic between them traverses HIP Proxy. We have defined a number of data structures to translate the traffic for the proxy. We introduce two new data structures: Proxy Database (proxydb) and Connection Database(conndb), which are shown in Figure 3.1.

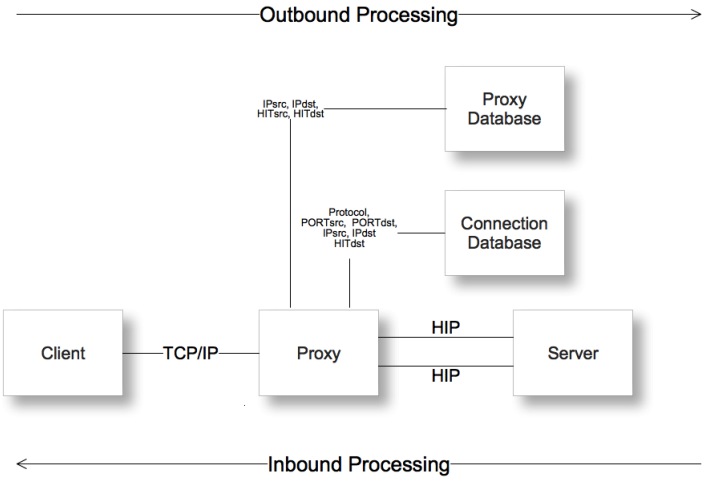

Figure 3.1: HIP Proxy

The proxy handles the outbound packets according to the records from both proxy and connection database while the proxy determines the inbound packet by consulting connection database only. We discuss the topic in detail in the following sections.

Proxy Database

Proxy database stores information related to the translated packet flows. Every time a packet arrives from a client, the HIP Proxy checks the proxy database to find a database entry by searching the IP address of the server and the client. If the destination IP address of the packets does not exist in the proxy database, the HIP Proxy tries a base exchange with the server and updates the proxy database accordingly. If the peer is identified as a HIP incapable host, the proxy records this in the proxy database and passes consequent traffic as it is without HIP transformation. Otherwise, the proxy records a successful base exchange also to the database and applies ESP transformation to the traffic.

Connection Database

As a HIP proxy acts as an on-path middlebox, all network traffic between the client and server traverses through it. The proxy maps IP addresses to the HIT of the server using proxy database. However, this is not enough. Different clients could connect to the same server. Thus, it is impossible to find the connection to clients from the proxy database just with proxy's and server's IP address. Connection database contains the address, protocol and port information of the client, server

and proxy. For outgoing packets, the proxy identifies the packet flow according to the connection database. The proxy updates the connection database together with the proxy database when a new packet flow is established.

3.1.2 Four scenarios for HIP Proxy Design

In this section, we discuss four different usage scenarios and explain how the proxy works in practice.

A HIP-incapable Client and A HIP-capable Server

HIP proxy packet processing from a HIP-incapable host to a HIP-capable host is illustrated in Figure 3.2 and described below.

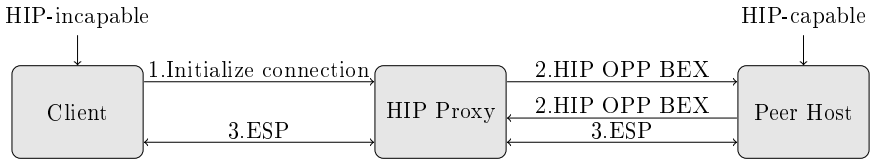

Figure 3.2: A HIP-incapable Host to A HIP-capable host

1. The client tries to establish a TCP or UDP connection to the server.
2. The Proxy captures the first network packet from the client and drops it, thus relying on client retransmissions. The proxy adds the connection to its database and initializes a HIP BEX with the server.
3. After the HIP BEX is completed, the proxy updates its connection database. The proxy starts to translate data packet flows.

A HIP-capable Host and A HIP-incapable host

Proxy packet processing from a HIP-capable host to a HIP-incapable host is illustrated in Figure 3.3 and described below.

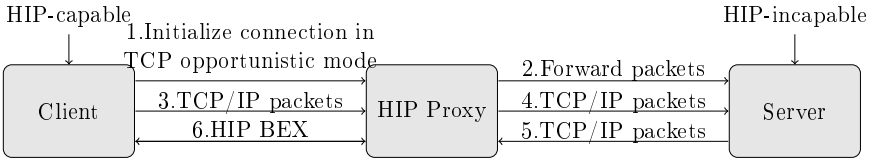

Figure 3.3: A HIP-capable Host to A HIP-incapable host

1. The client tries to establish a HIP BEX to the server in TCP opportunistic mode.

2. The proxy forwards the I1 packet coming from the client host.
3. As the peer host is not HIP capable, the client and proxy will never receive an R1 packet. The client times out and continues transfer without HIP as described in more detail in the thesis of Finez[15].
4. The proxy forwards the TCP/IP packets as they are without translation.
5. The TCP SYN reply from the server triggers the HIP BEX between the proxy and the client.
6. After HIP BEX, the proxy sets up a secure tunnel to the server. All data from/to the client host goes through it.

A HIP-capable Host and A HIP-capable host

The HIP Proxy packet processing from a HIP-capable Host to a HIP-capable host is illustrated in Figure 3.4 and described below.

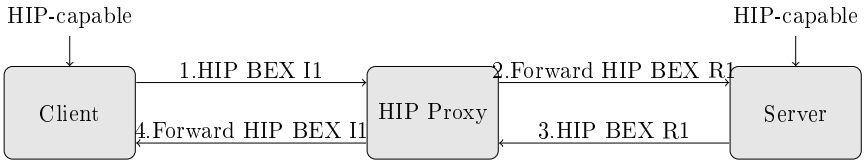

Figure 3.4: A HIP-capable Host to A HIP-capable host

1. The client tries to establish a HIP BEX to the server.
2. The HIP Proxy forwards the I1 packet originating from the client.
3. The proxy receives the R1 response from the server.
4. The proxy forwards the R1 packet to the client. Then, rest of the BEX is completed. The data transfer starts in IPSec mode and the proxy forwards the IPSec traffic.

A HIP-incapable Host and A HIP-incapable host

Proxy packet processing from a HIP-incapable Host to a HIP-incapable host is illustrated in Figure 3.5 and described below.

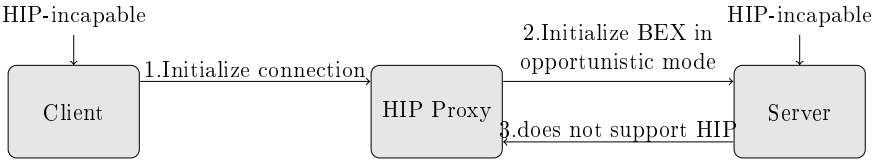

Figure 3.5: A HIP-incapable Host to A HIP-incapable host

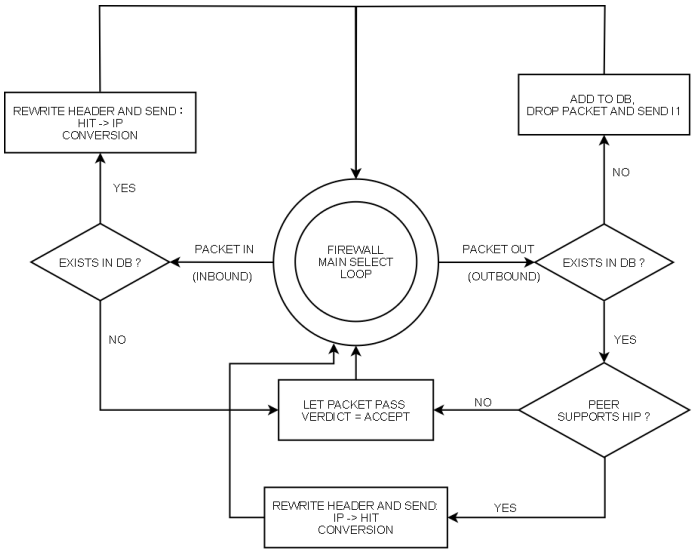

Figure 3.6: HIP Firewall State Machine

1. The client tries to establish a normal TCP or UDP connection to the server.
2. The proxy drops the first network packet coming from the client host and initializes a HIP BEX with the server in opportunistic mode.
3. As the server is not HIP capable, the client and proxy will never complete the HIP BEX. After a time-out, the proxy marks the server as a HIP-incapable host.
4. The client host re-transmits the first packet and the proxy forwards all the network traffic from the client host as it is without modifications.

