HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
- Exchange Student -

Backwards Compatibility Experimentation with Host Identity Protocol and Legacy Software and Networks

Final Project

Teresa Fínez Moral

Department of Media Technology
Espoo 2008

HELSINKI UNIVERSITY OF TECHNOLOGY	ABSTRACT OF THE FINAL PROJECT

Author:	Teresa Fínez Moral		
Title of thesis: Backwards Compatibility Experimentation with Host Identity Protocol and Legacy Software and Networks			
Date:	December 5, 2008	**Number of pages:**	74
Department:	Department of Computer Science and Engineering		
Professorship:	T-110		
Supervisor:	Andrei Gurtov Dr.		
Instructors:	Miika Komu M.Sc.		

HIP architecture decouples identifier and locator roles of IP addresses. A HIP communication starts with the Base Exchange, a four-way Diffie-Hellman-based key exchange that authenticates the end-hosts and sets up symmetric key material for IPSec. Before the Base Exchange is launched, the DNS looks up the Host Identity Tag (HIT) corresponding to the peer hostname. When the HIT of the peer is not available, the Base Exchange uses the opportunistic mode. This mode is a leap-of-faith implementation which allows the communication with no HIP-aware peers, without introducing new architecture or pre-configuration to be deployed.

This thesis focuses on adding IPv4 support for applications using HIP in the HIP for Linux (HIPL) implementation. To achieve our goal, we have designed and implemented the Local Scope Identifier (LSI) whose main purpose is to support backwards compatibility with the IPv4 API. Furthermore, LSIs and HITs allow communication amongst IPv4 and IPv6 applications. Therefore, it solves the problem with IPv4 and IPv6 interoperability at the application layer for HIP protocol.

In addition, the LSI design approach proposed can be applied to move an opportunistic mode library from user to system level, due to design similarities. This thesis presents an analysis between the opportunistic mode implemented as a shared library and the proposed design.

In the future we must keep down the loss of packets until the Base Exchange is established. Finally, we discovered during the analysis that FTP needs future support.

Keywords: HIP, LSI, interoperability, opportunistic mode

Acknowledgements

This thesis is an outcome of eight months working at HIIT in Helsinki, Finland. First, I specially would like to thank my instructor Miika Komu for giving me the opportunity to do my thesis in the InfraHIP group and also for all his support and enormous help during this months. I can not forget Andrei Gurtov, who gave me some comments and corrections.

It has been a pleasure to collaborate in a research environment. Furthermore, I want to thank all the people in the InfraHIP project who supported me and made things easier going and specially to René Hummen for his helps and discussions.

This work is dedicated to my family who offered me an unconditional support and all their love in spite of the distance. Thank you for encouraging me to go abroad.

Espoo, December 5, 2008

Teresa Fínez Moral

Contents

Abbreviations and Acronyms

AH Authentication Header

ALGs Application Layer Gateways

API Application Programming Interface

BEET Bound End-to-End Tunnel

BIND Berkeley Internet Domain

CA Certificate Authority

DSA Digital Signature Algorithm

DNS Domain Name System

DoS Denial of Service

ESP Encapsulating Security Payload

FQDN Fully Qualified Domain Name

FTP File Transfer Protocol

HIP Host Identity Protocol

HI Host Identifier

HIPL HIP for Linux

HIT Host Identity Tag

IANA Internet Assigned Numbers Authority

ICMP Internet Control Message Protocol

ICMPv6 Internet Control Message Protocol version 6

IETF Internet Engineering Task Force

IKE Internet Key Exchange

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

IPSec Internet Protocol Security

ISP Internet Service Provider

L4 Layer 4

LoF Leap-of-Faith

LSI Local Scope Identifier

MITM Man In The Middle

NAT Network Address Translation

OSI Open Systems Interconnection

PPP Point-to-Point Protocol

PTR Pointer Record

QoS Quality of Service

RFC Request For Comments

RR Resource Record

RSA Rivest Shamir Adelman

RTT Round-Trip Time

RTO Retransmission Timeout

RVS Rendezvous Server

SA Security Association

SAD Security Association Database

SLIP Serial Line Internet Protocol

SNMP Simple Network Management Protocol

SP Security Policy

SPD Security Policy Database

SPI Security Parameter Index

TCP Transport Control Protocol

TESLA Transparent and Extensible Session-Layer Architecture

UDP User Datagram Protocol

VPN Virtual Private Network

Chapter 1

Introduction

Since the Internet emerged, more than twenty years ago, the set of requirements have changed. The initial design principles and architectural model for the Internet have been shifting from an end-to-end towards an end-to-middle model [39]. In addition, the explosion of mobile technologies have revolutionized the networking. Due to these new demands and technology development, various proposals try to face some of these new challenges for the Internet. The typical solutions are related with the network-level protocols, e.g., Mobile IP. This protocol implies an additional infrastructure creating a home network which sends the packets between it and the mobile host since each mobile host has an invariant home address. The drawbacks of this approach are lack of security, performance and internetworking [58]. One such alternative is the Host Identity Protocol (HIP). Further details about a comparison between these two approaches can be found in [12].

HIP addresses a number of new requirements related to end-host multihoming, mobility and security by introducing a new cryptographic namespace, splitting identifier and locator roles of Internet Protocol (IP) addresses. In this new design, the network layer identifiers continue being bound to the IP addresses and are in charge of the routing. By contrast, the transport layer is bound to the endpoint's identity which the underlaying HIP layer defines. This approach has the benefit that although network locators change, transport-layer connections persist.

HIPL positions the HIP architecture towards IPv6-compatible applications, but there are still many IPv4-only applications. Although the IPv6 deployment is a fact, it will take some time until we can refer to an IPv6-only environment. Therefore, there will be a transition period where both IP versions must coexist. On the other hand, its architecture already supports IPv4 and IPv6 networks [1].

In addition, HIP prefers Public Key Infrastructure (PKI) [61] which is not globally deployed. Concretely, HIP identities are cryptographically based, where a Host

Identity consists on a public key. This Host Identity could be stored in DNS, a PKI or be anonymous.

This thesis tackles these two problems, support for IPv4-only applications and networks without PKI support, to boost HIP deployment.

1.1 Problem statement

The following is the problem statement this thesis deals with:

1. Facilitate the deployment of HIP to legacy applications and networks. We can subdivide the problem into two topics. The first point is to add support for IPv4-only applications. As a consequence, we add interoperability between IPv4 and IPv6 at the application layer using HIP. Our main contribution is the design and implementation of LSIs, an IPv4-sized identifier which allows to run IPv4 applications using HIP.

2. Support networks without PKI when the client is using HIP. The most common scenario is to establish communication with a peer not HIP-aware. In this case, HIPL fallbacks to a non-HIP communication.

We discuss more deeply the problems of deployment in chapter 3.

1.2 Scope

The scope of this thesis is to design solutions for HIP for Linux (HIPL) implementation to support IPv4-only legacy applications and to allow HIP-based connectivity without PKI. We implemented the IPv4 support based on Local Scope Identifiers (LSIs). Meanwhile HIP support for networks without PKI is based on the opportunistic mode and a shim layer between transport and network layers. The design and implementation use C [27] as programming language.

We also provide a performance analysis of the two new implementation features.

The main objective of the implementation is to prove that the design can work in practice. However, the performance and reliability are a secondary evaluation criterion.

1.3 Structure of the thesis

The rest of this thesis is organized as follows:

Chapter 2 details and examines the fundamental protocols we need for introducing the research area of this thesis. In addition, it defines the major linux libraries needed for designing and implementing our solution.

Chapter 3 discusses the problem statement presenting the deployment of the problem and deepening in the subproblems.

Chapter 4 presents our design and chapter 5 introduces implementation details of Local Scope Identifiers and system-based opportunistic mode.

Chapter 6 explains the implementation analysis. Furthermore, it presents some design alternatives for the current design. The analysis contains measured results and charts, and then discussion of the results. Moreover, we go through FTP referral problems and Maximum Transfer Unit (MTU) value modifications related to LSI and the LSI address space. Finally, we study the LSI compatibility with the current HIPL project and other extensions under development.

Chapter 7 summarizes the main conclusions of this project.

To finalize, chapter 8 describes directions for future work, giving thoughts and ideas for future improvements and extensions.

Chapter 2

Background

The aim of this chapter is to describe the background topics. We assume that the reader understands the basic concepts of the TCP/IP suite and has skills in C programming. The chapter is organized as follows: section 2.1 compares the main differences between IPv4 and IPv6. Secondly, section 2.2 gives a general overview of HIP. Following that, section 2.3 describes DNS and section 2.4 describes IPSec. Apart from the general overview given for DNS and IPSec, we also include an approach about how they are integrated into HIP. Then, we talk about issues related to UNIX programming: raw sockets, netlink, LD_PRELOAD, TUN/TAP and dummy interface. Finally, in section 2.9 we discuss also an extensible session-layer architecture using interposition libraries.

2.1 IPv4 vs IPv6

The main motivation for updating IP [49] was the lack of address space with IPv4. Network Address Translation (NAT) has been a short-term solution that prolonged the lifetime of IPv4. However, this and other challenges such as difficulties in deploying new protocols made a new protocol version necessary: IPv6. IPv4 and IPv6 have a lot of conceptual similarities, but IPv6 introduces new features in routing and network autoconfiguration that IPv4 does not have. Unfortunately, these IP versions are not directly compatible, hence programs and systems designed to one standard can not communicate with those designed with the other [34]. The differences between the protocol versions can be grouped into seven categories [6]:

1. Address size. IPv6 addresses are 128 bits long, while IPv4 address size is 32 bits. In other words, an IPv6 address is four times longer than an IPv4 address. This increases the address hierarchy, creating additional levels for addressing.
2. Optional header. IPv4 has a variable length header, meanwhile IPv6 simplifies it. IPv6 base header has a fixed size that can be followed by optional headers and, moreover, it does not include a checksum.

3. Improved options. IPv6 includes some options not available for IPv4. Furthermore, the wire format for the options speeds up the packet-processing time in routers. This is because they are handled with extensions that not all the routers have to process.

4. Security. IPv6 introduces security mechanism at the network level providing authentication, confidentiality and integrity using the IPSec protocol. IPv6 specifications mandate IPSec inclusion while this is optional with IPv4.

5. Provision for protocol extension. IPv6 design is more extendable in order to support new technology changes.

6. Support for address autoconfiguration and renumbering. IPv6 allows to assign local addresses automatically, as well as renumbering networks at a site more dynamically.

7. Support for resource allocation. IPv6 supports the same Quality of Service (QoS) features as IPv4, including differentiated service (DiffServ) indication. However, IPv6 includes a traffic flow field which can provide a solid base to build QoS protocols.

2.2 Host Identity Protocol

In this section, we give a HIP overview based on the related Request For Comments (RFCs), Internet Drafts and a number of articles.

2.2.1 New stack architecture

Processes use sockets to send and receive network data in the current network architecture. The traffic of different processes is demultiplexed by IP address, port and protocol. On the contrary, HIP binds transport layer sockets to Host Identifiers (HIs), allowing persistent bindings throughout IP addresses changes. As a result, HIP architecture splits the identifier and locator roles of the IP address [46]. While the HI is the end-point identifier, the IP represents the topological location of the host in the network. This requires a translation mechanism between the HI and the IP, and vice versa. For achieving this translation mechanism, HIP locates a new layer called Host Identity Layer in the TCP/IP stack, between the transport and networking layer. The comparison between the current Internet bindings and the ones introduced by HIP architecture is shown in Figure 2.1.

2.2.2 New name space layer and identifiers

HIP [52] introduces the HIP layer to the TCP/IP networking stack, as Figure 2.2 shows. The new layer is based on HIs and it is located between transport and net-

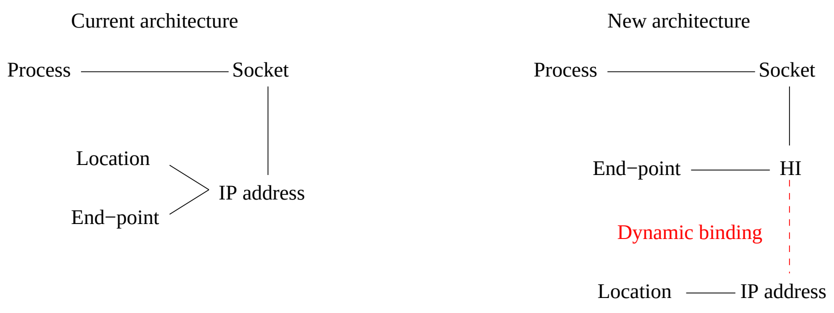

Figure 2.1: Bindings comparison [46]

work layers. The Host Identity namespace is decentrally administered. Each host creates and manages its own identities. A HI is effectively a public key of a private-public key pair.

A HI is a public key of an asymmetric key pair. Using a public-key-based HI we improve the security of the communication, increase protection against man-in-the-middle (MITM) attacks and provide end-host authentication. HIP supports public Rivest Shamir Adelman (RSA) and Digital Signature Algorithm (DSA) keys. Therefore, the cryptographic nature of HI makes identity theft computationally difficult.

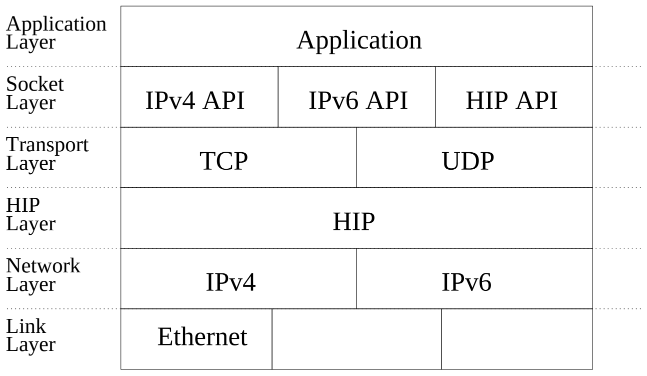

Figure 2.2: HIP layering architecture

HIT

A Host Identity Tag (HIT) is a 128 bit hash of a HI. The length of the HI depends on the strength of the public key pair. In the current implementations, it is 1024 bits. The fixed-size HIT is necessary because HIs can be of variable length which makes protocol encoding and sockets API binding challenging.

A HIT is a type of Overlay Routable Cryptographic Hash Identifier (ORCHID) [45], i.e. applications and APIs use HIT as an endpoint identifier. An ORCHID resembles an IPv6 address which is not routable from an IPv6 layer point-of-view and it has a prefix in order to differentiate it from a real IPv6 address. An algorithm generates the ORCHID taking as input the following parameters: a unique or statically unique bitstring, a context id which is randomly generated and defines the usage context, a hash function and a 28-bit prefix.

ORCHIDs are statically unique. However, there exist two situations where ORCHID collisions could be possible. The first scenario is when two different input bitstrings within the same context map to the same ORCHID. In this case, the state-set-up mechanism resolves the conflict. The second situation happens when two input bitstrings in different contexts map to the same ORCHID. The solution is to indicate a conflict when this ORCHID is already in use.

The most important properties of HIT are:

- Same length as IPv6 addresses. Thus, HIP can use HITs in existing IPv6 applications.
- Self-certifying. HITs are self-certifying because of the second-preimage resistance property of hash functions. In other words, given a Host Identity *K1*, finding a different Host Identity *K2*, where hash(K1) = hash(K2), is computationally hard. In the future, as the computational power increases, this feature could disappear. In this case, the length of the primary key should be increased in order to preserve security, nevertheless the HIT size is fixed. The best approach is to use the Host Identity when identification is needed [43].
- Probability of collision very low. For any given HIT the probability of collision is approximately $(2^{-126}) * N$ where N is the total number of HITs [8]. The birthday paradox states that given a large enough population and a hash space, there may be collisions. A 128-bit hash will have 0.001% collision chance in a $9x10^{16}$ population [53].

