Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Xin Gu

Host Identity Protocol Version 2.5

Master's Thesis
Espoo, June 28, 2012

Supervisors: Professor Antti Ylä-Jääski, Aalto University
Professor Peter Sjödin, Royal Institution of Technology
Instructor: M.Sc Miika Komu

Aalto University
School of Science

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Xin Gu		
Title: Host Identity Protocol Version 2.5			
Date:	June 28, 2012	**Pages:**	74
Professorship:	Data Communication Software	**Code:**	T-110
Supervisors:	Professor Antti Ylä-Jääski Professor Peter Sjödin		
Instructor:	M.Sc Miika Komu		

The Host Identity protocol (HIPv1) is a flexible protocol that solves a number of security, mobility and other problems in the current Internet. In this thesis, we present our work on HIP version two (HIPv2) and HIP version 2.5 (HIPv2.5) in the HIP for Linux (HIPL) project.

For HIPv2, we analyze the transition from v1 to v2 in the specifications and present design of a dual-version HIPL to facilitate this transition. We also show an implementation and validation of the new cryptographic agility framework as required by HIPv2.

As we expect HIPv2 to have the same deployment obstacles as HIPv1, we take a step forward and propose HIPv2.5. Contrary to HIPv2, HIPv2.5 does not change current HIP message definitions and handling processes, but provides a library-based solution to facilitate the adoption of HIP. We design and implement a prototype of a library-based HIPL, which shifts HIP from below the transport layer to above it while retaining the security and mobility features of HIP. The library provides new API closely modeled after the sockets API for easier adoption for network application developers, which has earlier been an issue for HIP adoption. We demonstrate the function of the library using an example test application.

Keywords:	HIP, mobility, application, API, session, authentication, library
Language:	English

Acknowledgements

For me, the thesis work on the first half of 2012 is a wonderful experience. I solve many interesting problems and fulfill one of my dream which is contributing to an open source software. I would not achieve those results without the help of many kind hearts.

First I would like to thank my two supervisors professor Antti Ylä-Jääski and professor Peter Sjödin and my instructor Miika Komu. Thanks for all your guide and help during these six months. Especially, I would like to express my gratitude to Miika, who fills my email box with documents and papers, patiently listens to each detail technical problem I have met and gives tons of comments on the thesis. I enjoy those discussions and brain storming we had and gain a lot of knowledge from them.

I also want to thank members in the HIPL project team, René Hummen, Diego Biurrun, Stefan Götz, Christof Mroz, David Martin and Tobias Heer. Thanks for your valuable review comments and tolerance when I made mistakes. I hope we still have chance to work together and improve the HIPL project in the future.

Of course, without the support of my friends, I would not be so optimistic on everything during this time period. I want to give my thanks to Hao Zhuang, Daoyuan Li, Guohua Liu, Xiang Gan, Antony Meyn and Vu Ba Tien Dung, who help me both on technical parts and daily life. Finally, I want to thank my father and mother for supporting me to join the Nordsecmob program, which gives me a colorful two-year memory in two beautiful Nordic countries.

Otaniemi, June 18, 2012

Xin Gu

Abbreviations and Acronyms

API	Application Programming Interface
ARM	Advanced RISC Machine
BEX	Base Exchange
CID	Connection Identifier
DNS	Domain Name System
DNSSEC	DNS Security Extensions
DoS	Denial of Service
DSA	Digital Signature Algorithm
DTLS	Datagram Transport Layer Security
ESP	Encapsulating Security Payload
FTP	File Transport Protocol
FQDN	Fully Qualified Domain Name
KDF	Key Derivation Function
NAT	Network Address Translation
HA	Host Association
HICCUPS	HIP Immediate Carriage and Conveyance of Upper-layer Protocol Signaling
HIP	Host Identity Protocol
HIPL	HIP for Linux
HIPv1	HIP version 1
HIPv2	HIP version 2
HIPv2.5	HIP version 2.5
HI	Host Identifier
HIT	Host Identity Tag
HSOCK	HIPL Socket
IANA	Internet Assigned Numbers Authority
ICE	Interactive Connectivity Establishment
IP	Internet Protocol
IPCS	Industrial Process Control System
IPsec	Internet Protocol Security