3.2 Architecture

3.2.1 HIP Firewall State Machine

The HIP firewall (`hipfw`) handles input and output packets processing for HIP Proxy in the HIPL implementation. Capturing the necessary network packets and applying further processing to the packets are essential functionalities for the HIP firewall. It should be noted that `hipfw` also handles HI-based access control in HIPL.

Figure 3.6 illustrates the mechanism of HIP Proxy state machine that is implemented within `hipfw`, which is referred here as HIP firewall state machine (`FW_SM`). The state machine summarizes HIP Proxy packets processing sequences for the incoming and outgoing packets.

For most outbound cases, a legacy client sends the first packet to a server which is routed through the proxy in the beginning. To activate proxy processing, the packet

should be a regular TCP or UDP packet. When the firewall is operating in the HIP proxy mode, it intercepts all packets (not just HIP and ESP packets, with or without UDP encapsulation). As soon as the firewall captures a plain TCP / UDP packet, it consults its proxy database and notices that there is no state because it is a completely new connection. The firewall adds a new database entry for the `{src_ip, dst_ip}` pair and drops the packet. It should be noted that the packet could also be cached for better efficiency, but we have assumed on transport layer retransmissions in this first version of the prototype.

Then, the firewall requests the `hipd` to start an opportunistic base exchange with the Responder. The `hipd` consults also hosts files, DNS and DHT to make a normal base exchange when applicable. The firewall requests the `hipd` to automatically add a new database entry for an opportunistic HIP base exchange and to trigger sending of I1.

When the server does not support HIP (i.e. server does not reply with a R1), the opportunistic mode in `hipd` timeouts and the `hipd` sends a message to the firewall without the HIT of the peer. When the firewall receives such a message, it sets the state of corresponding proxy state to `"HIP_PROXY_PASSTHROUGH"`. Such a state means that the firewall should not block connections between such `{initiator_ip, responder_ip}` pair because the peer is not HIP capable. The firewall assigns verdict "accept" to such a packet/flow. This is illustrated in the state machine in Figure 3.6 in a box that says 'peer supports HIP?".

For inbound traffic, packet processing starts from the firewall intercepting a HIT-based data packet. The firewall consults its proxy and connection database and finds out that there is an entry in TRANSLATE state. The proxy drops the original packet and translates HITs to IP addresses. When conversion occurs from HIT to IPv4, the proxy builds the whole header from scratch. Finally, the firewall injects the transformed packet back to the network stack and the stack delivers the packet back to the legacy client.

3.2.2 Outbound Processing Model

The outbound processing model describes the interaction between the legacy client and the proxy. The outbound packet processing is illustrated in Figure 3.7 and described below.

1. The HIP Proxy is switched on with `hipconf hipproxy on` command.
2. The user runs a client application and the application queries the resolver in order to translate the hostname.
3. The resolver queries the DNS for the given hostname.
4. The resolver returns the IP address assigned to this hostname.
5. The application begins a transport connection by calling e.g. `connect()`.

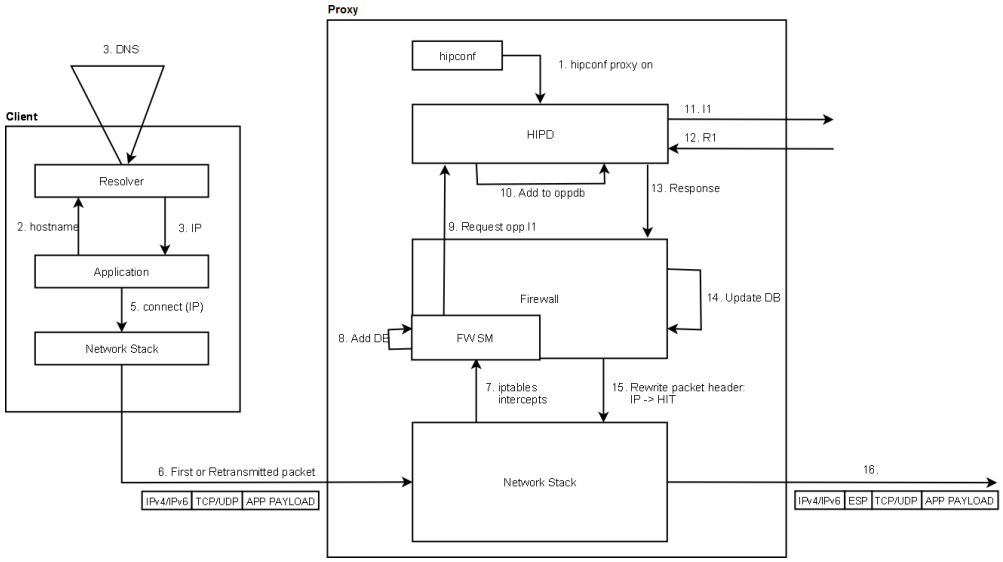

Figure 3.7: HIP Proxy Outbound Scenario

6. The network stack on the client host transmits the TCP or UDP packet to the proxy.

7. The iptables rule in firewall captures the packet and delivers a copy to the `FW_SM` module.

8. The firewall consults its proxy database and notices that there is no entry because it is a completely new connection. The firewall adds a new database entry and drops the packet.

9. The `FW_SM` request `hipd` to send an I1 packet in opportunistic mode.

10. The `hipd` adds a new database entry for an opportunistic HIP base exchange.

11. The `hipd` sends an I1 packet to the peer.

12. The `hipd` receives an R1 from the peer.

13. The `hipd` sends a "HIP BEX successful" response to the firewall

14. The firewall updates the packet flow state in the database accordingly.

15. The firewall rewrites the packet header with HITs and re-injects it back into the network stack.

16. The IPSec module translates HITs to routable IP addresses together with the SPI number, handles ESP encapsulation and transmits the packet on the wire.

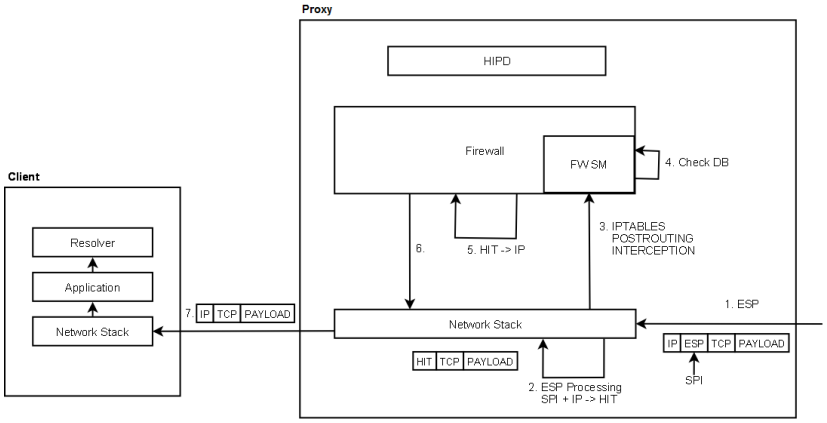

Figure 3.8: HIP Proxy Inbound Scenario

3.2.3 Inbound Processing Model

The inbound packet processing is illustrated in Figure 3.8 and described below.