LSI

The Local Scope Identifier (LSI) is a 32-bit representation for a HI. Thus, HIP can build an LSI taking the last bytes of the HIT. LSIs have the same length as IPv4 addresses in order to support IPv4-only legacy applications. LSIs are only valid in the context of the local host, similarly to socket descriptors. Another feature is that LSIs are shorter than HITs, increasing the probability of collision.

2.2.3 Interoperating with IPv4 and IPv6

Nowadays the deployment of IPv6 networks is slowly replacing IPv4 in the Internet. Nevertheless, during this transition period between the two versions, network nodes must be able to communicate with both protocols. The transition mechanism [2] may include:

1. Domain Name Server (DNS) upgrade, introducing AAAA Resource Records.
2. Dual protocol stacks. Parallel support for both IPv4 and IPv6.
3. Tunneling. IPv6 packets are tunneled over IPv4 regions.
4. A standard IPv6 programming interface to upgrade existing IPv4 applications and ease the development of IPv6 ones.

HIP decouples the transport and internetworking layer, thus it solves interoperability problems at the application and network node [22]. As shown in Figure 2.1, transport layer sockets are not bound to IP addresses, then a legacy IPv4 application can talk with a legacy IPv6 application. HIP supports IPv4 and IPv6 sockets because it is able to resolve an IP to an end-point identifier: an LSI for IPv4 applications or a HIT for IPv6 applications.

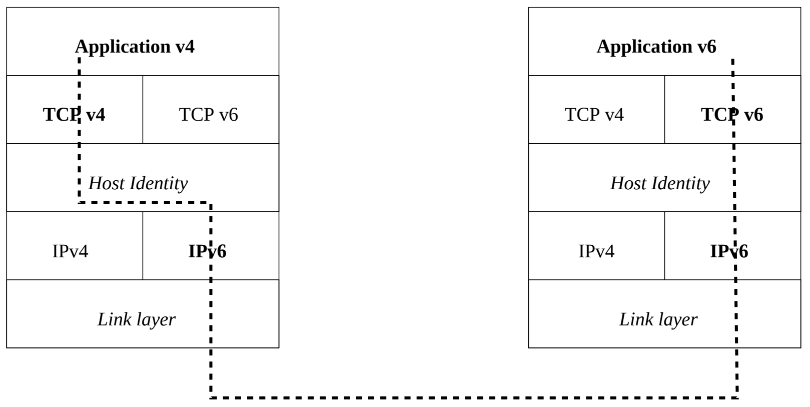

Figure 2.3: Example of communication across IP versions with HIP

Figure 2.3 illustrates a typical scenario where an IPv4 application communicates over an IPv6 network with an IPv6 application. The connectivity between both hosts can be either IPv4-based or IPv6-based, independently of the application.

2.2.4 HIP Base Exchange

The first phase in a HIP communication is the Base Exchange [52], as shown in Figure 2.4. The Base Exchange is a four-way handshake that performs end-to-end authentication. The first packet triggers the communication and the other three consist of a Diffie-Hellman key exchange which creates a piece of key material that IPSec uses for encrypting and protecting the data.

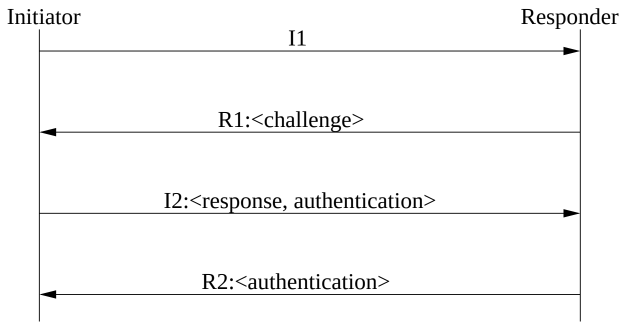

Figure 2.4: HIP Base Exchange

The host initiating the communication is the *initiator* and the peer host is the *responder*. Typically, the initiator is the client host and the responder a server. The initiator begins the handshake procedure by sending an I1 packet. When the peer receives the I1 packet, it generates an R1 packet. The R1 packet contains a puzzle with a configurable difficulty level, the initial Diffie-Hellman parameters and a signature. The puzzle is a cryptographic challenge that the initiator must solve in order to continue the Base Exchange. The initiator tries to solve the puzzle and answers to the responder with an I2 packet. The I2 packet must contain the solution of the puzzle and Diffie-Hellman parameters. If the solution for the puzzle is wrong, the peer discards the I2 packet and aborts the communication. Otherwise, the peer finalizes the Base Exchange by sending a signed R2 packet.

A more detailed analysis of the HIP Base Exchange protocol can be found in [57].

2.2.5 HIP Opportunistic Mode

The opportunistic mode is based on the Leap-of-Faith (LoF) mode of operation. LoF means that the initiator initiates the first connection without knowledge of peer identity. Subsequent communications can use a cached identity of the peer. It is based on the assumption that during the first connection, there is no attack against the connection [37]. In consequence, it is prone to MITM attacks because the initiator does not know the responder's identity. A third host placed between the initiator and responder can intercept the communication and impersonate the responder. Therefore, it is recommended to use it only in trusted environments [10].

Figure 2.5 illustrates an MITM attack with opportunistic mode. Alice is the initiator and Bob the responder. When Alice sends an I1 packet, Trudy intercepts the packet and responds with her own R1 to Alice. Consequently, Alice will think she is communicating with Bob, but she is communicating Trudy.

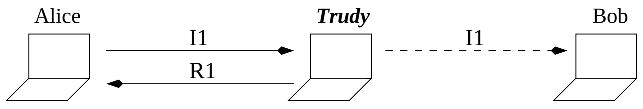

Figure 2.5: Man in the Middle attack during Base Exchange in Opportunistic Mode

TCP Extension for Opportunistic Mode

When the initiator starts the Base Exchange with opportunistic mode, it waits until it receives an R1 packet from the peer or times out. If the peer is not HIP-enabled, the initiator will be waiting until a timeout occurs. The extension discussed in this section provides a way to decrease latency when the peer is not HIP-enabled based on [5], improving user experience for HIP.

We explain the scenario independently of the peer capability to support HIP. Firstly, a client application starts communications with a peer. The HIP layer blocks the call and requests the HIT matching the IP. We assume that the initiator does not know the HIT of the peer. Thus, the Base Exchange starts in opportunistic mode, sending an I1 packet together with a TCP SYN message with a special TCP option.

If the responder is HIP-enabled, it processes the I1 packet and generates and sends an R1 packet. The initiator receives the TCP SYN ACK option and discards it. At this moment, the responder replies with R1 and the initiator application establishes the communication with the peer HIT.

On the other hand, if the responder is not HIP-aware, it does not understand the I1 packet and the TCP special option. It answers the initiator with a TCP SYN ACK message. As the special option is not present in the response, the initiator detects that the peer does not support HIP. The initiator unblocks the application and establishes the connection without HIP.

This solution only works for TCP traffic, because it is based on the handshake of the TCP protocol [50]. However, it is better than some other approaches, as e.g. including an option in the IP header, because most currently deployed middleboxes drop the packet [5].

2.3 Domain Name System

This section introduces the basics of the Domain Name System (DNS) protocol [29]. DNS acts as an "Internet phone book". It is a distributed database which maps a human name: a hostname, e.g. *infrahip*, or a Fully Qualified Domain Name (FQDN), e.g. *infrahip.hiit.fi*, to an IP address. The DNS name space follows a hierarchical organization where each node has a label. The domain name of a node is the list of labels separated by a period, in this case a dot, starting at that node and ending at the root node [54].

2.3.1 Resource Records

A Resource Record (RR) [55] is a DNS name record. There are different types of RRs:

- A: Maps a hostname into an IPv4 address.
- AAAA: Also called a *quad A* record. Maps a hostname into an IPv6 address.
- PTR: Called *pointer records*. Maps IP addresses into hostnames. Provides reverse resolution to A and AAAA RRs, translating an address, either IPv4 or IPv6. PTR records involve appending `in-addr.arpa` for IPv4 addresses and `ip6.int` for IPv6 addresses.

2.3.2 Resolvers and Name Servers

The *resolver* is usually implemented with the aim of providing communication between the application and the DNS server by sending DNS queries and managing the responses. UNIX hosts use mainly two library functions to access the resolver.

Function `gethostbyname` translates hostnames to IP addresses and `gethostbyaddr` provides the reverse functionality. Their main problem is that they only support IPv4.

Functions `getaddrinfo` and `getnameinfo` also support IPv6 because they return `sockaddr` structures. This structure is shown in Figure 2.6 and as it is visible, it defines a generic socket address structure.

```
struct sockaddr
{
  uint8_t    sa_len;
  int        sa_family;
  char       sa_data[14];
};
```

Figure 2.6: The `sockaddr` structure

The `getaddrinfo` function resolves a hostname to an IP address. The function and the associated data structure are shown in Figure 2.7. Function `getnameinfo` is the complement of `getaddrinfo`. It inputs an IP address and returns the hostname and the service.

```
struct addrinfo
{
  int                ai_flags;              /* Input flags */
  int                ai_family;             /* E.g. AF_INET6, AF_INET */
  int                ai_socktype;           /* Socket type */
  int                ai_protocol;           /* 0 or IPPROTO_xxx for
                                            IPv4 or IPv6 */
  size_t             ai_addrlen;            /* Length of ai_addr */
  char              *ai_canonname;          /* Canonical name */
  struct sockaddr *ai_addr;                 /* Pointer to socket address
                                            structure */
  struct addrinfo *ai_next;                 /* Points next addrinfo */
};

int getaddrinfo(const char *hostname, const char *service,
        const struct addrinfo *hints, const struct addrinfo **result)
```

Figure 2.7: The `addrinfo` structure and the `getaddrinfo` function

2.3.3 DNS Extension for HIP

DNS extensions for HIP are defined in [43]. They describe a new RR for HIP HIs where the resolver library returns a HIT for IPv6 or HIP-aware [38] applications, or an LSI for IPv4-only applications.

The information that is stored in DNS consists of:

- IP addresses, via A or AAAA
- HI, HIT and possibly a set of rendezvous servers (RVS) [31] via HIP.

The mechanism implemented in HIPL is shown in Figure 2.8. When the application triggers a DNS query, the resolver library returns the peer host HIT and IP. Firstly, this information is stored in HIP data structures and secondly, the resolver returns the HIT to the application. The application connects with the peer host using the HIT. Before the data traffic starts, HIP initiates the Base Exchange with the peer in order to establish the key exchange to protect the data.

The main advantage of storing HIs to DNS is to prevent MITM attacks. In order to prevent spoofing attacks, it is recommended to use DNSSEC [10]. On the other hand, the main disadvantage is that contemporary DNS can not resolve HIs to hostnames or IP addresses.

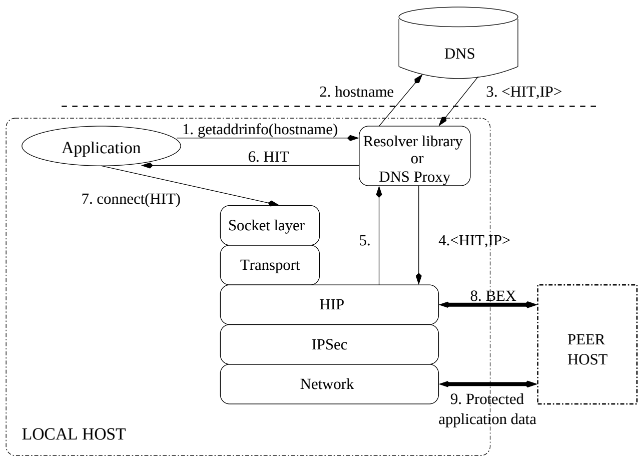

Figure 2.8: Resolver interaction

When user wants to use HIP with a legacy application, one problem for the previous solution occurs when the application wants to use an IP address instead of a HIT. One solution for this is to introduce a domain name prefix specific to HIP. E.g a "hip-www.example.com" DNS query could return a HIT or an LSI [62].

2.4 IP Security

IP Security (IPSec) [26] operates at the network layer providing security support for transport-level protocols. It secures communications by authenticating and/or encrypting each packet. As it is below the transport layer, it is transparent to applications.

2.4.1 Architecture

Security Policy

A Security Policy (SP) tells an IPSec implementation how to manage the different datagrams received or sent to the network device. These rules decide when the host applies or bypasses IPSec protection, or discards a packet. They are stored in the Security Policy Database (SPD).

Security Association

A Security Association (SA) defines a set of security-related information, including the symmetric key and algorithm, describing an IPSec protection between two devices. The management of SAs involves a three-tuple:

- **Security Parameter Index** (SPI). Pseudo-randomly derived number which identifies a particular SA uniquely between two machines. Its purpose is to distinguish among different SAs from each other. SPI is sent on the wire.
- **Destination Address**. Address of the device for whom the SA is established.
- **Security Protocol Identifier**. IPSec provides the following security protocols for the association:
  - Authentication Header (AH). Provides data integrity and authentication, but not privacy because the IP payload is not encrypted. AH is not commonly used in practice.
  - Encapsulating Security Payload (ESP). Encrypts and authenticates a connection data flow. The use of ESP [25] is more common than AH because it is a superset of ESP.

  Each protocol supports two modes of operation [26]:

  1. **Transport mode**. This mode protects the traffic between two end-hosts [42]. The transport mode is not compatible with NATs which is one reason why it is not used widely.
     - AH. Offers security to selected portions of the IP header, header extensions and selected options.
     - ESP. Offers security for protocols higher than IP, because only the IP payload is encrypted.
  2. **Tunnel mode**. Used for network-to-network, host-to-network or host-to-host communications over Internet. IPSec encapsulates the whole IP datagram, including the header. Therefore, the packet must be encapsulated within a second IP packet in order to be routed. If AH is employed, some parts of the built IP header can be encrypted.

Secure Association Database (SAD) stores information related with SAs. An SA is unidirectional, meaning that a single SA only handles inbound or outbound traffic. To secure bidirectional communication, two SAs must be created.

The main difference between SPs and SAs is that an SP specifies *what* we want to encrypt and an SA details *how* we want to secure it.

2.4.2 IPSec with HIP

Base Exchange sets up key material for SAs to encrypt the data traffic. After the Base Exchange, the ESP protocol [44] protects user data.

HIP currently supports a new ESP mode called Bound End-to-End Tunnel (BEET). This mode combines transport mode format and tunnel mode semantics. The "outer" addresses go on the wire and the "inner" addresses are the ones the application sees. The BEET mode with HIP uses HITs as inner addresses and IP addresses as outer addresses. Figure 2.9 shows the difference between BEET, tunnel and transport mode. As we can observe, transport and BEET mode headers have the same syntax. Furthermore, BEET mode semantics are similar to tunnel mode ones, although the tunnel mode introduces an extra IP header.

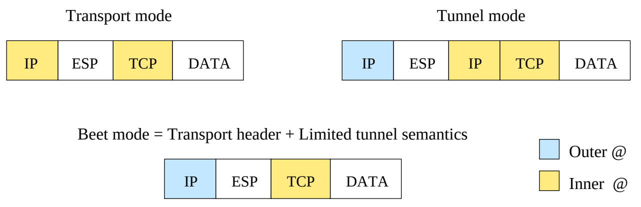

Figure 2.9: Comparison of BEET, transport and tunnel mode

The BEET mode does not include HITs in the packet which goes on the wire. Instead, the SPI implicitly defines the HITs. IPSec links each association to two ESP SAs, one for incoming and one for outgoing packets. An SA pair is indexed by 2 SPIs and 2 HITs. HIP updates these associations and removes them when the HIP association finishes.