1. The proxy receives an ESP packet from the server.
2. The IPSec module decrypts the packet and converts the IP header to an IPv6 header containing the HITs. After this, the proxy re-injects it back to the network stack.
3. The iptables rule captures the packet and delivers it to `FW_SM`.
4. The firewall searches its proxy and connection database and discovers that the client IP address from a database entry to be in TRANSLATE state.
5. The firewall rewrites the packet header with plain IP addresses.
6. The firewall re-injects the packet into the network stack.
7. The network stack transmits the packet on wire.
8. The packet arrives to the stack and the stack delivers the packet to the transport and application layer.

3.3 HIP proxy and hipconf extension

The hipconf extension for the proxy used as follows:

```
tools/hipconf hipproxy <on|off>
```

Figure 3.9: The hipconf command for switching proxy on/off

The proxy usage instruction are stored in `hipd`. The `hipfw` queries this information from `hipd` when it is started. As the communication between `hipfw` and `hipd` is message based, `hipfw` sends an internal message to `hipd`. Then `hipd` replies and indicates whether the proxy is in use or not.

Chapter 4

Implementation

HIPL consist five different components: HIP daemon (`hipd`), HIP configuration tool (`hipconf`), HIP socket handler, BEET for IPSec and HIP firewall (`hipfw`). We modifies the `hipd`, `hipconf` and `hipfw` components for the HIP Proxy implementation. In this chapter, we focus on the implementation of those three modified components in HIPL. We present the overall implementation architecture and the interaction between the different software components using sequence diagrams.

4.1 HIP Proxy

To use the proxy efficiently, the proxy is designed in a way that proxy clients can communicate with the proxy server simultaneously. An ESP tunnel between a proxy and a server may be shared between multiple clients. The proxy masquerades the clients in such way that all client connections appear to originate from the same IP address of the proxy, which is illustrated in Figure 4.1. In this section, we describe the changes to the existing implementation architecture and describe how they are used for the proxy functionality.

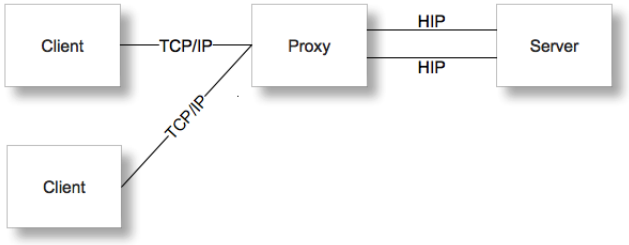

Figure 4.1: HIP Proxy

4.1.1 Data Structure in HIP Proxy

We introduce a new data structure for an internal database of the proxy in HIP Proxy. As shown in Figure 4.2, `hip_conn_key` is based on the IP address and port number of the client and the server.

```
struct hip_conn_key {
        uint8_t protocol;
        uint16_t port_client;
        uint16_t port_peer;
        struct in6_addr hit_peer;
        struct in6_addr hit_proxy;
}  __attribute__ ((packed));
```

Figure 4.2: HIP Connection Key structure

This structure collects the all necessary information for each connection in the Proxy Connection Database which will be discussed in detail in the following sections. We use this information to identify the different connections from each other in the outbound processing.

4.1.2 Databases in HIP Proxy

HIP Proxy introduces two databases that store connection information for outbound and inbound connections. Both databases also record relationship between clients and servers.

HIP proxy uses the database to store connection information for outbound and inbound connections. Each entry in the proxy database contains the mapping from

the client IP address to the HIT of the server. If multiple clients shares the same tunnel to the server, their packets to the server look same. The proxy cannot find the client information according to the information of inbound packets. Thus, a "connection database" is introduced. It records the IP address of the client and the server, as well as the port numbers, so that the proxy can use this information to identify the inbound packet flows.

Proxy Database

The HIP Proxy Database (`proxydb`) contains outbound connection related information as shown in Figure 4.3. The proxydb contains the IP addresses and HITs of the proxy itself, the server and also the client IP address. Variable `hip_capable` indicates if the current connection uses HIP between the client and the server. The variable `state` defines the packet processing state. It has three different states: `HIP_PROXY_I1_SENT`, `HIP_PROXY_PASSTHROUGH`, and `HIP_PROXY_TRANSLATE`. When the proxy triggers a HIP base exchange, the state transitions to `HIP_PROXY_I1_SENT`. Once the proxy establishes a HIP association with the server, the proxy transitions the association state to `HIP_PROXY_TRANSLATE`. If the proxy discovered the server to be HIP-incapable, the proxy transitions the state to `HIP_PROXY_PASSTHROUGH` and the proxy passes the traffic without any modifications. The proxy discovers this through a timeout in the opportunistic HIP based exchange[15].

```
typedef struct hip_proxy_t {
        hip_hit_t hit_proxy;
        hip_hit_t hit_peer;
        struct in6_addr addr_client; // addr_proxy_client
        struct in6_addr addr_peer; // addr_proxy_server
        struct in6_addr addr_proxy; // addr_proxy
        int state;
        int hip_capable;
} hip_proxy_t;
```

Figure 4.3: HIP Proxy outbound connection Database

Connection Database

In addition to HIP Proxy Database, the proxy includes a second database called HIP Connection Database (`conndb`). Its objective is to record the connection related information associated with inbound connections.

```
typedef struct hip_conn_t  {
        struct hip_conn_key key;
        int state;
        struct in6_addr addr_client; // addr_proxy_client
        struct in6_addr addr_peer; // addr_proxy_peer
} hip_conn_t;
```

Figure 4.4: HIP Proxy inbound connection Database

4.2 HIP Firewall State Machine

As described in the previous section, the HIP firewall state machine ties together the components of the HIP Proxy. The `hipfw` initializes the necessary components and inserts rules for network traffic capture. The HIP firewall state machine uses `LIB_IPQ` interface of netfilter to receive the intercepted packets. After analyzing and processing, `hipfw` delivers the verdict for each packet.

4.2.1 Packet Processing

To enable the HIP Proxy functionality, `hipfw` captures the network datagrams with the different firewall rules for inbound packets and outbound packets. For outbound connections, `hipfw` captures all the forwarded packets. By default, it allows all HIP packets to traverse to the server because the proxy will be transparent for HIP-capable hosts. This can be restricted using the ACL functionality in the `hipfw`. The proxy acts on outbound traffic only when a client sends non-HIP traffic. The iptables rules for capturing outbound traffic are as follows:

```
iptables -I HIPFW-FORWARD -p tcp -j QUEUE
iptables -I HIPFW-FORWARD -p udp -j QUEUE
```

For inbound connections, i.e., traffic from a server to the proxy, the proxy captures traffic from the ESP tunnel. The proxy is the destination of the actual ESP tunnel. The following firewall rules capture inbound traffic:

```
ip6tables -I HIPFW-INPUT -p tcp -d 2001:0010::/28 -j QUEUE
ip6tables -I HIPFW-INPUT -p udp -d 2001:0010::/28 -j QUEUE
```

Basically, the proxy should support all transport protocols above the IP layer. However, the proxy implementation is based on Linux operating system. And it does

not allow the modification of the source address from a userspace process for IPv6. Thus, the proxy only translates IPv4 traffic from clients.

4.2.2 Interaction between the Components

Interaction between proxy components is illustrated using sequence diagrams in this section. The sequence diagrams show the control flow through different function calls. We do not show the full execution trace of functions but instead focus on the most relevant functions.

HIP Firewall State Machine (HFSM) is a conceptual state machine model for the proxy. It implements processing logic of the proxy. Figure 4.5 shows how the HFSM handles outbound connections. Initially, the outbound packets are sent to HFSM according to the firewall rules using function call `hip_fw_handle_other_forward()`. HFSM calls function `request_hipproxy_status()` to check the status of the proxy from `hipd`. If the proxy is turned on, the packets are delivered to the outbound component for the further processing. In the end of the flow, the processed packets are sent to the HFSM which is responsible for re-injecting packets back to the network stacks for network delivery.

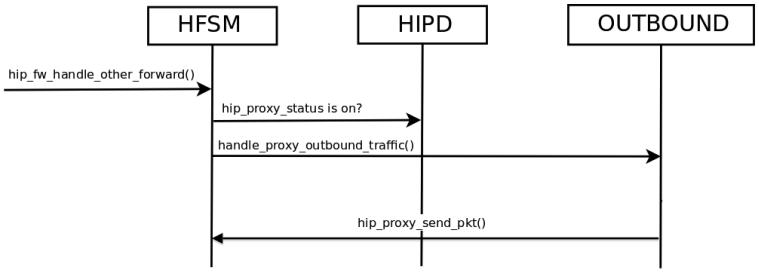

Figure 4.5: HFSM with an outbound connection

The proxy handles inbound connections similarly to outbound connections. An inbound interaction scenario is depicted in Figure 4.6 .

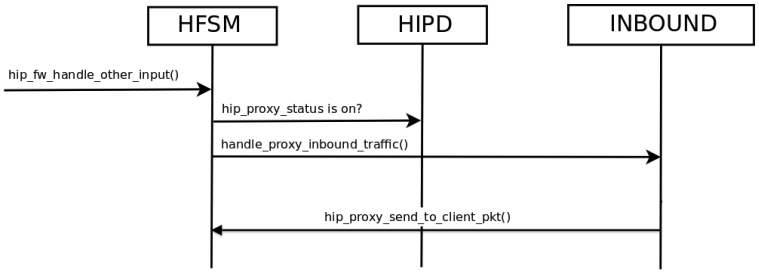

Figure 4.6: HFSM with an inbound connection

4.3 Outbound Processing

This section summarizes the overall functionality of the system. It also describes interactions within the outbound component which is illustrated in Figure 4.7. The outbound component handles connections originating from the clients. When a packet from a client traverses through the network stack to the server, the `HFSM` captures it and sends it to outbound components using function call `hip_fw_handle_other_forward()` (in step 1).

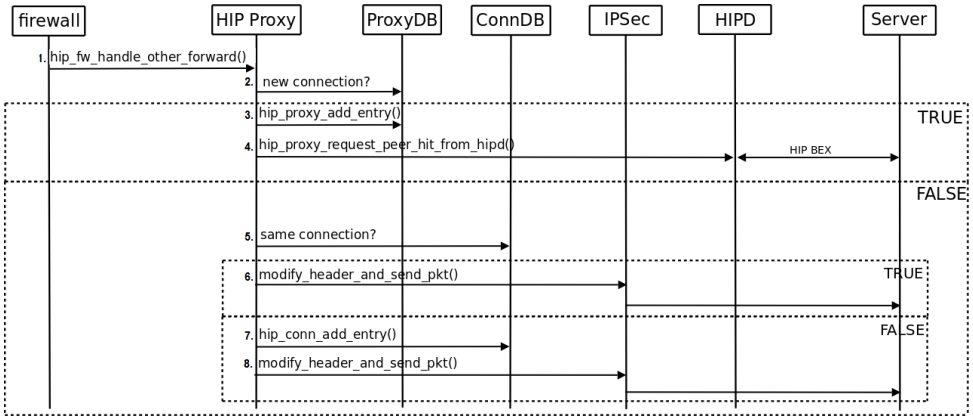

Figure 4.7: Outbound Model sequence diagram

4.3.1 Connection Establishment

The proxy checks the state of the connection in `proxydb` (in step 2). If the proxy does not find the state information in database, it adds a new entry into `proxydb` to cache the connection information (in step 3). It also issues a call to `hip_proxy_request_peer_hit_from_hipd()` to request `hipd` for the server's HIT (in step 4). If the proxy has not established a prior HIP connection with the server, the proxy triggers a base exchange (in step 4). The proxy transitions the state in database entry to `HIP_PROXY_I1_SENT` and discards all the packets to the server until `hipd` sends a confirmation. If the server is HIP-capable, the proxy receives a notification of successful base exchange and updates the state to `HIP_PROXY_TRANSLATE`. Otherwise, the connection in the `proxydb` is marked as `HIP_PROXY_PASSTHROUGH`. If the same connection is found in `proxydb`(in step 5), the proxy modifies the packet header and sends it into ESP tunnel (in step 6). If not, the proxy adds a database entry to `conndb` (in step 7). Finally, the clients can proceed with communication with HIP capable servers through the proxy (in step 8).

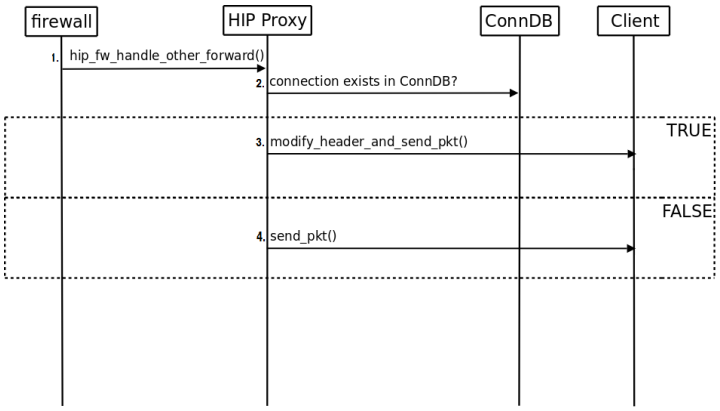

Figure 4.8: Inbound Model sequence diagram

4.3.2 Packet Processing

Once state is successfully established, the proxy modifies the header of an outbound IP packet and re-inject it back to the network stack. The re-injection process consists of replacing the IPv4 header with an IPv6 header where the source and destination address correspond to the proxy's HIT and the server's HIT. The function call `hip_proxy_send_pkt()` delivers the IPv6 packet using a raw socket, which injects the packet into the ESP tunnel.

4.4 Inbound Processing

The inbound direction refers to packets originating from the server. All connections are initialized by clients and, therefore, the inbound processing occurs after the proxy has established state with the server. Figure 4.8 illustrates the sequence of the most relevant function calls for the inbound processing.

4.4.1 Packet Processing

The destination of all inbound packets is the proxy. The proxy identifies the connections by checking the `conndb` (in step 2). If the proxy finds the record in the database, it replaces the IPv6 header with HIT with new IPv4 header and re-injects the packet back to the networking stack with the new destination set to the client's IP address contained in the `conndb` (in step 3). The proxy fills the source and destination fields by the server's IP address and the client's IP address and sends the translated packet using the function call `hip_proxy_send_client_pkt()` (in step 3). If proxy does not find the state in `conndb`, the proxy forwards the packet (in step 4).

4.5 Protocol Translation

The HIP proxy handles the traffic flow translation between IPv4 based legacy clients and HIP capable servers. A HIP-capable server uses IPv6-networking (i.e. HITs) as the application layer identifier. Thus, the proxy provides a basic address translation mechanism between IPv4 and IPv6.