2.5 Raw sockets

A socket [63] in the context of TCP/IP is an end-point of a bidirectional communication flow. It allows communication between an application and the TCP/IP stack.

Basically, we can divide socket types into three types [17]:

- connectionless sockets (`SOCK_DGRAM`)
- connection-oriented sockets (`SOCK_STREAM`)
- raw sockets `SOCK_RAW`

However, we limit the discussion to raw sockets. One of the most important features of raw sockets that we use in the LSI implementation of HIPL is the possibility to access the packet header and change it. Non-raw sockets usually strip the header and receive only the payload.

Raw sockets allow us to implement Layer 4 protocols (L4) or L3.5 like HIP in the userspace and to implement some processing from those protocols that are normally processed in the kernel [4].

2.5.1 Creation

When an application establishes network communications, it first needs to open a socket by calling the *socket()* function. The first argument to the function sets the address family. It can be `AF_INET` for IPv4 or `AF_INET6` for IPv6-enabled sockets. The second argument determines the socket type. For example, the `SOCK_RAW` value creates a raw socket. Finally, the third argument defines the protocol. Figure 2.10 shows the prototype of the *socket* system call for creating a raw socket.

```
int socket(int family, int type, int protocol)
```

Figure 2.10: The `socket` function

Raw sockets can use the function *bind()* in Figure 2.11. It can set the source address used to send output packets over the raw socket and it also acts as an input filter. One of the most common errors from `bind` is *Address already in use.* This error is due to an existing connection that is already listening on the port in which the server tries to bind. One possible solution is to bind the port to the wildcard interface after the `send` call finishes.

2.5.2 Output packet

The user process must calculate and set the header checksum of the transport layer packet before sending it, assuming that the specific transport protocol employs checksums. Raw sockets allow that the protocol implementation can either specify the

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

Figure 2.11: The `bind` function

whole IP header or let the networking stack creates it. The implementation controls this using the `IP_HDRINCL` socket option. Only IPv4 sockets can use the `IP_HDRINCL` option. Figure 2.12 shows the *sendto* interface.

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

Figure 2.12: Function for sending data

2.5.3 Input packet

The kernel receives a datagram and passes it to the raw socket only if the three following conditions are true:

- The raw socket protocol and the received datagram protocol field match.
- If the local IP address bound to the raw socket by *bind* matches the destination IP address of the packet or the socket has not been bound or it is bound to `INADDR_ANY` for IPv4 or `IN6DDR_ANY_INIT` for IPv6.
- If a foreign IP address was specified with *connect* and matches the source IP address of the packet.

If these three prerequisites are met, the kernel passes the datagram to the raw socket, including the IP header if the `IP_HDRINCL` socket option is activated.

2.6 Introduction to the libipq library

Libipq [33] is a development library for iptables userspace packet queuing [32]. Netfilter provides a mechanism to queue packets to the userpace and then outputting them back to the kernel with a verdict that determines whether the kernel must accept or drop the packet. HIPL uses this mechanism for modifying the packet and reinjecting it to the kernel. We will give an introductory overview to the basic API usage.

Firstly, in order to redirect the packets to the userspace, an iptables rule must be set up with the argument QUEUE as follows:

```
iptables -I OUTPUT -d LSI -j QUEUE
```

The before rule queues all output packets whose destination address is an LSI. The next step is to initiate an ipq handle to read the queued packets and set the mode to get the data. There are two possible modes: metadata (`IPQ_COPY_META`) or payload (`IPQ_COPY_PACKET`). These functions are illustrated below.

```
h = ipq_create_handle(0, PF_INET)
ipq_set_mode(h, IPQ_COPY_PACKET, BUFSIZE)
```

Function *ipq_read* reads packet contents and function *ipq_message_type* returns information about this content. Specifically, if it is a network packet, it returns the value `IPQM_PACKET` and if it is an error message, it returns the value `NLMSG_ERROR`. Then, *ipq_get_packet* gets the packet content. At this point, packet content can be modified prior to reinjection back into the kernel. If an application modifies a packet, it must update checksums. Otherwise, the kernel discards the packet because of an invalid checksum. Furthermore, queue handlers are IP protocol specific, hence the packet IP family must stay unchanged. Function *ipq_set_verdict* allows us to set the packet verdict: `ACCEPT` or `DROP`. If the verdict is `ACCEPT`, the packet continues the process through the stack. A `DROP` verdict drops the packet. The function *ipq_-set_verdict* sets the verdict depending on the id of the packet, this is a field which allows libipq to distinguish the packets because its value is unique.

```
ipq_read(h, buf, BUFSIZE, 0)
ipq_message_type(buf)
ipq_get_packet(buf)
ipq_set_verdict(h, packetId, NF_ACCEPT,0, NULL)
```

Finally, the next call closes the handle:

```
ipq_destroy_handle(h)
```

2.7 Introduction to Netlink

Netlink [15] allows userspace applications to communicate with the kernel. It is used e.g. by Berkeley Internet Domain (BIND) [54] to configure routing-related information. It sets the following aspects of the network control plane [24]:

- `NETLINK_ROUTE`. Userspace routing daemons use netlink to update the kernel routing table.

- `NETLINK_FIREWALL`. IPv4 firewall code sends packets through netlink.
- `NETLINK_NFLOG`. Communication channel for the userspace iptable management tool and kernel space Netfilter module.
- `NETLINK_ARPD`. Manages the ARP table from userspace.

To use netlink, the application has to create a socket. The netlink address family is `AF_NETLINK` and the type is either `SOCK_RAW` or `SOCK_DGRAM`. The protocol type refers to the netlink feature the socket uses.

2.7.1 Message Format

Figure 2.13 illustrates netlink messages. All netlink messages consist of a header plus a payload.

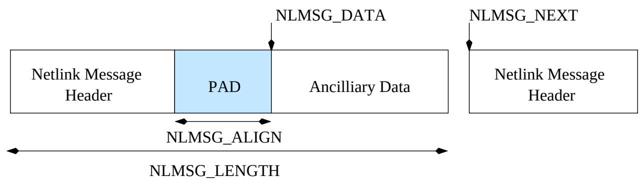

Figure 2.13: Netlink message and netlink macro interaction

The header contains metadata about the message. The header is depicted in Figure 2.14.

```
struct nlmsghdr{
        __u32 nlmsg_len;    /* Message length: Header + data */
        __u16 nlmsg_type;   /* Message type */
        __u16 nlmsg_flags;  /* Additional flags */
        __u32 nlmsg_seq;    /* Sequence number */
        __u32 nlmsg_pid;    /* Sending process PID */
}
```

Figure 2.14: The netlink header structure

2.7.2 Netlink Macros

The following macros build and manipulate the messages. For more information, see [15].

- `NLMSG_ALIGN`. Used internally by the other macros. Rounds up the length of a netlink message.
- `NLMSG_DATA`. Given a pointer to a netlink header structure, this macro returns a pointer to the ancillary data.
- `NLMSG_LENGTH`. Sets the nlmsg_len field of a netlink message header. The macro returns the size of the payload including the header, rounded up to the nearest `NLMSG_ALIGNTO` bytes.
- `NLMSG_NEXT`. If a netlink message has more than one response, this function finds the next response.

2.8 Introduction to Dynamic Linking with LD_PRELOAD

Dynamic linking provides the possibility of intercepting a function call that an application makes to any shared library at running time [9]. Once the interposer library intercepts the call, it can call the real function the application intended to call, as well as overwrite the function call and modify the original functionality. E.g. in HIPL we can load the variable with three intercepted libraries:

```
export LD_PRELOAD=libinet6.so:libhiptool.so:libhipopendht.so
```

The interposition with the real call is divided into two steps:

1. Create the interposer library. This will overwrite functions from other libraries, like e.g. glibc. The interposer library may contain the implementation of the functions to intercept.
2. Set the LD_PRELOAD shell environment variable with the absolute path to the interposer library.

When the application calls the function that the interposer library redefines, as this library is loaded before the other ones, the loader first finds the symbol of the modified library when trying to resolve the external reference [19].

If the user or administrator does not want to use the interposition library, he or she must undefine the LD_PRELOAD variable.

We must take into account that setuid programs disable LD_PRELOAD, in order to prevent security problems. In addition, LD_PRELOAD is not available for all operative systems and might have problems with some Linux extensions, such as e.g. Security-Enhanced Linux (SELinux). Another disadvantage is the use of LD_-PRELOAD with chaining applications, i.e., when an application is preloading the same function with two different interposition libraries, how do we know which is the library that the loader calls first?

2.9 TESLA

TESLA is a transparent and extensible session-layer architecture for end-to-end network services [21]. There has been an increasing importance of session-layer services in the Internet, i.e., services which have a common flow between a source and destination, and produce groups of flows using shared code and possibly shared state. Current research focuses on increasing transport-level functionality, some examples are: sharing congestion information, end-to-end session migration for mobility, encryption services or setting up multiple connections to improve the throughput between a source and a destination. Originally these services were in the kernel level, though it would be advantageous to have them in the user level. This later approach is a complicated process, because the implementation must specify the internal logic, algorithms and handle details such as process management or interprocess communication, increasing the programmer's time and effort. The three design goals of TESLA are:

1. Provide a high-level abstraction to session services, which operate on network flows and treat flows as objects.

2. Configure session services transparently from the application. TESLA allows services to define APIs to be exported to TESLA-aware applications.

3. Provide the possibility of composing different services to offer new functionality. TESLA writes session services as event handlers with a callback-oriented interface.

TESLA is a C++ framework which could be configured using an interposition library to modify the interaction between the application and the system. An implementation of this design is available in [60].

2.9.1 Architecture

As we previously explained, TESLA is an interposition layer between the application and the operating system or libraries, providing an abstraction level for session-layer services. It uses the *flow handler* concept to provide this abstraction. Each session service is an instance of a flow handler where TESLA allows communication between the different session services. On an abstract level, a flow handler deals with traffic corresponding to a single socket. There is another flow type called *network flow* which is a stream of bytes sharing a logical source and destination. The flow handler operates or transforms an input byte stream, such as e.g. compressing or encrypting the input network flow, and returns one or more network flows. Therefore, input flows correspond to upstream handlers, such as end application, and output flows map to down-stream handlers, e.g. `send` routine.

2.9.2 Interposition

TESLA acts as an interposition library. The library `libtesla.so` is added as a shared library with the `LD_PRELOAD` environment variable, a concept discussed in section 2.8. This library contains the interposition library but not the handlers. The handlers are provided by the `flow_handler` API. Flow handlers are virtual methods characterised by being asynchronous and event-driven, hence methods must return immediately. We can divide them into two groups:

- *Downstream methods.* Invoked by the input flow, they provide an abstract flow to the upstream handler.
- *Upstream methods.* Invoked by the handler output flow. They act as callbacks invoked by the upstream handler.

Some flow control methods exist in both groups in order to avoid bottlenecks in the application or in TESLA. A flow handler may signal the input flow to stop sending data and later on restart the handler, as well as an output flow may check whether the upstream handler is available. Sometimes the handler must re-enable data flow after certain period of time and TESLA provides a `timer` for this purpose. Moreover, in some cases the handler must provide additional services. For this reason, TESLA provides a mechanism to send events asynchronously to the application.

2.10 TUN/TAP

TUN and TAP [30] are virtual network kernel drivers. TUN/TAP is similar to a Point-to-Point or Ethernet device, which receives and sends packets between userspace programs. TUN/TAP usually sends Ethernet or IP frames, depending on the chosen driver, between the host network and a process. The main purpose of TUN/TAP is tunneling network traffic.

TUN simulates a network device and operates at the network layer (Layer 3) of the OSI model. It is used for routing packets. A userspace application can send IP frames to the interface */dev/tunX* and the kernel will receive the frame from this interface, as well as the other way around.

TAP simulates an Ethernet device and operates in the data link layer (Layer 2) of the OSI model. It can be used to create a network bridge. A userspace application can send Ethernet frames to the interface */dev/tapX* and the kernel can receive frames from this interface.

The main difference between TUN and TAP is that TUN operates with IP frames and TAP with Ethernet frames.

TUN/TAP is used for Virtual Private Networks (VPNs). Some related projects using it are OpenSSH or OpenVPN. In addition, TUN/TAP is also used for virtual machine networking. Projects such as Bochs or VirtualBox use it.

2.11 Dummy interface

TCP/IP uses a dummy interface to assign an IP address to the localhost. Serial Line Internet Protocol (SLIP), Point-to-Point Protocol (PPP) and HIP for HIPL implementation use dummy interfaces. This interface is Linux-specific.

The reason for a dummy interface is the need for an internal IP address although the host is not connected to an Internet Service Provider (ISP). There are network-aware applications such as mail which need to have an IP address to connect to, even if does not lead anywhere. For example, consider a laptop which is disconnected from the network, with the loopback interface as its single network active device. An application may want to send data to another application in the same host. The application tries to send data with the IP assigned by the ISP but the laptop is not connected. Thus, the kernel does not know the IP and discards the datagram, returning an error to the application. However, if the dummy interface serves as the alter ego of the loopback interface with the external IP address assigned and routed, every datagram to this IP is delivered locally [28].

Chapter 3

Problem Statement

3.1 Deployment Problem

Amongst the active implementations of the HIP protocol, HIPL developed by HIIT [13] is the only one which does not yet offer total support for IPv4-only applications. We must note that IPv4-only applications are only supported by HIPL in the opportunistic mode [14].

Although IPv4 is still the facto of today's Internet, network infrastructure and applications are transitioning towards the next generation of Internet Protocols, IPv6. Nowadays, as we can not talk about an IPv6-enabled Internet, both IP protocols must coexist.

The current HIPL implementation already supports IPv4 and IPv6 networks, as well as IPv6 applications. For IPv4 support, the strategy is to map an IPv4 address in an IPv6 structure. Thanks to this mechanism, the whole of the code structure for IPv6 can be reused for IPv4. However, there are still some problems with IPv4 and IPSec, such as the legacy NAT traversal [1].

However, HIPL does not yet support IPv4 applications. The existence of IPv4 support is important because there is yet a lack of IPv6 applications for Linux.

We assume that there will be no flag day to deploy HIP and especially that the Internet infrastructure (DNS servers) will not have HIP support immediately. Therefore, we assume that HIP requires tools for incremental deployment. The opportunistic mode can be used as a mechanism to discover when the communications can be established using HIP with the peer. Currently, the opportunistic mode design is an interposition library, but this thesis proposes to change this schema.

The main topics of this thesis are:

- Describe the LSI identifier.
- Support IPv4-only applications. The scope is limited to the ICMP and ICMPv6 protocol, as well as TCP and UDP transport protocols.
- Add interoperability between IPv4 and IPv6 applications.
- Support opportunistic mode as a system library.

The design and implementation are concentrated around boosting HIP deployment.

3.2 Elaboration of the Deployment Problems

This section elaborates the main deployment problems that we already listed in the previous section. We describe the main problems and introduce a brief description of how our solution can solve them.

3.2.1 Describe the LSI Identifier

The LSI identifier is syntactically an IPv4 address but semantically represents a HIP identifier. There is an open discussion about which must be the most convenient range for LSI address space. This address space may be static or dynamic. However, whatever option is chosen introduces different problems which would be analysed in this thesis.

One aim of this thesis is to widely document the LSI identifier which is always mentioned very briefly in different RFCs and drafts.