The implementation uses a raw socket to inject packets because it can manipulate the packet header, as opposed to standard sockets which can alter just the packet payload but no headers. In outbound processing, the proxy builds the new IPv6 packet in which the source address belongs to the proxy and the destination address belongs to the server. It also copies the payload into the newly constructed IP packet and recalculate the checksum for the transport layer header. In inbound processing, the proxy translates IPv6 into IPv4 by using the same, but reverse, mechanism used in the outbound processing.

Chapter 5

Analysis

This chapter first presents the test environment of the HIP proxy performance evaluation. Secondly, we show the performance measurement results organized in charts and tables. The analysis also includes related problems we discovered during the testing and analysis phases.

5.1 Testing Environment for Performance Evaluation

In this section, we introduce the testing environment of the performance evaluation for the HIP proxy. Furthermore, we present the test platform and the software used for the experimentation.

5.1.1 Test Platforms

Hardware Environment

We performed the measurements on three hosts: a client, a client-side proxy and a server. The proxy acts as the initiator and the server as the responder. We built the test bed using virtualization on one laptop pc. The client host and the server host are based on VMware player v2.5.2. Proxy was being run in the underlying PC without virtualization. Table 5.1 below describes the hardware and operating system characteristics of the hosts used in the tests. It should be noted that the hardware did not have any native support to accelerate virtualization.

Client	Proxy	Server
Ubuntu Jaunty 8.04 Kernel 2.6.24-1 with BEET Patch	Ubuntu Jaunty 9.04 Kernel 2.6.30-1 with BEET Patch	Ubuntu Jaunty 8.04 Kernel 2.6.24-1 with BEET Patch
256M Memory	2G Memory	256M Memory
VMware	Intel(R) Core(TM) 2 CPU	VMware
1000M Ethernet	1000M Ethernet	1000M Ethernet

Table 5.1: Test Configuration

Network Configuration

We connected all the hosts using a virtual 100Mbit Ethernet link and all connections are based on IPv4. The MTU default value used in the performance evaluation is 1280 bytes. The network topology is shown in Figure 5.1.

Figure 5.1: Network Configuration for Performance Evaluation

The proxy is equipped with two network interface cards which bridge the connections between two different network domains. The legacy client is located in the 192.168.74.00/24 in which connections are based on plain TCP/IP protocol. The HIP-capable server is located in 10.0.0.0/24 network only. The connectivity between the proxy and server are based on HIP.

5.1.2 Test Software and Configuration

We used `bzr` revision trunk/1855 of HIPL software on the client, the server and the proxy for performance testing. The test software is Iperf version 2.0.2. Iperf shows the bandwidth and throughput by streaming for 20 seconds. For UDP connection, the bandwidth for testing was set as 1000Mbits.

5.1.3 Test Procedure

We use the following test procedure for our test. The proxy acts as the Initiator and the server host as the Responder, where we repeat the test 20 times. The tests measure the throughput of TCP on top of the ESP tunnel. We also compare the performance to ESP-based end-to-end data traffic without the HIP proxy. It should be note that base exchange delay was included in the measurements.

5.2 Results and Analysis of Performance Measurements

In this section, we illustrate the results obtained using the test configuration which we described in the previous section. This section focuses on analysis of the TCP throughput with and without the proxy.

5.2.1 TCP Throughput

We tested data traffic both with end-to-end HIP connections and end-to-middle using the HIP proxy. Figure 5.2 shows the TCP throughput as well as the standard deviation.

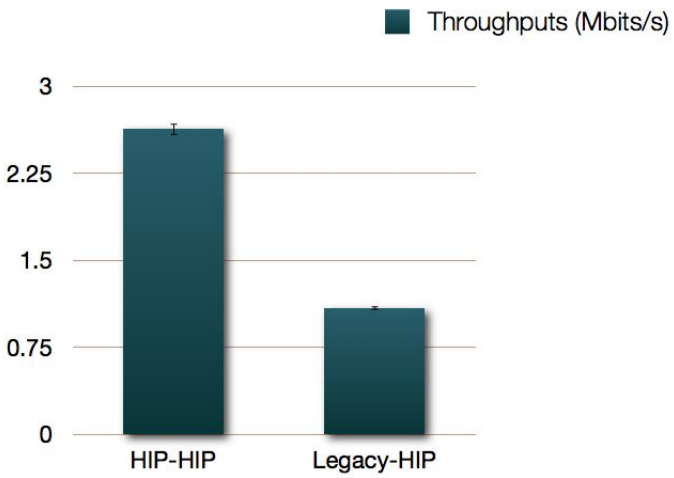

Figure 5.2: Comparison of Throughput for TCP connections using HIP

The chart shows that the average throughput for a TCP connection via the HIP proxy is 1.09Mbit/s with a standard deviation value of 0.19Mbits/s. A HIP end-to-end connection has an average value of 2.63Mbits/s with a standard deviation of 0.32Mbits/s. As the measurement shows, the proxy prototype decreases throughput. There are several reasons for the lowered performance. First, the implementation is running on virtual machines without hardware support for virtualization, i.e. hosts share the same physical resources. Second and more importantly, the implementation is an unoptimized prototype and it has not been profiled for performance bottlenecks. Third, the proxy prototype is implemented on top of `hipfw`. The single threaded implementation of `hipfw` has to process both input and output data flows, and we suspect it could therefore be the main performance bottleneck.

5.3 Issues for ICMP

The Internet Control Message Protocol (ICMP) is not supported by the current HIP proxy implementation. As one of the core protocols of the TCP/IP suite, it is mainly used to send error messages indicating, for instance, that a requested service is not available or that a host or router could not be reached. As shown in Figure 5.3, the ICMP packet consists of a header and the protocol payload. The header contains only three fields: type, code, and checksum. Type specifies the type of the message. The value of the code field depends on the message type and provides an additional level of message granularity. The checksum field provides a minimal level of integrity verification for the ICMP message.

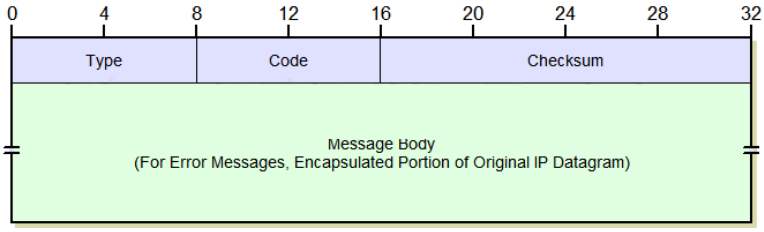

Figure 5.3: The header of ICMP and ICMPv6

We studied also ICMP in our test environment. The current proxy implementation is not yet compatible with the ICMP protocol. The reason is that communication of the legacy client are assumed to be based on IPv4 and the server is based on IPv6. ICMP protocol is recognized as ICMPv4 in IPv4 and ICMPv6 in IPv6. They are different protocols that have different definitions for their fields. Thus, the proxy cannot handle ICMP packets correctly by using the same protocol translation mechanism used for TCP/IP.

One solution for the ICMP issue is to implement a mapping table between ICMPv4 and ICMPv6 inside the proxy. The table contains the field translation definitions for both ICMPv4 and ICMPv6. The proxy uses the table to translate ICMPv4 to ICMPv6 according and vice versa. We did not have time to implement this solution during this thesis.

An alternative solution is to perform IPv4 to LSI translation at the proxy. By using LSI support, the ICMP packets could traverse the proxy without any problem. However, we did not have time to experiment with this solution in this thesis.

5.4 An Issue with FTP

Some application layer protocols include explicit network addresses within their application payload, which creates some problems in NATed environment as well as with the HIP Proxy. Some examples of those applications and protocols are File

Transfer Protocol (FTP), Session Initiation Protocol (SIP) and Simple Network Management Protocol (SNMP). In this section, we focus on the FTP case.

File Transfer Protocol (FTP) is a network protocol used to exchange files over a TCP/IP-based network. FTP is built based on the client-server paradigm and uses two separate TCP connections for communications, one for control and another for data. By default, the FTP server employs TCP port 21 as the control channel which transfers FTP requests and replies during the FTP session. Port 20 is usually used for data, i.e., to transfer actual files.

FTP can run under "active" or "passive" mode, which controls how the data connection is established. The client sends the server the IP address and port number that the client should use for the data connection. In active mode, the server opens the connection according to the information from the client. In passive mode, the client asks the server for the IP address and port number where it can connect by sending a `PASV` command. So, in this mode, the client opens the connection to the server. If the server uses an IPv6 address, the command must be EPSV.

The FTP protocol encapsulates an address and port number into application payload in both modes, which leads to a problem with the HIP proxy. To illustrate the problem, we demonstrate using a practical example with the HIP proxy. The FTP server uses Pure-FTPd version 1.0.21-11 and the client uses lftp version 3.5.11. Below is a log in which the client tries to list the directory at the server using FTP.

```
whu@virtual1:~$ ftp 10.0.0.129
Connected to 10.0.0.129.
220---------- Welcome to Pure-FTPd [privsep] [TLS] ----------
220-You are user number 1 of 50 allowed.
220-Local time is now 01:22. Server port: 21.
220 You will be disconnected after 15 minutes of inactivity.
Name: (10.0.0.129:whu):
331 User whu OK. Password required.
Password:
230-User whu has group access to ftpgroup whu  fuse  admin  scanner
230- plugdev  dip  audio  tape  floppy  cdrom  fax  dialout
230- adm
230 OK. Current directory is /home/whu
Remote system is UNIX.
Using binary mode to transfer files.
ftp> ls
500 I won't open a connection to 192.168.74.129 (only to 2001:1a:9b4
:cc61:f92f:57b4:d289:fd3e)
ftp: bind Address already in use.
ftp> passive
425 You cannot use PASV on IPv6 connections. Use EPSV instead.
Passive mode refused.
```

When the client executes `ls` command, the server refuses the connection. The server informs that it cannot establish the connection to the client (`192.168.74.129`) but only to the address `2001:1a:9b4:cc61:f92f:57b4:d289:fd3e`. This occurs because of the protocol translation mechanism at the HIP Proxy. The client establishes an IPv4 connection to the proxy and the proxy transparently translates the tunneled connection with inner addresses of the tunnel based on HITs.

In FTP active mode, the client specifies the IP address and port number to which the server should connect back by encapsulating this information into the FTP payload. As the connection between the server and the proxy is based on HIP, the server cannot recognize the IPv4 address as the client address.

In FTP passive mode, the client sends `PASV` command to ask the server for the IP address and port number. With IPv6, the client must use `ESPV` command. In the example, the client uses the `passive` command. In this case, the passive mode does not work because the server expects the `ESPV` command since it runs on top of IPv6. Thus, the client receives the error "425".

The analysis above shows that the proxy may not be suitable for all application-layer protocols which encapsulate addresses inside their payload. The reason is that the proxy handles both IPv4 and IPv6 connections. The translation between IPv4 and IPv6 only occurs at the internet and transport layer, which means only the IP

header is translated and not the application payload. A possible solution is that the proxy could check the protocols and rewrite the protocol contents. An LSI-based translation at the application layer is not suitable because of their local scope.

5.5 Compatibility Assurance with Other HIP Extensions

The HIP proxy implementation should remain compatible with other extensions included in the HIPL implementation. In this section, we try to analyze compatibility with some of the extensions.

5.5.1 HIP Firewall

The HIP proxy design is based on packet capture at lower layers. The `hipfw` daemon captures packets using the predefined rules for iptables. The proxy uses a forward rule for outbound packets and an input rule for inbound packets. To test the HIP proxy compatibility with firewall access control functionality in `hipfw`, the following rule is inserted into `etc/hip/firewall_conf`.

```
INPUT -src_hit 2001:001b:444b:6062:4f84:1c0f:148d:e1d6 DROP
```

After setting the rules in `firewall_conf`, data traffic between the client and the server was successfully blocked. And the traffic continues when we change the rule to the following.

```
INPUT -src_hit 2001:001b:444b:6062:4f84:1c0f:148d:e1d6 ACCEPT
```

This means there is no conflict between iptables access control rules for the proxy and the `hipfw`. The `hipfw` access control functionality worked as expected when the proxy was enabled.

5.5.2 NAT Traversal

Network Address Translation (NAT) devices are common deployed to alleviate the exhaustion of IPv4 address space with the use of IP address spaces. The internal network interface of the NAT device communicates with the external network by translating the source address of outgoing requests with that of the NAT device and by relaying the replies back to the originating device. This causes connectivity problems when a host inside the private address realm as a server or otherwise accepts incoming connections. In this section, we will discuss different techniques of NAT traversal for HIP and their relationship with the HIP proxy.

Client-side NAT traversal

HIPL supports connections initiated behind a NAT. The idea is to use UDP tunneling in which the initiator encapsulates HIP control packets and ESP data packets in UDP. In this way, both the initiator and responder have to support NAT extensions in order to traverse a NAT. We tested client-side NAT traversal by using the following command to configure the `hipd` at the initiator (i.e. the proxy):

```
hipconf nat plain-udp
```

As a result, the proxy was able to translate the traffic and was compatible with the client-side NAT traversal. The traffic between the proxy and the server was tunneled on top of UDP and the proxy just translated packets into the ESP tunnel and back. The outern-most UDP tunnel was transparent for the proxy and visible only to the IPSec.

Teredo

Teredo is a NAT traversal solution for HIP. It has been designed to provide IPv6-based access to end-hosts located in IPv4-based private address realms. Teredo protocol detects the type of NAT and can communicate with native IPv6 and Teredo-based hosts. When a Teredo host contacts another host using native IPv6, the contacting host encapsulates IPv6 packets over UDP and IPv4 through a Teredo relay which then decapsulates the IPv6 packets and relays them on the IPv6 Internet. Teredo protocol requires IPv6 support at both client and server side. Thus, the HIP Proxy implementation is not yet compatible with the Teredo protocol because it currently supports only IPv4 traffic originating from the client side. At the moment, this scenario remains unsupported by the implementation due to time limits for this thesis.

Interactive Connectivity Establishment

HIP has an alternative NAT traversal solution which is called Interactive Connectivity Establishment (ICE) [11]. ICE allows two end-hosts located in different private address realms to communicate over HIP with each other. With ICE technique, both client and server can be located behind different NAT boxes and obtain connectivity using HIP and ICE. The ICE-based HIP solution does not require changes in the application software and works also with IPv4-based applications. To use the ICE, a HIP/ESP Relay Server is needed that relays HIP control and possibly also ESP data traffic. A HIP Responder uses the HIP Registration Extension to register their HIT->IP address mappings to the relay. After that, a HIP initiator can initiate a base exchange using the IP address of the relay instead of the IP address of the re-

sponder they attempt to connect. Thus, the responder are reachable with the relay's IP address.

In this master thesis, we did not test the compatibility with ICE solutions because the ICE implementation was not finished during the thesis work. However ICE provides an end-to-end NAT traversal solution between two HIP-capable hosts, which can be the proxy (the initiator) and the server (the responder). Thus, we believe the ICE solution should be compatible with the HIP proxy.

5.5.3 HIP Opportunistic Mode Extensions for TCP

HIP opportunistic mode extensions for TCP are usually used between two HIP-capable hosts. In HIPL, both the proxy and the server are HIP-capable hosts. The main concern is the conflict of the iptables rules with the HIP opportunistic mode extensions for TCP. In proxy implementation, the proxy only has the `FORWARD` rule for IPv4 traffic and the `INPUT` rule for IPv6 traffic. The rules do not affect that the TCP SYN packet processing and the HIP opportunistic mode extensions for TCP are compatible with the HIP proxy.