3.2.2 Supporting IPv4-only Applications

Currently IPv6 applications are supported by HIP using the HIT as an application layer identifier. HITs resemble IPv6 addresses, thus developers do not have to modify the application to make it use HIP and the application is unaware of HIP. A similar mechanism is necessary to support IPv4 applications. Therefore, HIPL implementation must be extended in order to support a new identifier called LSI.

LSIs resemble IPv4 addresses. The choice we made implies that LSIs can be interpreted correctly only by the localhost because they are valid only within local context. This statement generates referral-related problems. The referral problem originates from broken application layer protocols that send IP addresses (HITs or LSIs in the case of HIP) on the wire. When such a protocol sends LSIs on wire, they become invalid or incorrect at the peer host. In this thesis, we study how to solve this problem and we will see that the system-based opportunistic mode could be one possible alternative for solving that.

One aim of this thesis is to analyze the differences and performance between HITs and LSIs, as well as to show an example of the referral problem using LSIs.

Applications the LSI implementation supports are limited to the ones using the ICMP protocol, as well as TCP and UDP transport protocols. Other protocols are out of the scope of this thesis.

3.2.3 Interoperability between IPv4 and IPv6 Applications

The design and implementation adds support for IPv4 applications. As the implementation already supports IPv6 applications, both IP versions will be supported by HIP and, therefore, it is possible to enable communications between different IP versions.

Owing to the transition between both IP versions, IPv4 applications may continue working with IPv6 applications. For example, an IPv4 application on the client must be able to communicate with the IPv6 application on the server, and the other way around.

Moreover, the use of IPv4 applications working over an IPv6 network using HIP, could be a compelling story for HIP. This is because in general IPv6 migration requires dual stacks, that is, a host has both an IPv4 and an IPv6 protocol stack, but HIP uses that in a different manner. At some point, in an IPv6-only world, systems will be able to turn off their IPv4 stack [55]. In a HIP case where there are an IPv4 client and an IPv6 server, for example, the IPv4 address used by the client application corresponds to the LSI, and this LSI would be translated to a HIT by the HIP layer. At this point, we suppose that the network address is IPv4. When the packet arrives to the server the packet is translated until it contains the IPv6 identifiers called HITs. In this case, the difference to a non-HIP environment is that HIP provides two identifiers to support both IP versions. However, without using HIP, the IPv4 and TCP modules detect that the destination socket is an IPv6 one, and converts the IPv4 address into the equivalent IPv4-mapped IPv6 address.

One goal in this thesis is to decide when the use of one protocol is preferable over the other depending on the IP version of the application of destination. Moreover, we want to provide communication between applications which use different IP versions with our implementation.

3.2.4 Supporting Opportunistic Mode as a System Library

As discussed in the background chapter, the dynamic linking library shims system calls. Interposition libraries intercept these calls and change the expected functionality. The LD_PRELOAD environment variable can load the interposition libraries. The current HIPL opportunistic mode library uses this mechanism.

Our proposal is to change this behaviour and include the opportunistic mode during the static compilation, moving the opportunistic library from the user to the system level. The main advantage is that the system-based opportunistic mode does not need extra libraries and is independent of support to individual sockets API functions. Furthermore, this method can make the user experience easier using HIP because the user does not have to configure any environmental variable.

Both approaches do not need to modify the source code of applications thus, the library supports legacy applications.

In addition, the system-based opportunistic mode allows to support networks without PKI. A PKI [20] provides end-to-end security and it is an arrangement that binds user identities with their public keys by means of a Certificate Authority (CA). There is a registration process for the binding which must be carried out by software or under human supervision.

However, HIP does not need a PKI. Since the identity is represented by the public key, any proper protocol able to check that the party owns the private key corresponding to its public key is enough. In our case, this authentication protocol is included during the Base Exchange which creates and negotiates the keys. However, if the peer does not have a PKI, the HIP host must use a non-PKI. The deployment of global PKI infrastructure is virtually impossible, thus the opportunistic end-to-end security, which is based on the concept of Leap of Faith security or weak authentication, is enough for heterogeneous wired and wireless networks.

As there is no flag day where the majority of the hosts will support HIP, the most common scenario would be to establish communication with a non-HIP-aware peer. In this case, HIPL must fallback to a non-HIP communication. The system-based opportunistic mode includes this feature.

One aim of this thesis is to analyze the differences and performance between these two approaches.

Chapter 4

Design

This chapter describes the semantics of the LSI identifier. We discuss the behaviour designed on both client and server side and also the interoperability design between the two IP versions. Afterwards, we introduce a design proposal for an opportunistic mode that reuses the LSI design.

4.1 Local Scope Identifier

This section defines the main features of LSI and explains the different techniques HIPL uses in order to generate this identifier.

4.1.1 Definition

In our design, LSI is independent from the HIT. Hence, an LSI is not derived from HIT. This approach eliminates collision and security problems raised when HIP generates an LSI from a HIT, because an LSI is much shorter than a HIT.

In our implementation, an LSI address is allocated from the 1.0.0.0/8 subnet by default, but the user or administrator can change the prefix dynamically. As initial decision, we chose the prefix 192.168.0.0/24 but due to NAT problems, we moved to the other prefix. These problems arose because IANA allocates the range 192.168.0.0/16 for private-use networks. Consequently, HIP could not differentiate between a NAT address and an LSI identifier. The actual approach continues to generate problems with NATs because the 1.0.0.0/8 address space is unallocated by IANA, thus not registered to HIP. The related problems and possible solutions are discussed in chapter 6.

4.1.2 LSI Generation

When the user or the system starts running `hipd` and runs an IPv4 application, there must exist a mapping between the HIT, the LSI and the IP. The user can execute the mapping manually. Otherwise, `hipd` establishes the mapping automatically. The `hipd` does not send the LSI on wire but it needs to know the corresponding HIT and IP for HIP and ESP-related processing. We must notice that the peer LSI must be unique in the machine, because the host can not have two different peer host names mapped with the same peer LSI. There are different ways for establishing the mapping between the identifiers and the IP:

1. Manually. The `hipconf` tool provides an option which the user uses to maps LSIs manually.
2. DNS interception. The DNS Proxy extension for HIPL includes support for LSI resolution. When the application makes an AAAA record request, the DNS Proxy module returns a HIT as an AAAA record. When the application makes an A record request, the DNS Proxy program allocates and returns an LSI as an A record response.
3. Base Exchange. The first two possibilities show the mapping from the point of the initiator, but the responder must map LSIs. The responder generates an LSI for initiator automatically. The `hipd` handles the automatic LSI generation during the Base Exchange, specifically when the responder receives the I2 packet.

4.2 Packet Processing

The LSI processing consists of inbound and outbound packet handling. The HIP firewall (`hipfw`) handles input and output packet processing for LSIs in HIPL implementation. It should be noted that `hipfw` is actually a multi-purpose daemon that can also handle HIP-based access control (hence the name `hipfw`). Moreover, interoperability with HIPL userspace IPSec needs further work.

Below is an example of a client-server scenario. The systems are running `hipd`, which is responsible of HIP control plane signalling, and `hipfw` daemon. A client and a server are running applications, e.g. netcat [41] which is a networking utility which reads and writes data across network connections, and both of them using TCP as the transport layer protocol. The server starts up a service at port 5555. The client application tries to connect to the server over TCP. Then the server accepts the connection, reads the received data and prints it on the screen.

4.2.1 Output Packet Processing

The output packet processing is illustrated in Figure 4.1 and described below.

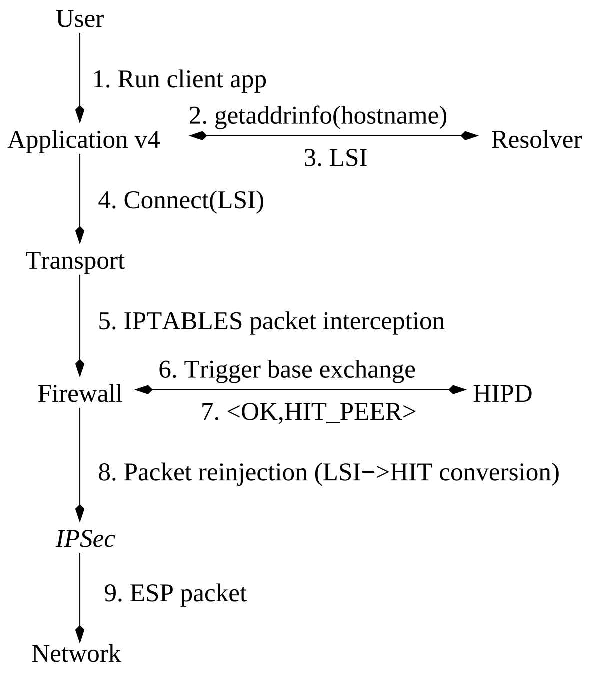

Figure 4.1: Client-side schema

1. The user runs a client application.
2. The application queries the resolver in order to translate the hostname.
3. The resolver returns the LSI assigned to this hostname.
4. The application contacts the peer LSI by calling `connect()`.
5. The packet traverses through sockets and TCP layers until `hipfw` captures it in the network layer owing to an iptables rule.
6. The `hipfw` checks whether the local database already contains the peer HIT corresponding to the peer LSI. If it does not find the entry, `hipfw` triggers the Base Exchange through `hipd`.
7. When the Base Exchange finishes correctly, `hipd` communicates the peer HIT and LSI to `hipfw`.
8. The `hipfw` can build the new packet, translating the LSI pair to the HIT pair. In addition, `hipfw` caches the HIT-LSI mapping for future packets. In this

way, there is no triggered second Base Exchange for the same connection and the process speeds up. Then, it reinjects the packet again to the network stack with HITs converted to LSIs.

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

4.2.2 Incoming Packet Processing

Figure 4.2 shows inbound packet processing. We focus mainly on the inbound packets that the server processes.

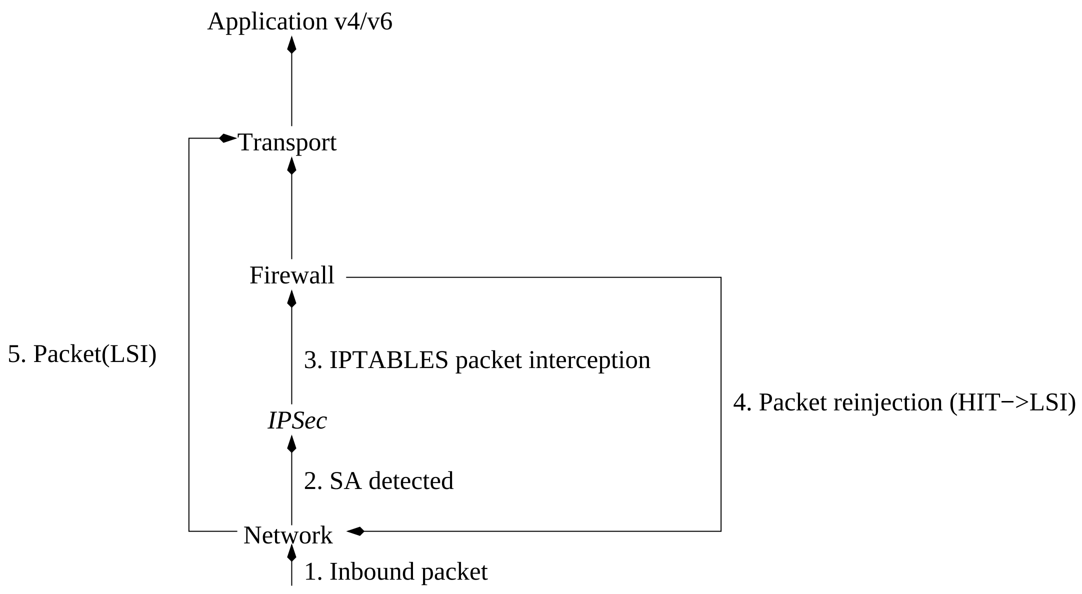

Figure 4.2: Server-side schema

This scenario is handled as follows:

1. The server receives a packet.
2. A SA entry in the SADB demultiplexes the SPI field of the ESP header. The ESP module decrypts the packet and converts the IPv4/v6 header to an IPv6 header containing the HITs.
3. An iptables rule in `hipfw` captures the packet. At this point, the `hipfw` checks whether LSI processing is necessary or not, as we describe in more detail in the following section.
4. If `hipfw` applies LSI processing, it reinjects the packet to the stack again, but this time with an IPv4 header containing LSIs.

5. The packet with the LSIs arrives to the stack and the stack delivers the packet to transport and application layers.

4.2.3 Interoperability IPv4 and IPv6

As we have explained in subsection 2.2.3, HIP supports IPv4 and IPv6 interoperability. This feature occurs during inbound packet processing. The problem is how `hipfw` manages to demultiplex an incoming packet to an LSI or a HIT. An operating system process is identified by an IP address, a port number and the network protocol. We must notice that a single port can be occupied by the same application but with different address family (or protocol), as Figure 4.3 illustrates.

Following policy considerations specified in [?], we design a local policy in order to decide when a HIT must be translated to an LSI. The `hipfw` demultiplexes to an LSI if no process is listening to IPv6 on the particular port number. In other words, HITs are preferred over LSIs because they do not have the LSI disadvantages, as e.g. non-routable or callback problems and furthermore, we expect IPv6 to dominate in the future. This logic applies only to TCP and UDP and not to ICMPv4 and ICMPv6. This is because the ICMP protocol is part of IP where ICMP messages are usually generated as a response to errors in IP datagrams.

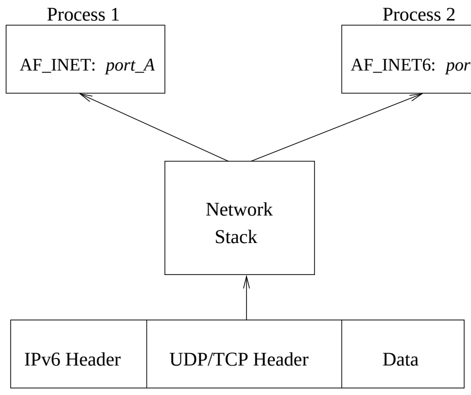

Figure 4.3: UDP and TCP headers contain the destination port number. The hipfw decides which IP protocol to choose depending on the absence or presence of the listening process in the IPv6 or IPv4 address family

4.3 Alternative Design for LSIs

Our design is centered around the idea of capturing LSI packets with iptables rules in `hipfw` and transforming them only when it is necessary. This approach overloads `hipfw` work and adds some problems, such as the increased latency per each packet because of the processing time.

The alternative design is to manage LSIs using IPSec SAs instead of the iptables rules. An SA pair could be associated with two SPIs, two HITs and also two LSIs. Furthermore, there must exist an SP that matches packets with LSIs or an LSI prefix.

We discarded this design because current Linux IPSec cannot associate both an LSI pair and a HIT pair into a single SPI.

4.4 Opportunistic Mode Design

The opportunistic mode was already supported in the HIPL implementation as a user library. Our implementation goal is to move the opportunistic mode library to the system level by reusing the LSI design for supporting the opportunistic mode. The main advantages of this approach are:

- Solving opportunistic library bugs. chapter 6 examines the supported system calls. But, we can already comment that supporting all socket calls is cumbersome.
- Solving library dependency problems. There are some problems with chaining of LD_PRELOADed applications. All chained libraries must support chaining properly or otherwise the application's network connectivity is broken.
- Supporting more applications. The system opportunistic mode supports more applications because it is independent from the operating system and only depends on the transport layer protocols supported by `hipfw`.

The library does not translate raw sockets or sockets already bound to HITs and it can translate IPv4 or IPv6 addresses to HITs. As the address size for the different IP versions is different, it creates for each IP-based socket a completely new HIT-based socket [36].

The system-based opportunistic mode design is described below. We must differentiate two possible scenarios, depending on the peer capability to support HIP. However, both scenarios have the first steps in common, before the peer answers the Base Exchange. We must notice that the local or peer address can change during the communication, requiring a new Base Exchange.

Figure 4.4 shows a first scenario where the peer host is HIP enabled. We describe the outbound packet processing below.

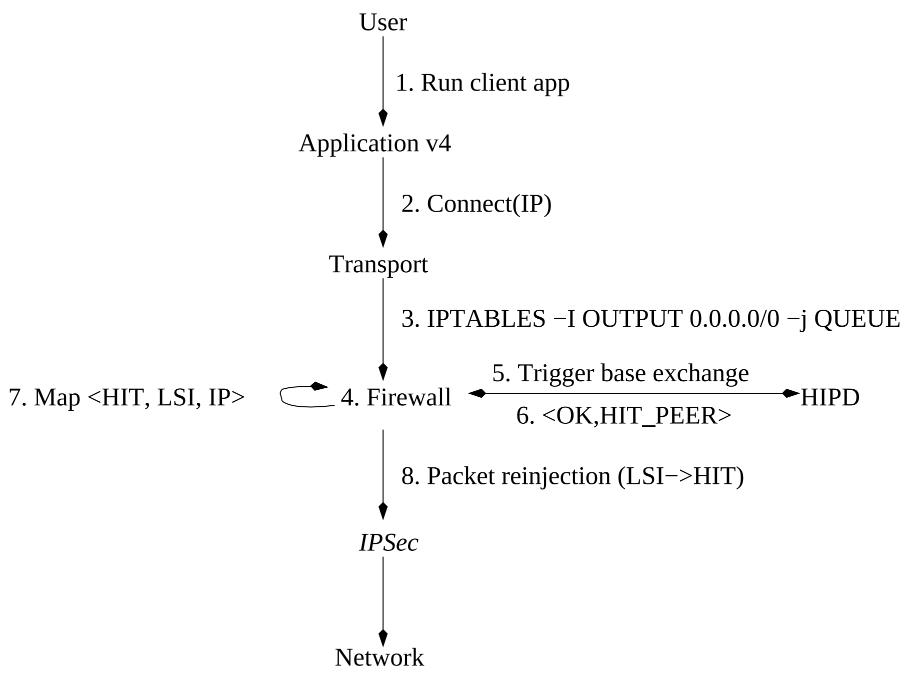

Figure 4.4: Opportunistic design: Peer HIP-aware

1. The user or administrator runs an application.
2. The application layer calls *connect(IP)*.
3. The `hipfw` intercepts the packet.
4. The `hipfw` checks its local database. If the destination address is already associated with a HIT, `hipfw` goes to the final step.
5. The `hipfw` triggers the Base Exchange in opportunistic mode with a HIT peer empty value. In this step, we must notice Base Exchange can use the TCP extension discussed in chapter 2.
6. Base Exchange is completed successfully and returns the peer HIT value.
7. The `hipfw` maps in its local database the peer HIT, the peer LSI and the peer address.
8. The ESP stack encrypts the packet and sends it to the peer host.

Figure 4.5 shows a second scenario where the peer host is not HIP-enabled. We describe the outbound packet processing below.

1. The user or administrator runs an application.

2. The application layer calls *connect(IP)*.
3. The `hipfw` intercepts the packet.
4. The `hipfw` checks its local database. If the destination address is already associated with a HIT, we skip to the final step.
5. The `hipfw` triggers the Base Exchange in opportunistic mode with a HIT peer empty value. In this step, we must notice Base Exchange can use the TCP extension discussed in chapter 2.
6. Base Exchange is not established, therefore the peer does not support HIP.
7. In order to avoid HIP-capability detection the `hipfw` maps the IP address in its local database, indicating with a flag that the peer is not HIP-enabled.

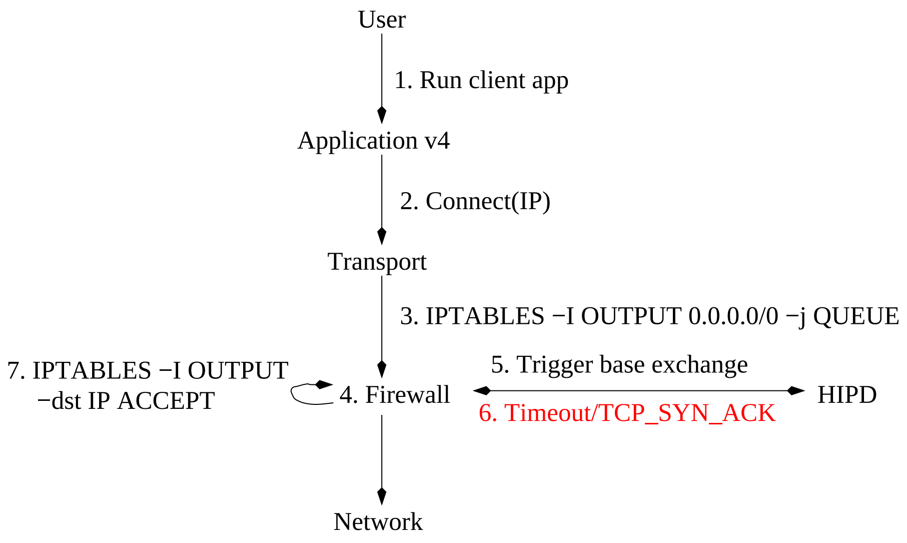

Figure 4.5: Opportunistic design: Peer not HIP-aware

We must clarify the second step in Figure 4.5 and Figure 4.5, where the IP is referring to an address different from a HIT or an LSI. Our implementation of the opportunistic mode allows to fallback to non-HIP communications if the peer does not support HIP, but using the LSI identifier, we assume that the peer supports HIP, and, if the connection with HIP is not possible, the connection should be rejected.

At the input side, the receiver of a packet should analyze the source and destination identifiers to decide when to apply LSI transformation.

Chapter 5

Implementation

Our LSI implementation is based on the HIPL implementation. Originally, HIPL supported only kernel-based IPSec. The IPSec implementation of HIPL was supported in the userspace by the end of this thesis. This feature is relevant for the LSI module because both implementations modify `hipfw` component. Then, there is a need to integrate the two extensions.

HIPL is divided into five main components. The HIP daemon (`hipd`), the HIP configuration tool (`hipconf`), optional HIP socket handler, the BEET extension for IPSec and the optional firewall (`hipfw`). We have changed the `hipd`, `hipconf` and `hipfw` components.

In this chapter, we show different scenarios which illustrate the interaction between the different software components using sequence diagrams. In order to simplify, the full execution trace is not shown. Instead, we focus on the most relevant functions. Furthermore, the reader should not be confused if he or she tries to find the corresponding functions from the code because we have used shorter names here. The reader must also bear in mind that function parameters have been omitted for simplicity.

5.1 Local Scope Identifier

This section studies the data structure for LSIs in HIPL implementation. Secondly, we explain implementation details about the different techniques HIPL uses in order to generate this identifier and to link it to a virtual interface.

5.1.1 Data Structure for LSIs

We introduce a new identifier in the HIP protocol. As shown in Figure 5.1, an LSI is based on the `in_addr` structure.

```
typedef struct in_addr hip_lsi_t;
```

Figure 5.1: The lsi structure

This structure handles Internet IPv4 addresses and it has a variety of representations in the different systems because of a `union`. The common field is `s_addr`, a 4-byte number where each byte represents an IP address digit.

5.1.2 LSIs on the Virtual Interface

In HIPL, the host can have up to four HITs with each associated with the corresponding LSI. The LSIs and HITs are contained in *dummy0* virtual interface to provide routes to the linux networking stack. There are three ways of configuring it:

1. Ioctl calls made (as ifconfig handles it).
2. Netlink calls (as iproute handles it).
3. Through the /proc filesystem.

In the LSI implementation, we used the netlink library to populate the device with the respective LSIs and the ioctl library for removing them. It is important to notice that although netlink supports more than one IPv4 address without creating aliases, ioctl does not support it. The example in Figure 5.2 shows the difference between the two commands for creating a multihomed device.

```
ip addr add <networkaddress>/<prefixlength> brd + dev <device>

ifconfig <device>:<aliasnumber> <address> netmask <netmask> up
```

Figure 5.2: Command line syntax for iproute and ifconfig to bring up a device with an address

5.1.3 LSI Generation

We already explained the three possible ways LSIs are generated in chapter 4. Here we will dwell on the implementation details.

Manual configuration

The `hipconf` tool is a command line interface for `hipd`. This tool parses the commands to the `hipd` and it has been extended in order to support LSIs. The user can specify the LSI in the hipconf command, otherwise the daemon will generate the LSI on the fly, as shown in Figure 5.3.

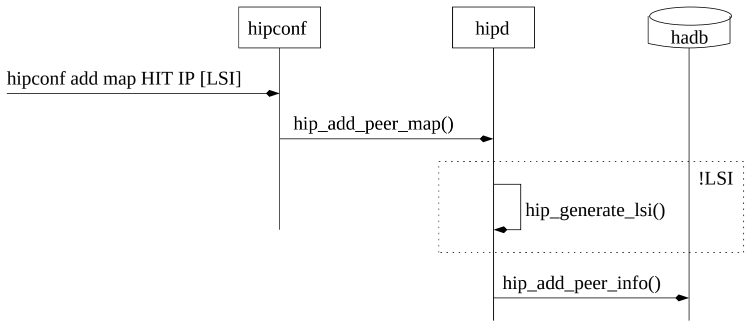

Figure 5.3: Hipconf add map sequence diagram

An example of how to use the `hipconf` tool for this purpose is shown below in Figure 5.4.

```
>hipconf add map 2001:001b:2b1d:55f7:798a:f476:af0a:f826
 128.214.114.58 1.0.0.7
>hipconf add map 2001:001b:2b1d:55f7:798a:f476:af0a:f826
 128.214.114.58
```

Figure 5.4: A typical execution of the command `hipconf add map`

Automatic configuration

Automatic configuration consists of creating the mapping between the peer identifiers without using the `hipconf` tool. For this purpose, the user or administrator must add the HIT and optionally the LSI to the HIP resolver file and the IP address in the hosts file. The automatic configuration was already working for IPv6, although the LSI information was not read from the resolver. Currently this functionality is supported. And we added too the automatic configuration for LSIs which is depicted in Figure 5.5.

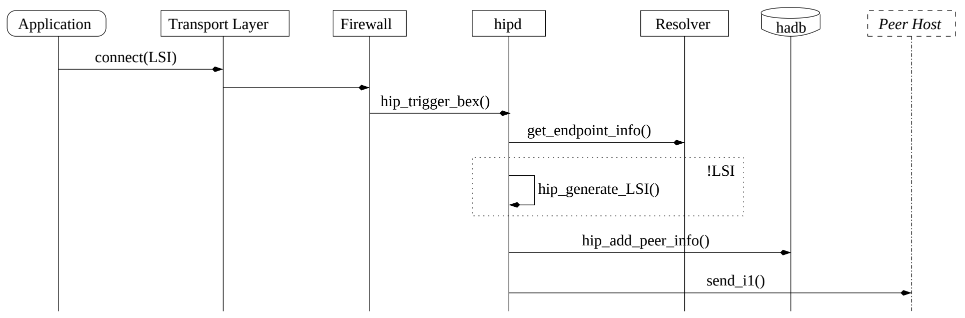

Figure 5.5: Automatic peer information configuration sequence diagram

Initially, the user runs a legacy IPv4 application using the LSI identifier, whose packets are captured by the firewall. The firewall detects that the Base Exchange was not established for the peer LSI. Therefore, it triggers the Base Exchange and the `hipd` calls the resolver in order to look up the peer host information. Once the `hipd` has the information, this is added to the HADB and afterwards the Base Exchange is set up with the peer host.

DNS Proxy resolution

The DNS Proxy extension for HIP provides HI/HIT-based look-up service for the end-host. Unlike the rest of the implementation, this extension is implemented with Python. It intercepts DNS requests from an end-host and returns a HIT instead of an IP address if it finds one.

5.1.4 Modified Database Structures

HIPL has mainly two databases which store information about the local host identities and the current associations with the different peers. In order to support LSIs, we modified these two data structures.

Host Identity Database

The Host Identity Database (HIDB) contains the localhost Host Identities and related information, as shown in Figure 5.6. The HIDB contains four local HITs/HIs and also LSIs with each HI tied to an LSI.

`hipd` initializes the database upon startup.

```
struct hip_host_id_entry {
        struct hip_lhi lhi;
        hip_lsi_t lsi;
        struct hip_host_id *host_id; /* allocated dynamically */
        struct hip_r1entry *r1; /* precreated R1s */
        struct hip_r1entry *blindr1; /* pre-created R1s for blind*/
        /* Handler to call after insert with an argument, 0 if OK*/
        int (*insert)(struct hip_host_id_entry *, void **arg);
        /* Handler to call before remove with an argument, 0 if OK*/
        int (*remove)(struct hip_host_id_entry *, void **arg);
        void *arg;
};
```

Figure 5.6: HIDB record structure

Currently, `hipd` generates the local LSIs statically when it starts running. As shown in Figure 5.7, `hipd` stores LSIs statically in an array and copies them to the HIDB with the corresponding HIT. Furthermore, the first LSI ("1.0.0.1") is always the default LSI corresponding to the default HIT.

```
static char *lsi_addresses[] = {"1.0.0.1","1.0.0.2",
                                "1.0.0.3","1.0.0.4"};
```

Figure 5.7: HIDB static initialization

Host Association Database

Independently of the HIDB, there is a second database called Host Association Database (HADB). Its objective is to record HIP-related state information about peer hosts. The database is indexed by the pair of local and peer HIT. The LSI extension includes the related lsi pair, composed by the local and the peer LSIs, to the entries in the database.

5.2 Packet Processing

The `hipfw` daemon already handles data plane interception and it was therefore a natural place also to implement LSI processing. Furthermore, in order to preserve modularity, the LSI `hipfw` related functions are in a separated module, which contains functions that handle incoming and outgoing LSI packets and the ones that reinject the packet again to the network stack. The LSI module of `hipfw` caches information about the peer HITs and the peer LSIs. `hipfw` uses this information in order to trigger the Base Exchange or not. In case the Base Exchange is already established, `hipfw` gets the correspondent HIT pair from the database.

Next, we present some sequence diagrams from the point of view of a client-server model.

5.2.1 Output Packet Processing

When a datagram traverses through the network stack, `hipfw` captures it with the following rule:

```
iptables -I OUTPUT -d 1.0.0.0/8 -j QUEUE
```

The `hipfw` output handler chain consists of hip, esp and lsi handlers amongst some other handlers. As shown in Figure 5.8, the *lsi* module checks the state of the Base Exchange in the firewall database (*fwdb*). If the answer is negative, it triggers the Base Exchange. Otherwise, `hipfw` reinjects the packet. The reinjection process consists of replacing the IPv4 header with an IPv6 header where the destination and source addresses correspond to the HITs associated to the LSIs. `hipfw` handles the reinjection using raw sockets.