5.5.4 HIP Userspace IPSec

Userspace IPSec extension is an alternative design for kernel space IPSec. The userspace implementation does not require any changes to the kernel for Linux OS below 2.6.27 version. In HIPL implementation, userspace IPSec relies on receiving HIT-based and non-HIT packets. Figure 5.4 shows the packet processing in outbound scenario.

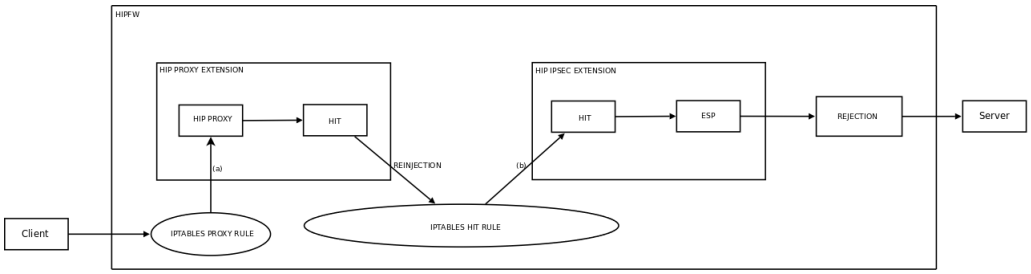

Figure 5.4: The outbound processing with Proxy and userspace IPSec extensions

The proxy captures the outbound packet in step (a) while the userspace IPSec implementation captures packets in step (b). Both of them modify and re-inject the packets back into the network stack in the end of the processing. The main difference is that the proxy only modifies the packet headers while userspace IPSec module encrypts the packets and transfers them using the BEET mode.

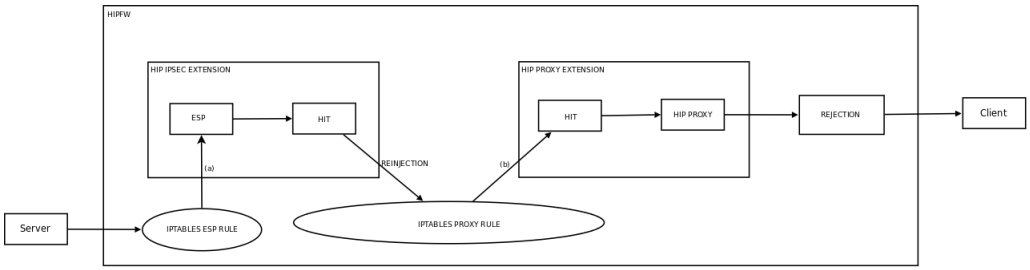

Figure 5.5: The inbound packets processing including Proxy and userspace IPSec extensions

Figure 5.5 shows the packets processing in inbound scenario. The situation is a little bit trickier here. Both the proxy and userspace IPSec are configured with the INPUT iptables rules. In step (a), the userspace IPSec module captures the ESP packets which are encrypted by the server. The proxy module captures the packets which have the HIT as their destination address in step (b). Thus, there is no iptables rule conflict between these two modules. In practice, the firewall uses the userspace IPSec module to decrypt the inbound packet and replaces the HITs into the IPv6 header when the ESP packets arrives. After that, the userspace re-injects packets back to the network stack and then they are captured by the `hipfw` again for further proxy processing.

5.6 Comparison to Ericsson Implementation

Another HIP proxy implementation was done in Ericsson [27]. The implementation is based on 3G networks, which serves legacy UEs, providing them with the advantages of HIP for the part of the connection that goes over the Internet. The implementation architecture is similar with the implementation discussed in this thesis. Both implementations are client side proxies and use the firewall to capture packets. The difference is that the Ericsson implementation is supporting IPv6 on the client side which is not supported in the HIPL implementation; and the Ericsson implementation is developed for FreeBSD while the HIPL implementation is developed for Linux. One fundamental difference also is that our proxy operates in opportunistic mode, i.e., does not depend on HIP-based DNS infrastructure.

Chapter 6

Future Work

The design and implementation efforts with the HIP Proxy have brought up a number of future research and development ideas that are described in this chapter.

6.1 Client side HIP Proxy

This section describes client-side issues with the HIP proxy that need future work.

6.1.1 ICMP Protocols Support

The protocol support in our HIP proxy implementation is quite limited. The ICMPv4 protocol is not supported in the current implementation as mentioned in Chapter 5. The proxy implementation does not yet implement the ICMPv4 support.

6.1.2 Referrals

Some applications use IP addresses as referrals, which means that they pass addresses from one host to another within application-layer protocol. For example, File Transfer Protocol (FTP) applications use referrals so that the FTP connection could not pass through the HIP Proxy on our experiments. A possible solution is that the proxy analyzes the application-layer protocol and translates it. This still requires further work.

6.2 Server side HIP Proxy

Our HIP proxy implementation is designed to act as a client side proxy which only supports the communication initialized by a legacy HIP-incapable client. It limits the usage of the HIP proxy to some cases. In this section, we illustrate an alternative design for a server side HIP proxy.

6.2.1 Architecture and Design

The server side proxy is an alternative approach to support communication between HIP-capable clients and legacy servers. Opportunistic mode should not be used for the server-side HIP proxy because the proxy may be serving multiple legacy servers and may not be able to determine the ultimate destination server. Unlike the client-side HIP proxy, the server side HIP proxy requires HIP DNS support [20] to resolve the server host names into HITs. As the server side proxy has multiple HITs, it cannot figure out which HIT the client is requesting. Thus, Figure 6.1 shows the design for the server side proxy.

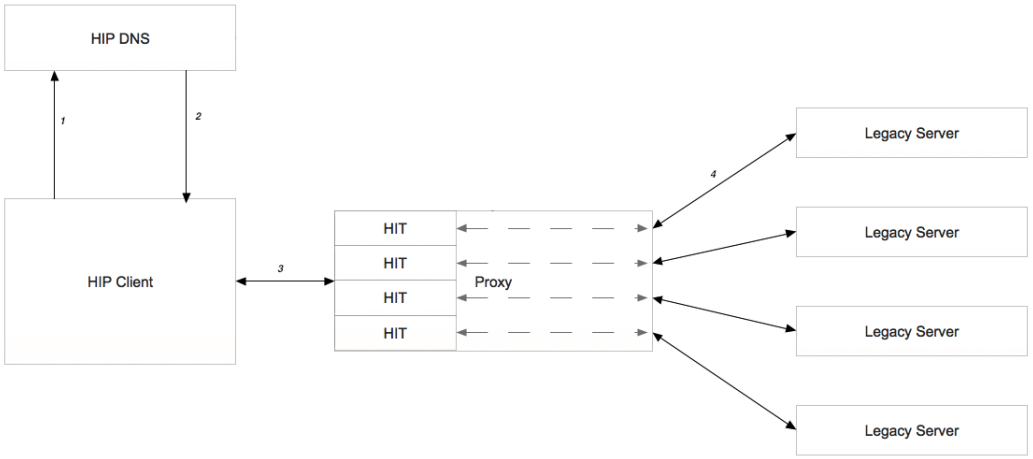

Figure 6.1: HIP Server Side Proxy

In Figure 6.1, the outbound processing concerns the packets originated from the HIP clients. When the HIP client initializes a connection to a legacy server, the client asks the HIT of the legacy server from DNS in step 1. Once the HIP client receives the HIT from the HIP DNS in step 2, the client triggers a base exchange to the HIT in step 3. The proxy checks the HIT of incoming packets and finds the IP address of the legacy server matching to the HIT. Finally, the server side proxy terminates the HIP/ESP tunnel and forwards the packets to the legacy server in step 4.