5.2.2 Incoming Packet Processing

In this subsection, we focus on inbound packets at the server side. As well as during the outbound process, `hipfw` captures incoming HIT-based packets using the following rule:

```
ip6tables -I INPUT -d 2001:0010::/28 -j QUEUE
```

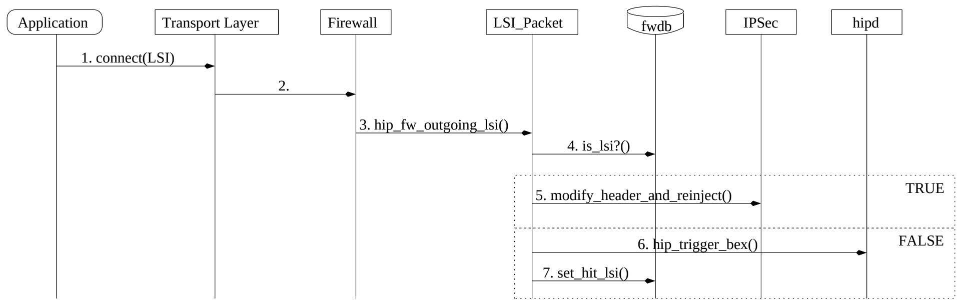

Figure 5.8: Outgoing sequence diagram on the client host

The `hipfw` input chain contains the same handlers as the output chain. We have been inspired by the mechanism used by *netstat* application. `hipfw` looks up the transport protocol-related files placed in the directory */proc/net.* If `hipfw` finds the destination port in the file "/proc/net/tcp6", it does not change the packet, otherwise `hipfw` changes the IPv6 header to an IPv4 header which includes the LSIs and reinjects the new packet to the stack. Then, the application receives the reinjected packet. Because reinjection with "mangle" does not support interfamily transformations, the kernel queues the packet in the outgoing queue instead of the inbound one, consequently the process requires to check the identifiers in order to decide the right direction. The incoming scenario is depicted in Figure 5.9 where we assume that the transport protocol is TCP.

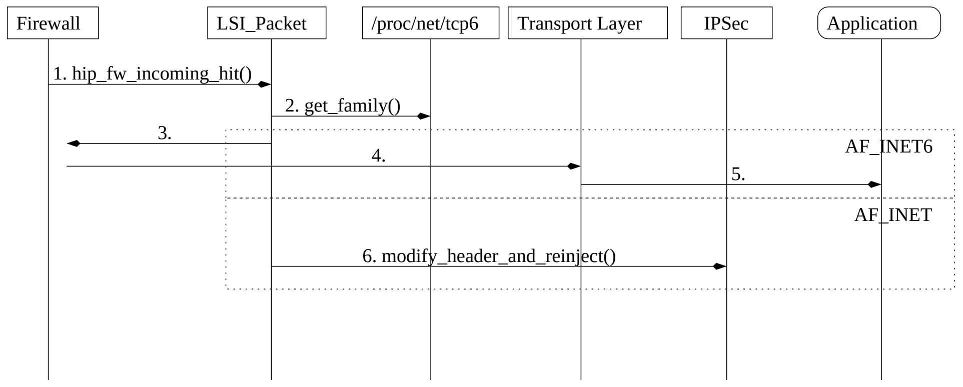

Figure 5.9: Incoming sequence diagram on the server host

5.3 Protocol translation mechanism

As we already explained in the previous section, the packet suffers a translation mechanism from IPv4 to IPv6. In this section, we dig into this process, showing the consequent header transformations.

The change of IP header affects the transport layer protocols and ICMP. Therefore, the unique and main change which suffers the header of these protocols is the recalculation of the checksum field, since these header protocols are based on an overlay header which contains the IP addresses.

As we are using raw sockets, with the outgoing packets is the kernel which builds the new IP header. Then, the transport layer changes are enough. However, we had to build the IPv4 header for incoming packets whose destination application only supports IPv6. The reason is that `bind` function can not bound the source address to the socket because this IP address is from the peer. In this case, we further simplified the process not translating extensions or options. For ease of implementation, the header length and total length fields are re-computed. The address fields are set according to the address mapping between LSI and HIT. We directly copied to TTL the value from the Hop Limit field in IPv6.

Unlike the above-mentioned fields, there are some fields where there is not an equivalence manner to express the information. This is the case of the quality of service or flow-related fields. This is because the semantics in the IPv6 context for the quality of service differ from the ones used by IPv4 [35]. Then, we decided to set these fields to a default static value, e.g. the value of zero for the type of service field in the IPv4 header.

As the MTU value for the dummy0 interface is limited to 1280 bytes, there is no fragmentation and in consequence we did not have to deal with the translation of fragments.

Chapter 6

Analysis

Firstly, this chapter presents the performance evaluation configuration. Secondly, we analyze the results with different charts and tables for TCP and ICMP protocols. In addition, we study other related problems that we found during the analysis and testing processes. We introduce these topics in the following order: LSI address space, Maximum Transfer Unit, referral problem and LSI compatibility with other extensions.

6.1 Performance Evaluation Configuration

In this section we present configuration of the performance measurements in different scenarios. Furthermore, we explain the test platforms and software used.

6.1.1 Test Platforms

We performed our measurements on two machines following the client-server schema. The client was the initiator and the server the responder. We used the *hipl–userspace–2.6–patch-1817* version in both machines for testing LSI, HIT and Opportunistic System Based scenarios performance. We connected the machines using a direct 1 Gbit link. The MTU default value in the dummy interface was 1280 bytes. Below in Table 6.1 we describe the hardware and operative system characteristics on both machines used during the tests. The network layer is always IPv4 and the upper layer address is IPv4 when it is applicable.

6.1.2 Test Software

We used *Iperf version 2.0.2* for the TCP throughput testing tool [18]. For a TCP connection, *Iperf* shows the bandwidth and throughput, and by default sends an

Initiator	Responder
Ubuntu Hardy 8.04 64-bits Kernel 2.6.25.8 (64-bits) and BEET PATCH	Ubuntu Hardy 8.04 64-bits Kernel 2.6.25.8 (64-bits) and BEET PATCH
2048 KB	4096 KB
Intel(R) Core(TM) 2 CPU T6400 2.13GHz	2 x Intel(R) Core(TM) 2 Duo T7300 2.00GHz

Table 6.1: System configuration of the testing environment

8KB array for 10 seconds. In addition, we tested the performance of the system call `connect()` with the application *conntest-client-hip* which is included in *hipl–userspace–2.6–patch-1817*. In both test scenarios, we used the tool *hipconf rst all* in order to reset the established connections. Therefore, we can compare the increase of time when `hipd` has to establish a new connection with a peer and the time needed during normal traffic, when both hosts have already set up the connection.

We tested the ICMP and ICMPv6 protocols using the *ping* and *ping6* tools [47]. The *ping* program sends an *ICMP echo request* message and expects an *ICMP echo reply* to be returned. In addition, it can be used to measure the Round-Trip Time (RTT) to a host. We show the arithmetic mean and the standard deviation for the RTT values captured. We show a testing scenario of 20 samples for LSI, HIT, System Based and User Based Opportunistic Modes and finally a plain ICMP scenario.

6.1.3 Test Procedure

We used the following test plan to conduct our measurements. We tested TCP and ICMP running `hipd` and `hipfw` on both machines. A desktop computer acts as the *initiator* and a laptop as the *receiver*, where each measured result consists of 20 samples. Firstly, we focus our tests in the throughput during data traffic using TCP over ESP where we compare the performance of LSIs, HITs and the user-based opportunistic mode. Finally, we tested ICMP and ICMPv6 protocols with LSI and HIT identifiers, using the opportunistic mode and without a HIP environment.

6.2 Results and Analysis of the Performance Measurements

This section shows the results that we obtained with different scenarios following the configuration that we explained in the previous section. We focus on TCP studying its throughput and *connect* performance. Finally, we move to the ICMP results.

6.2.1 TCP Throughput

In Figure 6.1, we illustrate the average and the standard deviation of the throughput of data during HIP traffic. The x axis presents different modes of HIP and the y axis displays the throughput of the communication in Mbits/s.

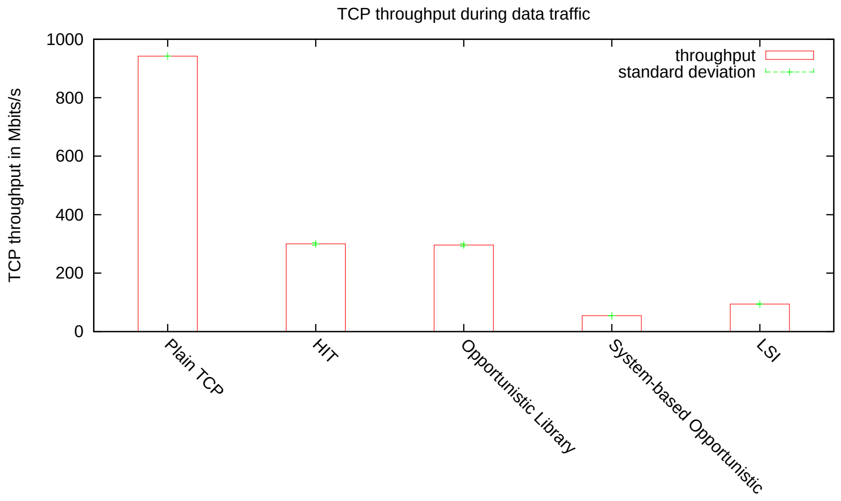

Figure 6.1: Throughput of different HIP modes

The chart shows that the average value for LSIs is 94 Mbits/s with a standard deviation value of 1.3 Mbits/s. The average throughput for HITs is 300 Mbits/s with a standard deviation of 5.1 Mbits/s. In the opportunistic mode scenario, the average of the user-based mode is 296 Mbits/s with a standard deviation of 5.0 Mbits/s and with the system-based mode 54 Mbits/s with a standard deviation 0.7 Mbits/s. A plain TCP connection has an average value of 942 Mbits/s with a standard deviation of 0.3 Mbits/s. As the measurements show, a communication under HIP achieves the best results when it uses the HIT identifier, followed by the user-based opportunistic library, although the difference in throughput between HIT and library-based scenarios is almost non-existent. Furthermore, LSI has almost twice as good performance as the system-based mode. The LSI identifier and system-based modes are three times slower than HIT and library-based modes in processing and delivering data.

6.2.2 TCP connect

In Figure 6.2, we illustrate the average and the standard deviation of the system call `connect()`. In our testing environment, it is the program *conntest-client-hip* that calls this function. The x axis represents different modes of HIP and the logarithmic y axis displays the time to complete the Base Exchange and the TCP handshake in ms after the application calls the system call `connect()`.

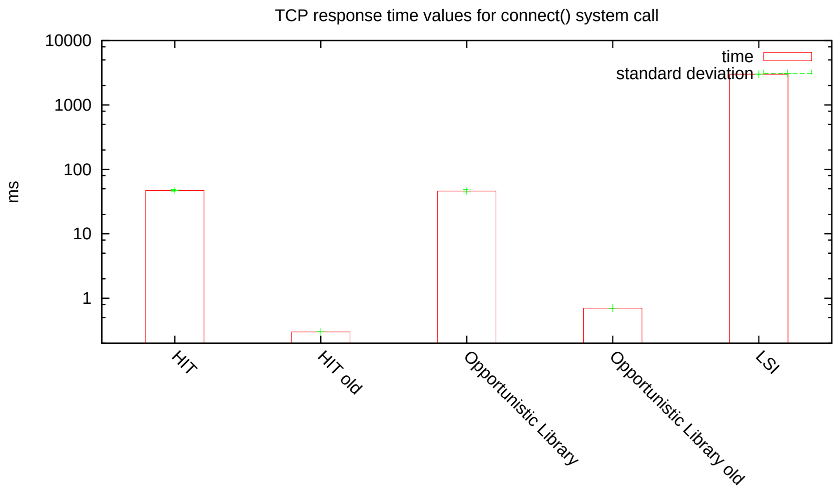

Figure 6.2: TCP *connect()* performance

The results we obtained, without the logarithmic scale on the y-axis, show that the average value for LSIs is 3000 ms with a standard deviation value of 2 ms. The average time for HITs is 47 ms with a standard deviation of 2 ms. This time consists of the time to complete the Base Exchange and the TCP handshake. In the opportunistic mode scenario, the average of the user-based mode is 46 ms with a standard deviation of 3 ms. Each of these measurements set up the connection. As we can observe with the following measurements, this process highly increases the response time of the system call. Once the state is established, the average value for HITs, called HIT old in the chart, decreases until 0.3ms with a negligible standard deviation. The same goes for the opportunistic user-based mode, named Opportunistic Library old in the chart, where the second scenario shows an average value of 0.7 ms.

This test shows that the LSI has a surprisingly bad performance with a constant 3 seconds delay. This is related with the initial value of TCP's Retransmission Timeout (RTO) timer. In a Linux operating system, the initial RTO is 3 seconds by default. When the SYN packet does not arrive to the server or the SYN ACK to the client, it causes a timeout with the initial RTO value [40] [59]. The effect of RTO in throughput [3] is:

- The sender transmits few packets til it expires.
- Sender shall restart in slow start mode with the initial value of the congestion window (cwnd).

In the HIPL context, this is due to LSI implementation dropping data packets until both peers establish the Base Exchange. Therefore, there are multiple packet losses which cause the RTO. The [3] introduces how to avoid RTO caused by multiple packet losses with the modified fast recovery algorithm of TCP New Reno or selective acknowledgement (SACK) option and it explains how to avoid RTO with retransmitted packets. Both methods should be applied in our context in order to avoid RTO.

On the other hand, this behaviour is not present in the library implementation because it blocks the socket calls until the host establishes the Base Exchange with the peer. Then, no packets are lost [36].

6.3 ICMP

The results of the different tested scenarios are shown in Table 6.2.

The mean RTT is 0.261 ms without using HIP in our testing scenario, where the deviation of the sample is 0.13 ms. The mean RTT using LSIs is 0.658 ms with a standard deviation of 0.044 ms. The mean RTT using HITs was 2.424 ms with a standard deviation of 0.147 ms. The system-based opportunistic mode has a mean RTT value of 1.119 ms with a standard deviation of 0.097 ms.

Scenario	Average	Deviation
Plain ICMP	0.261 ms	0.13 ms
HIT based	0.319 ms	0.030 ms
LSI based	0.658 ms	0.044 ms
Sys-based Opp. Mode	1.119 ms	0.097 ms

Table 6.2: RTT values in a HIP and plain communication

Results show that a HIP communication increases the RTT. As we can observe, the RTT performance is twice as good with HITs instead of LSIs. Furthermore, the system-based opportunistic mode is almost four times higher than the HIT identifier and twice the LSI average.

6.4 List of Supported IPv4 Applications

We tested ping, netcat, ssh and ftp with LSIs during the development of the LSIs. We used the LSI directly in the application layer because DNS Proxy support for LSIs is not yet available.

6.5 LSI Address Space

We decided to use a fixed address space in the range 1.0.0.0/8. This address space is unallocated by IANA [16] which means it can not be used by NATted networks or the LSI identifier. However, we can take into account that the use of LSI is local in contrast to NAT addresses which go on the wire. Below we present the potential problematic scenarios.

Imagine a host with an application calling *connect()* with an LSI. Moreover, the host also has assigned an IP address which corresponds to a private IPv4 address in the LSI range. The `hipfw` will capture the packet because of the iptables rule and will change the LSIs to the HITs. This scenario does not present any problem unless the destination address, which goes on the wire, is also in the LSI range. In this case, our design creates an infinite loop, because `hipfw` translates the LSI to a HIT and afterwards to an IP equal to the LSI prefix, thus `hipfw` captures the packet again by the LSI output rule in iptables.

On the input side there is no problem. The packet just arrives to an interface such as *eth0*.

We present three possible solutions for avoiding LSI collisions with NAT namespaces. Firstly, we propose the static allocation scheme, reserving the 1.0.0.0/8 address space for HIP protocol. The second alternative is to use 1.0.0.0/8 and allow IANA to allocate each address individually. The third alternative is to perform the LSI to HIT translation higher in the TCP/IP stack, consequently a mechanism realizes the conversion before the packet reaches the kernel routing table. This last solution implies kernel changes or interposition libraries.

6.6 Maximum Transfer Unit and LSIs

The maximum size of a packet is theoretically determined by the IP protocol. Specifically, the maximum size of an IPv4 datagram is 65535 bytes. This is because of the 16 bits value for the `len` field in the IPv4 header. The maximum size of an IPv6 datagram is also 65575 bytes because of a 16 bits field in the header. However, layer 3 in the OSI model defines the Maximum Transfer Unit (MTU) which represents the maximum size of an IP datagram that the network device can handle without fragmentation. The common MTU for an Ethernet device is 1500 bytes, and it is the value used by default with the *dummy0* virtual interface. When the frame size is bigger than the MTU, the packet must be fragmented.

We observed a problem using LSIs when transmitting a file with size of 1408 bytes or larger. As an example, let us consider an application that sends a file of 1500 bytes. The **`hipfw`** captures a 1500 bytes packet where:

1. The IP header is 20 bytes, as we do not set up any option.
2. The IP payload is 1460 bytes, where 32 bytes correspond to the TCP header and the remaining 1428 bytes to the data.

As shown in Figure 6.3 the transformation process from IPv4 to IPv6 increases the size of the packet 20 bytes. After the IP translation is performed, we also add the ESP header to the packet, further increasing the final packet length.

The change of the MTU value allowed us to receive the total amount of data that the client requested in a passive FTP connection where the client was using HITs and the server LSIs.

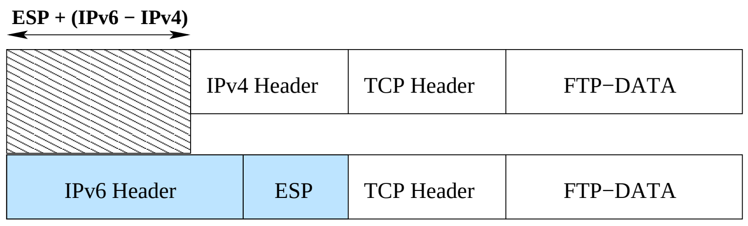

Figure 6.3: LSI-to HIT conversion and effects on MTU

6.7 TUN/TAP mechanism

Instead of using a virtual interface *dummy0*, we could use a TUN virtual tunnel device where the outgoing LSI packets can be redirected. The TUN device makes available the received packets for the `hipd` through the /dev/net/tun device. When a new packet is read from the virtual device, the LSI can be translated to the HITs and afterwards transformed into an ESP packet. The advantage of this method is that the userspace locates the buffer where the virtual device queues the packets, thus the buffer can be larger than in the kernel space. Afterwards, we can continue using raw sockets for packet reinjection into the IP stack. TUN/TAP is also more portable.

6.8 The referral problem

Currently, IP addresses are used in different ways by different applications. Between the possible categorizations [7], we focus on the following:

- Callbacks. The application at the localhost retrieves the IP address of the peer and the peer uses it to later communicate with the same localhost.
- Referrals. In an application with more than two hosts, host B obtains the IP address of host A and passes that to host C. After that, host C uses this IP address to communicate with host A.

The deployment of applications carrying IP addresses in the data stream creates some problems in NATted environments as well as with LSIs. Some examples of these applications or protocols are FTP or Simple Network Management Protocol (SNMP) MIBs for configuration [11]. In the next subsection, we concentrate on the FTP case.

6.8.1 FTP and Referrals

File Transfer Protocol (FTP) [51] is a network protocol used for storing and retrieving files over TCP connections. FTP uses two separate TCP connections for communication, one for data and one for control. The FTP server listens by default on port 21 for the control channel, which transfers FTP requests and replies. On the other hand, the FTP server listens by default on port 20 for the data channel, which transports files. There are two types of data transfers:

1. Active. The client specifies to the server the IP address and port number where the server should connect back. The server port is 20. The client program sends the `PORT` command to the server specifying the IP address and port number where it should connect back. If the client host is using an IPv6-enabled FTP, the command is EPTR.
2. Passive. The client asks the server for the IP address and port number where it can connect and receive the data. The client program sends the `PASV` command

to ask the server which IP address and port it must connect to. If the server host uses an IPv6-enabled FTP, the command is EPSV. This mode is usually the one used by default in web browsers.

The FTP protocol uses both modes including the address and port within the protocol, creating the callback problem. In particular, the FTP PORT command and the PASV responses include the IP address in ASCII in the FTP control packet payload [23]. We found in the LSI scenario that the application treats the LSIs as IP addresses, and as we have mentioned in this thesis, this identifier has a local scope, meaning initiator and responder can not have the same LSIs. Thus, the peer host application has no system context to resolve the LSI back to a HIT or an IP address.

We studied referrals with FTP using *hipl–userspace–2.6–patch-1661* version. Table 6.3 shows that when the application uses HITs on both sides, there is full support for both FTP modes. We explain below the scenario when the application uses LSIs on both sides.

Client	Server	Active	Passive
HIT	HIT	OK	OK
HIT	LSI	KO	OK
LSI	HIT	KO	KO
LSI	LSI	KO	KO

Table 6.3: Results obtained using *lftp 3.6.1* on the client side and *proftpd 1.3.1* on the server side

We now show an example where HIP fails due to the referral problem. We want to set up a HIP-enabled FTP session between two hosts. The FTP server uses *Proftpd* version 1.3.1 and the client uses *lftp* version 3.6.1. We show an output example when we run the client and try to list the directories of the peer machine.

```
ftp -v 1.0.0.7
Connected to 1.0.0.7.
220 ProFTPD 1.3.1 Server (ProFTPD Default Installation) [1.0.0.1]
Name (1.0.0.7:tfinez):
331 Password required for tfinez
Password:
230 User tfinez logged in
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
500 Illegal PORT command
ftp: bind: Address already in use
ftp> passive
```

```
Passive mode on.
ftp> ls
227 Entering Passive Mode (1,0,0,1,213,223).
ftp: connect: Connection refused
```

As we can see, we obtain a control connection to the server because the connection establishment is done via TCP. This results in error 500 and we solve it using passive mode. Afterwards, when we retry the `ls` command, the server refuses the connection. As we can see, the server address is `1.0.0.1`. This is the default LSI identifier value of the server, but in the client context this LSI is its own LSI and not the server one, as we show with the netstat output at the client:

```
Proto Recv-Q Send-Q Local Address    Foreign Address     State
tcp        0      0 1.0.0.1:46191      1.0.0.7:21      ESTABLISHED
```

This is the netstat output at the server:

```
Proto Recv-Q Send-Q Local Address    Foreign Address     State
tcp        0      0 1.0.0.1:21        1.0.0.5:46191   ESTABLISHED
```

As we can observe, there is no relation between the LSIs. Therefore, when the LSIs are passed on the protocol payload, we lose the control to determine the source and destination addresses.

It could seem odd that the case where HITs are used in the client and LSIs in the server works in passive mode. The explanation is that the server is responding with the EPSV command where only the port is specified and not the server address. The active mode in this scenario does not work because the server does not support the IPv6 network protocol, thus the client receives the error 522. The reason could be because the server receives the EPTR command with the HIT and it does not know how to manage the IPv6 address because the server is bound to an IPv4 address.

In addition, FTP can use referrals instead of callbacks. This feature also represents a problem, although referrals with FTP are rarely used. A possible solution is that `hipfw` can dynamically modify the contents of the control connection, rewriting the LSIs in the packet with the right LSI pair in the host. Otherwise, the user or administrator can use a similar application, e.g. *sftp*. This application is the secure form of FTP. It is an FTP based on ssh. As the last option, the user or administrator can use an IPv6 FTP server because the HIT identifier works properly with both modes.

6.8.2 Solution for FTP using LSIs

As we pointed out, one solution is to modify the LSI contained in the payload by the one that has a context in the local machine. We already explained that an incoming packet is decrypted and afterwards the packet containing the HITs is captured by a rule in `hipfw`. If the destination application is IPv6 there is no problem, otherwise we need to verify when the application protocol is FTP. In this case, not only the `hipfw` must translate the IPv6 header to an IPv4 one, containing the LSIs, but also the `hipfw` must check the payload in order to change the LSI to the peer LSI value represented locally according to the corresponding ESP tunnel.

6.9 LSI Compatibility

The LSI module must be compatible with the rest of the extensions that include the HIPL implementation. This section focuses on the most interesting compatibility issues related with this master thesis: normal firewall access control and userspace IPSec.

6.9.1 Normal Firewall Access Control

The LSI design is based on capturing and queuing packets that contain LSI addresses to change this identifier by its correspondent HIT. The `hipfw` carries out this capture process with new rules defined with iptables. Basically, there is a rule for outgoing packets specifying the LSI pattern and another rule for incoming packets specifying the HIT pattern. The input rule that captures HITs, is compatible with `hipfw` access control because the LSI module modifies the packet only when the destination application only supports IPv4. Otherwise, `hipfw` processes the packet and the behaviour is the same as before adding the LSI support. On the other hand, the output rule affects only LSI traffic because as we already discussed, the LSI address space is unassigned by IANA. However, it could cause some problems related with misbehaving NATs.

6.9.2 Userspace IPSec

The userspace IPSec extension was implemented almost simultaneously as the LSI extension. The userspace IPSec design relies on the same base as LSIs. Both implementations must be compatible. Userspace IPSec relies on getting HIT-based outbound packets as input and non-HIT inbound packets as output. Figure 6.4 shows the processing of the outbound packet. As we can see, both extensions modify and reinject the packet again. It must be noticed that, while the LSI extension only modifies the IP header, the IPSec extension is a bit more tricky because it encrypts the data and adds the ESP header using the BEET mode.

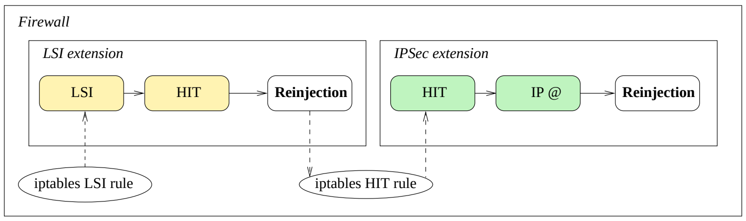

Figure 6.4: Firewall outbound packets schema including LSI and IPSec extensions

Next, we show in Figure 6.5 the input processing. As we can see, now **`hipfw`** applies the userspace IPSec extension in the first place in order to decrypt the packet and to add the HITs into the IPv6 header. Furthermore, we must not forget that **`hipfw`** translates HITs to LSIs only if the destination application does not support IPv6.

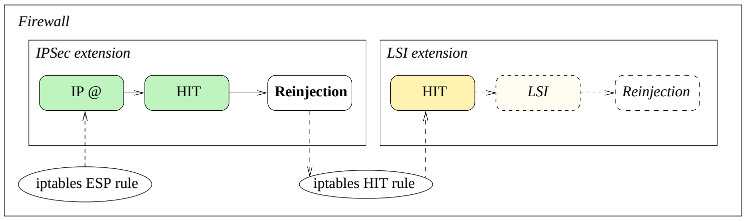

Figure 6.5: Firewall inbound packet schema for LSI and IPSec extensions

6.10 Opportunistic Mode

This section discusses the advantages and disadvantages between system-based and library-based opportunistic modes. Then, we compare the TESLA approach with the opportunistic system library.

One of the main disadvantages of the opportunistic library is the lack of system call implementation. As a consequence, HIPL supports applications in this scenario depending on the system calls that the running application uses. Below we enumerate the supported system calls in the library-based opportunistic mode:

1. socket
2. bind

3. connect
4. send, sendto and sendmsg
5. recv, recvfrom and recvmsg
6. accept
7. write and writev
8. read and readv
9. close
10. listen
11. poll

Although the most used system calls by applications are wrapped, there are still other important functions missing, e.g. setsockopt, getsockname, getpeername, clone, dup, dup2, fclose and select from the current implementation. According to this approach, the user must trace the system calls [56] made by the application in order to find the not supported ones. This objective becomes more complicated when the source code is not available. On the other hand, the system-based opportunistic mode solves this problem because there is no interception library. In this case, it is the `hipfw`, which captures the packets to trigger the opportunistic mechanism. The library fully supports the TCP protocol, but the UDP [48] and ICMP protocols are not yet supported. As we can see, the system-based mode moves the binding from system calls present in the library below the TCP layer.

The second problem that the user library model introduces is a conflict between other LD_PRELOADs. The point to solve is to decide which library the application must process first in case we have another library which is already modifying the same system call as the opportunistic library. Thus, we must take into account how to manage chaining of the libraries in the LD_PRELOAD approach.

With the user library from the TESLA we could use the approach which we introduced in chapter 2. TESLA and the library-based mode share the interposition library schema. But TESLA provides an upper layer of abstraction because the service operates in network flows rather than sockets. However, TESLA is also based on the LD_PRELOAD approach, adding the disadvantages that we described before.

Chapter 7

Conclusions

In this thesis, we focused on supporting legacy IPv4 applications in legacy systems. Despite IPv6 being designed to replace IPv4 in the long term, the reality is that IPv4 will live together with IPv6 during quite a long period. Another important consideration is that there will always be IPv4-only legacy applications whose source code is not available or written in an archaic programming language. Modifying existing applications or rewriting new ones is expensive. By using LSIs provided by HIP, even legacy applications can use IPv6 without modifying them, and also benefit from other features provided by HIP.

We designed and implemented LSI support to enable interoperability between IPv4 and IPv6 applications in HIP for Linux (HIPL), the implementation was selected as the reference HIP implementation.

Apart from the design and implementation of LSIs, we conducted tests on the performance with different HIP identifier scenarios and the two different approaches to the opportunistic mode library. Different tests have been carried out using the following protocols: TCP and ICMP. The LSI and system-based opportunistic mode are currently less efficient in our code than the userspace library-based approach. The implementation does not yet have packet queues and threads which explains the difference in performance. In addition, the LSI module realizes input/output queries per each packet received, in order to decide which IP version the destination application supports, increasing the processing time of the packet. We also experienced a 3-second delay per each measurement during a TCP connection using LSIs. Furthermore, the library-based approach faces some practical problems.

We found some problems with referrals and callbacks. The problem is related to the definition of LSIs, which are limited by its local scope. We studied these scenarios deeply with FTP. We showed that the communication with HITs at both sides works as well as when the FTP client application is IPv6 and the FTP server uses IPv4 in passive mode. During this study, we tried to transfer high quantities of data, realising that the MTU value used by default in HIP had to be changed.

In addition, we reviewed the compatibility with userspace IPSec and the normal firewall access control. As a result, we provided some guidelines for future work.

We proposed to move the opportunistic library from the user to the system level reusing the LSI design. The system level approach solves some current bugs in the opportunistic library, removes library dependency problems and increases the number of applications that HIP supports.

In addition, we compared our approaches with the TESLA approach. TESLA has in common with the opportunistic user library that both of them use the interposition library mechanism. TESLA provides a high-level abstraction to session services and it allows to compose a chain of services. However, TESLA may have the same inherent limitations as the user-based opportunistic library.

To recap, the LSI-based and system-based opportunistic mode approach seem promising ways to facilitate HIP deployment and ease the transition towards IPv6-based networks.

Chapter 8

Future Work

The design and implementation efforts have brought up some future research and development ideas that we describe in this chapter. Initially we discuss future work related with LSIs and afterwards that related with opportunistic mode. Finally we discuss the future guidelines for integrating, in a compatible way, the different extensions in the `hipfw` module.

8.1 LSI Future Work

This section focuses on the related issues with this project that need future work.