The inbound processing is much simpler than the outbound processing. In inbound processing, the legacy server sends the plain IP packet to the server side proxy. Then, the server side proxy looks up the database to find the destination HIT and sends the packets to the HIP client over the ESP tunnel.

Chapter 7

Conclusion

Host Identity Protocol (HIP) offers host authentication, mobility support, data security and IP address agility for the existing Internet architecture. However, deploying HIP to Internet wide is a costy and long process. To ease the deployment, we presented a proxy-based approach of using HIP that avoids HIP deployment completely at the client side.

The proof-of-concept prototype introduced in this thesis is based on HIPL implementation. The HIP proxy provides interoperability between legacy HIP-incapable clients and HIP-capable servers. Using the proxy, an unmodified legacy client can establish HIP connectivity with a HIP-capable server through the proxy.

We showed initial performance measurements of the HIP-proxy prototype. The implementation is immature enough and should be profiled for optimal performance. We expect the performance to improve radically when the prototype matures.

We discovered some future work items with the HIP proxy and referrals. The referral problem is related to those protocols that encapsulate the addresses into application-layer payloads. The study of the scenario with FTP shows that the communication does not work when the FTP client is running on the IPv4 client in either passive mode or active mode. As a straw-man solution, we suggested address-translation also at the application layer.

In addition, we reviewed the compatibility with the normal firewall access control and the userspace IPSec extension. Both of them showed to be compatible with the HIP proxy. We also studied the compatibility with UDP tunneling, Teredo and ICE techniques.

Our design and implementation of the HIP proxy can be used to connect legacy client networks to HIP-capable server networks to support incremental deployment of HIP. In a way, the HIP proxy acts a "reverse VPN". To support normal VPN-like connectivity, i.e. between HIP-capable clients and legacy servers, more work is needed. When both client and server side proxy are available, HIP support can be implemented completely at middleboxes.

Bibliography

[1] D. Eastlake 3rd. *RFC 3110: RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS).* Internet Engineering Task Force, 2001. `http://www.ietf.org/rfc/rfc3110.txt`.

[2] A. Gurtov A. Pathak, M. Komu. *Host Identity Protocol for Linux: HIPL gives your Linux box a name*, nov 2009.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, Sebastopol, California, October 2000.

[4] R. Braden. *RFC 1122: Requirements for Internet Hosts - Communication Layers.* Internet Engineering Task Force, 1989. `http://www.ietf.org/rfc/rfc1122.txt`.

[5] B. Carpenter. *RFC 1958 Architectural Principles of the Internet*, jun 1996. `http://www.ietf.org/rfc/rfc1958.txt`.

[6] C. Perkins D. Johnson and J.Arkko. *RFC 3775: Mobility Support in IPv6.* IETF, June 2004. `http://www.ietf.org/rfc/rfc3775.txt`.

[7] D. EastLake. *RFC 2536: DSA KEYs and SIGs in the Domain Name System (DNS).* Internet Engineering Task Force, 1999. `http://www.ietf.org/rfc/rfc2536.txt`.

[8] Sven Goldt, Sven van der Meer, Scott Burkett, and Matt Welsh. *The Linux Programmer's Guide*, 2000.

[9] The HIPL Group. *Host Identity Protocol for Linux.* `http://infrahip.hiit.fi`.

[10] C. Huitema. *RFC 4380: Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs).* Internet Engineering Task Force, 2006. `http://www.ietf.org/rfc/rfc4380.txt`.

[11] M. Komu, T. Henderson, H. Tschofenig, and J. Melen. *RFC 5770: Basic Host Identity Protocol (HIP) Extensions for Traversal of Network Address Translators.* Internet Engineering Task Force, February 2010. `http://www.rfc-editor.org/authors/rfc5770.txt`.

[12] Miika Komu and Janne Lindqvist. Leap-of-faith security is enough for ip mobility. In *Proceedings of the 6th IEEE on Consumer Communications and Networking Conference*, 2009.

[13] J. Lindqvist. *Establishing Host Identity Protocol Opportunistic Mode with TCP Option*. Internet Engineering Task Force, 2006. `http://tools.ietf.org/id/draft-lindqvist-hip-opportunistic-01.txt`.

[14] Paul Mockapetris. *RFC 1034: Domain Names — Concepts and Facilities*. Internet Engineering Task Force, November 1987. `http://www.ietf.org/rfc/rfc1034.txt`.

[15] Teresa Finez Moral. *Backwards Compatibility Experimentation with Host Identity Protocol and Legacy Software and Networks*, December 2008. `http://infrahip.hiit.fi/hipl/thesis_teresa_finez.pdf`.

[16] P. Nikander. *RFC 4423: Host Identity Protocol (HIP) Architecture*. Internet Engineering Task Force, 2006. `http://www.ietf.org/rfc/rfc4423.txt`.

[17] P. Nikander, P. Jokela, and T. Henderson. *RFC 5201: Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)*. Internet Engineering Task Force, 2008. `http://www.ietf.org/rfc/rfc5201.txt`.

[18] Croucher P. *Communications and Networks*. British Library publication, 2007.

[19] C. Vogt P. Nikander, T. Henderson and J. Arkko. *RFC 5206: End-Host Mobility and Multihoming with the Host Identity Protocol*. Internet Engineering Task Force, 2008. `http://www.ietf.org/rfc/rfc5206.txt`.

[20] J. Laganier P. Nikander. *RFC 5205: Host Identity Protocol (HIP) Domain Name System (DNS) Extension*. Internet Engineering Task Force, 2008. `http://www.ietf.org/rfc/rfc5205.txt`.

[21] J. Melen P. Nikander. *A Bound End-to-End Tunnel (BEET) mode for ESP Mode with TCP Option*. Internet Engineering Task Force, 2008. `http://tools.ietf.org/id/draft-nikander-esp-beet-mode-09.txt`.

[22] Jon Postel. *RFC 768: User Datagram Protocol*. Internet Engineering Task Force, August 1980. `http://www.ietf.org/rfc/rfc768.txt`.

[23] Jon Postel. *RFC 793: Transport Control Protocol*. Internet Engineering Task Force, September 1981. `http://www.ietf.org/rfc/rfc793.txt`.

[24] P. Nikander R. Moskowitz. *RFC 5202: Host Identity Protocol (HIP) Architecture*. Internet Engineering Task Force, 2008. `http://www.ietf.org/rfc/rfc5202.txt`.

[25] E. Rescorla. *RFC 2631: Diffie-Hellman Key Agreement Method*. IETF, June 1999. `http://www.ietf.org/rfc/rfc2631.txt`.

[26] J. Rosenberg. *RFC 5245: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. Internet Engineering Task Force, 2010. `http://www.ietf.org/rfc/rfc5245.txt`.

[27] Patrik Salmela. *Host Identity Protocol proxy in a 3G system*. Internet Engineering Task Force, 2005. `http://www.cs.hut.fi/~pmrg/publications/VHO/2005/Salmela_HIPPS.pdf`.

[28] Netlink S.a.s. Netlink - communication between kernel and user. `//www.netlink.it/`.

[29] P. Srisuresh and M. Holdrege. *RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations*. Internet Engineering Task Force, 1999. `http://www.ietf.org/rfc/rfc2663.txt`.

[30] W. Richard Stevens. *TCP/IP Illustrated: the protocols*. Addison-Wesley, February 1994.

[31] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1997.

[32] Andrew S Tanenbaum. *Computer Networks*, 2002.

[33] Sasu Tarkoma. *Understanding Multi-layer Mobility*. IGI Global, 1999.