8.1.1 Assign an Address Space

Our proposal defines a fixed LSI prefix, but there exist active discussions on this topic because IANA has not assigned the range 1.0.0.0/8 for LSIs. In the future, there may be a need to reserve a name-space for LSIs. This option is not very feasible at the moment because HIP is not deployed widely yet. On the other hand, we can think that the approach where the mapping between LSIs and HITs is done on higher levels of the TCP/IP stack is better. However, this option implies kernel modifications which require more effort and have to be accepted to the linux kernel. Therefore, we decided to implement the other approach that is more suitable for legacy systems.

8.1.2 Support DNS Resolution

DNS Proxy support for LSI was not implemented yet during the writing of this thesis. When the application makes an AAAA request, the DNS Proxy module returns a HIT as an AAAA response if there is a HIP RR. If the application makes an A request and there is a HIP RR, the DNS Proxy module returns an LSI as an A response. The allocated LSI can be received using the output of "*hipconf get ha all|HIT*".

8.1.3 Withdraw Packets Loss

We must provide a mechanism to queue the packets until the hosts establish the Base Exchange. The current LSI implementation drops the packets until the | establishes the Base Exchange, creating an RTO. This approach is detrimental for the LSI performance. One approach can be to block the initiator application after the first query until a positive answer is obtained from the Base Exchange. The loss of packets is not a problem for the TCP protocol because it will retransmit them again, but the same does not apply to UDP or ICMP.

The reinjection mechanism used with LSIs needs to be improved because it adds extra time during the processing of the packet. Namely, userspace IPSec, system-based opportunistic mode and LSI modules immediately reinject the packet after processing it, whereas it would be more efficient for all of them to modify the packet and only then to reinject it. However, this did not work because of interfamily transformations.

8.1.4 Solve the Referral Problem

Applications using referrals or callbacks with LSIs have to be supported in some way. For example, `hipfw` should implement referral conversion for FTP control packets, similarly to what NAT devices and Application Layer Gateways (ALGs) do today. As an alternative solution, the system-based opportunistic mode could be used to handle referral problems. This approach does not break the FTP RFCs and avoids modifications to FTP implementations.

In addition, there are many applications that actually have addresses to application layer headers, although they do not necessarily use them. For example, IM protocols, IMAP, POP, SIP or HTTP. If these protocols have IPv6-based applications, the user or administrator can use HITs.

Unfortunately, the reality is that many IPv6-based applications are still being run without IPv6 support. In this case, we must realize whether the application is just logging the IP addresses but not necessarily using them for anything. In this case, the LSI module works although the application logs the wrong LSI values. The referral problem brought up that more tests must be done with a wide variety of IPv4 applications to find out possible problematic scenarios.

8.1.5 Optimize the Implementation

The HIP implementation of Ericsson handles LSIs closer to the application, at the kernel socket handler which may be a better option because it solves current problems which need future work. Ericsson's implementation solves referral and misbehaving NAT problems because it translates the LSI to the HIT before arriving to the routing tables.

In general, we must improve the LSI performance optimizing the implementation. Moreover, we must find a solution in order to decrease the 3s consumed by the system call `connect()` in the TCP protocol, because the other identifiers have a pretty low delay compared to LSIs.

8.1.6 Improve /proc access

Currently `hipfw` implements a mechanism to decide if it must modify an incoming packet with HITs by a packet containing LSIs or opportunistic IP addresses. `Hipfw` manages this process checking if the destination port is in the file "/proc/net/tcp6".

The main disadvantage is that the daemon does this verification per each packet, running a synchronous input/output operation where the application blocks until the system call is complete. The input/output system calls are the most time consuming ones. Thus, this behaviour creates a bottleneck that we must solve in order to improve the run-time performance.

We would recommend to improve the /proc access by broadcasting the incoming packets. The alternative consists on broadcasting each incoming packet to HIT, LSI and opportunistic IP at the same time. TCP drops packets with duplicate, i.e. out of order sequence. Assume an application able to listen both address families. Then, as TCP uses a sequence number to identify the order of the bytes sent from the client, it processes a first packet and increments the expected sequence number. As the other two packets have a wrong sequence number, TCP discards them without generating any response.

However, the results of the experimentation proof that there are some problems. Let's assume an application listening on port 1111. The hosts receives an incoming packet with LSIs. Then, `hipfw` broadcasts an LSI and HIT-based packets. The LSI packet is delivered properly to the application, but as no application receives the HIT-based packet, the TCP stack generates an RST packet and sends it to the initiator. In such situation, the initiator aborts the TCP communications.

On the other hand, UDP is an unreliable transport protocol, thus duplicate packets could exist unless the application implements duplicate detection.

In conclusion, currently HIPL checks if there is an IPv6 application but broadcast it otherwise (to LSI and system opportunistic-based applications). If we have an IPv6-based system opportunistic mode in the future, we may have to change this behaviour.

8.2 Integrate Different Extensions

The component `hipfw` manages three extensions: opportunistic TCP, userspace IPSec and LSI. The last two extensions follow the reinjection schema explained in this thesis. This procedure adds several disadvantages that we must take into account in order to improve the integration of the extensions. First of all, the number of times iptables process a packet is the number of active extensions plus one. Consequently, `hipfw` enqueues a packet in the output queue of iptables various times, increasing the loading of the queue. Secondly, reinjected packets arrive at the output queue, which makes it hard to distinguish amongst inbound and outbound packet modifications.

To avoid these problems, we could take a new approach. This new proposal changes the reinjection mechanism by creating a chain of handlers that implements all the necessary packet modifications before reinjecting the packet. We must be aware to keep the correct order of extension processing for handling input and output packets to ensure compatibility when more than one extension is in use.

As userspace IPSec relies on receiving HIT-based packets, it must be the last extension processing outbound packets. The LSI extension relies on receiving LSI identifiers and therefore, it must be the first extension `hipfw` processes in order to create a HIT-based packet. On the other hand, userspace IPSec extension must process inbound packets first because it relies on getting non-HIP packets as input and then by the LSI extension because it expects to receive HIT-based packets in order to decide on the translation to an LSI-based packet. The scenario described integrating both extensions is shown in Figure 8.1.

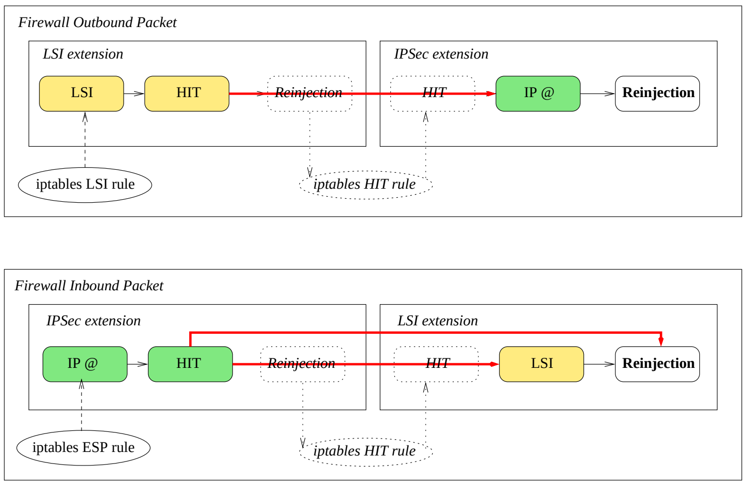

Figure 8.1: Firewall inbound and outbound packets schema including the integration of LSI and IPSec extensions, saving iptables queuing and unnecessary reinjections

Bibliography

[1] A. Pathak and A. Gurtov. IPv4 Support for HIP, May 2006.

[2] L. Aarhus and J. Riisnaes. Emerging Network Protocols From IPv6 and RSVP to ATM, April 1998.

[3] B. Kim, Y. Kim, M. Oh and J. Choi. Microscopic Behaviors of TCP Loss Recovery using Lost Retransmission Detection. In *Consumer Communications and Networking Conference, 2005. CCNC. 2005 Second IEEE*, pages 296–301, January 2005.

[4] C. Benvenuti. *Understanding Linux Networks Internals.* O'REILLY, 1st edition, 2005.

[5] B. Bishaj. Efficient leap of faith security with host identity protocol, June 2008.

[6] D. E. Comer. *Internetworking with TCP/IP Principles, protocols and architecture.* Pearson-Prentice Hall, 5th edition, 2006.

[7] E. Nordmark. *Multi6 Application Referral Issues.* IETF, Jan 2005. [Internet Draft].

[8] HIT collisions statistics. http://www.ietf.org/mail-archive/web/hipsec/current/msg00727.html.

[9] G. Nakhimovsky. Debugging and Performance Tuning with Library Interposers (Originally published in Unix Insider under the title "Building library interposers for fun and profit", July 2001), July 2001.

[10] A. Gurtov. *Host Identity Protocol (HIP). Towards the Secure Mobile Internet.* Willey, 1st edition, 2008.

[11] T. Hain. *RFC 2993: Architectural Implications of NAT*, November 2000.

[12] Thomas R. Henderson. Host mobility for IP networks: A comparison. *IEEE Network Magazine*, 17(6):18–26, November 2003.

[13] InfraHIP Official HomePage. `http://infrahip/`.

[14] InfraHIP Official HomePage Manual. `http://infrahip/manual`.

[15] N. Horman. Understanding and programming with netlink sockets. Technical report, December 2004.

[16] IPv4 Global Unicast Address Assignments. `http://www.iana.org/assignments/ipv4-address-space/`.

[17] Internet Socket. http://en.wikipedia.org/wiki/Internet socket.

[18] Iperf - The TCP/UDP Bandwidth Measurement Tool. `http://dast.nlanr.net/Projects/Iperf/`.

[19] J. Calcote. *Autotools: a practitioner's guide to Autoconf, Automake and Libtool*, 2008. `http://www.freesoftwaremagazine.com/books/`.

[20] J. Dankers, T. Garefalakis, R. Schaffelhofer and T. Wright. Public Key Infrastructure in Mobile Systems. *Electronics Communication Engineering Journal*, 14(5):180–190, October 2002.

[21] J. Salz, A. C. Snoeren and H. Balakrishnan. TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, volume 4. USENIX Association, March 2003.

[22] J. Ylitalo and P. Nikander. A new Name Space for End-Points: Implementing Secure Mobility and Multi-homing across the two versions of IP. In *Proceedings of the 5th European Wireless Conference, Mobile and Wireless Systems beyond 3G*, pages 435–441, February 2004.

[23] P. Francis K. Egevang. *RFC 1631: The IP Network Address Translator (NAT)*, May 1994.

[24] K. Kaichuan. Kernel Korner - Why and How to Use Netlink Socket. *Linux Journal*, January 2005.

[25] S. Kent. *RFC 4303: IP Encapsulating Security Payload (ESP)*, December 2005.

[26] S. Kent and R. Atkinson. *RFC 2401: Security Architecture for the Internet Protocol*, November 1998.

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[28] O. Kirch and T. Dawson. *Linux Network Administrator's Guide: Network Administrator's Guide : [a Unix-compatible Operating System]*. O'REILLY, 1st edition, 2000.

[29] J. Klensin. *RFC 3467: Role of the Domain Name System (DNS)*, February 2003.

[30] M. Krasnyansky. Universal TUN/TAP driver. `http://vtun.sourceforge.net/tun/`.

[31] J. Laganier and L. Eggert. *RFC 5204: Host Identity Protocol (HIP) Rendezvous Extension*, April 2008.

[32] *libipq*, 2001. Linux Programmer's Manual.

[33] The quick intro to libipq. http://www.imchris.org/projects/libipq.html.

[34] V. K. Lam M. E. Fiuczynski and B. N. Bershad. The design and implementation of an ipv6/ipv4 network address and protocol translator. In *Proceedings of the Annual Conference on USENIX*, June 1998.

[35] M. E. Fiuczynski, V. K. Lam and B. N. Bershad. The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator. In *Proceedings of the 1998 USENIX Conference*, 1998.

[36] M. Komu and J. Lindqvist. Leap of Faith Security is Enough for Mobility. In *Proceedings of the 2008 Applied Cryptography and Network Security Conference*, June 2008.

[37] M. Komu and J. Lindqvist. Leap-of-Faith Security is Enough for Mobility, January 2009. To appear in the 6th Annual IEEE Consumer Communications Networking Conference IEEE CCNC 2009.

[38] M. Komu and T.Henderson. *Basic Socket Interface Extensions for Host Identity Protocol (HIP)*. IETF, July 2008. [Internet Draft].

[39] M. S. Blumenthal and D. D. Clark. Rethinking the design of the Internet: The end to end arguments vs the brave new world. 1, August 2001.

[40] N. Seddigh and M. Devetsikiotis. Studies of TCP's Retransmission Timeout Mechanism. 6:1834–1840, June 2001.

[41] Netcat. http://netcat.sourceforge.net/.

[42] P. Nikander and J.Melen. *A Bound End-to-End Tunnel (BEET) mode for ESP*. IETF, August 2006. [Internet Draft].

[43] P. Nikander and J. Laganier. *RFC 5205: Host Identity Protocol (HIP) Domain Name System (DNS) Extensions*, April 2008.

[44] R. Moskowitz P. Jokela, Ed. and P. Nikander. *RFC 5202: Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)*, April 2008.

[45] J. Laganier P. Nikander and F. Dupont. *RFC 4843: An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID)*, April 2007.

[46] P. Nikander, J. Ylitalo and J. Wall. Integrating Security, Mobility, and Multi-homing in a HIP Way. In *Proceedings of Network and Distributed Systems Security Symposium. NDSS 2003*, volume 3, pages 2120–2125, February 2003.

[47] *ping and ping6*, 2007. System Manager's Manual: iputils.

[48] J. Postel. *RFC 768: User Datagram Protocol.* Internet Engineering Task Force, August 1980.

[49] J. Postel. *RFC 791: Internet Protocol (IP)*, September 1981.

[50] J. Postel. *RFC 793: Transport Control Protocol.* IETF, September 1981.

[51] J. Postel and J. Reynolds. *RFC 959: File Transfer Protocol (FTP)*, October 1985.

[52] P. Jokela Ed. R. Moskowitz, P. Nikander and T. Henderson. *RFC 5201: Host Identity Protocol*, April 2008.

[53] R. Moskowitz, P. Jokela, P. Nikander and T. Henderson. *Host Identity Protocol.* IETF, June 2004. [Internet Draft].

[54] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, 1st edition, 1994.

[55] W. Richard Stevens. *UNIX Network Programming, Volume1: The sockets networking API.* Addison -Wesley, 3rd edition, 2004.

[56] *strace*, 2003. Linux Programmer's Manual.

[57] T. Aura, A. Nagarajan and Andrei Gurtov. Analysis of the HIP Base Exchange Protocol. In *10th Australasian Conference on Information Security and Privacy*, July 2005.

[58] T. R. Henderson, J. M. Ahrenholz and J. H. Kim. Experience with the Host Identity Protocol for Secure Host Mobility and Multihoming. In *Wireless Communications and Networking. WCNC 2003. 2003 IEEE*, volume 3, pages 2120–2125, March 2003.

[59] TCP delay. http://www.linuxquestions.org/questions/linux-networking-3/3000ms-delay-on-tcp-connections-613670/.

[60] The Transparent Extensible Session-Layer Architecture for End-to-End Network Services. `http://www.sds.lcs.mit.edu/projects/tesla/`.

[61] The PKI page. http://www.pki-page.org.

[62] T.Henderson, P. Nikander and M. Komu. *Using the Host Identity Protocol with Legacy Appplications.* IETF, July 2008. [Internet Draft].

[63] Joel M. Winett. *RFC 0147: The Definition of a Socket.* Internet Engineering Task Force, May 1971. `http://www.ietf.org/rfc/rfc0147.txt`.