ISP	Internet Service Provider
IoC	Inverse of Control
MPTCP	Multi-Path TCP
MTU	Maximum Transmission Unit
NSENT	Number of bytes Sent
NRECV	Number of bytes Received
OGA	ORCHID Generation Algorithm
ORCHID	Overlay Routable Cryptographic Hash Identifiers
P2P	Peer-to-Peer
POSIX	Portable Operating System Interface
RFC	Request For Comments
ROCKS	Reliable Sockets
RTT	Round Trip Time
RVS	Rendezvous Server
SCADA	Supervisory Control And Data Acquisition
SEQ	Sequence
SRTP	Secure Real-time Transport Protocol
SSH	Secure Shell
STUN	Session Traversal Utilities for NAT
SYNC	Synchronization
TCP	Transmission Control Protocol
TESLA	Transparent Extensible Session-Layer Framework
TLS	Transport Layer Security
TLV	Type, Length and Value format
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
RTCWEB	Real-Time Communication in WEB-browsers

Contents

Abbreviations and Acronyms **4**

1 Introduction **10**
1.1 Problem Statement . . . 11
1.2 Structure of the Thesis . . . 12

2 Background **13**
2.1 Host Identity Protocol (HIP) . . . 13
2.1.1 Host Identifier (HI) . . . 13
2.1.2 Host Identity Tag (HIT) . . . 14
2.1.3 HIP Message Format . . . 15
2.1.4 HIP Base Exchange (BEX) . . . 15
2.1.5 HIP Native API . . . 17
2.2 HIP Version 2 (HIPv2) . . . 19
2.2.1 HIT Generation Mechanism . . . 19
2.3 Engineering Efforts on HIP . . . 20
2.3.1 Daemon-based Approach . . . 21
2.3.2 HIPL Modularization Framework . . . 21
2.4 HIP NAT Traversal . . . 23

3 Design **25**
3.1 HIP Version 2 . . . 25
3.1.1 Dual-version HIP . . . 25
3.1.1.1 Analysis on HIPv1 to HIPv2 Transition . . . 25
3.1.1.2 Dual-version Support . . . 27
3.1.2 Agile Cryptographic Framework . . . 28
3.2 HIP Version 2.5 . . . 31
3.2.1 Requirement Specification . . . 31
3.2.2 Legacy Compatibility Analysis . . . 32
3.2.3 Demultiplexing of the Control and Data Plane . . . 34
3.2.4 API Design Alternatives . . . 35

3.2.5 Mobility . . . 36
3.2.5.1 Buffer Management and Data Consistence . . 37
3.2.5.2 Connectivity Status Detection . . . 39
3.2.5.3 Handover . . . 40
3.2.6 Comparison between Sockets and HIPL Library API . 43

4 Implementation **44**
4.1 Overview . . . 44
4.2 Select Support . . . 45
4.2.1 Delayed BEX . . . 47
4.2.2 On Blocking Operations . . . 47
4.3 Handover . . . 50

5 Experimentation **52**
5.1 System-test Application . . . 52
5.2 Hipnetcat . . . 53

6 Related Work **54**
6.1 Mobility . . . 54
6.1.1 Mobile IP . . . 54
6.1.2 Reliable Sockets (ROCKS) . . . 54
6.1.3 Multi-Path TCP . . . 55
6.1.4 DTLS Mobility . . . 55
6.1.5 SSH and TLS Resilient Connections . . . 56
6.1.6 TESLA . . . 56
6.1.7 Other Transport Layer Mobility Solutions . . . 56
6.2 Name-based Sockets . . . 57

7 Future Work **58**
7.1 HIPv2 . . . 58
7.2 HIPv2.5 . . . 58
7.2.1 Protection of the Data Plane . . . 59
7.2.2 Integrate HIPL Library to Existing Applications . . . 59
7.2.3 Adaptive Handover Timer . . . 59
7.2.4 Inverse of Control (IoC) . . . 59
7.2.5 Concurrent execution on the Library API . . . 60
7.2.6 HIP RVS Incompatibility . . . 60
7.2.7 Built-in Network Address Translation (NAT) Support . 60
7.2.8 TCP User Timeout Option . . . 61
7.2.9 Library API Improvement . . . 61
7.2.10 Opportunistic Mode and Fallback Mode . . . 61

7.2.11 IPv6 Support . . . 62

8 Conclusion **63**

A HIPL Library API **70**
A.1 Initialize the Library . . . 70
A.2 Creating a Socket . . . 70
A.3 Binding a Socket . . . 70
A.4 Setting Up a Listening Socket . . . 71
A.5 Accepting a New Connection . . . 71
A.6 Connecting to a Remote Server . . . 71
A.7 Sending Data . . . 71
A.8 Receiving Data . . . 71
A.9 Closing a Socket . . . 72

B Hipnetcat **73**
B.1 Usage . . . 73
B.2 Example Runs . . . 74

List of Figures

2.1 The Current Internet Binding Model (left) and the HIP Binding Model [31] . . . 14
2.2 HIP Message Format . . . 16
2.3 HIP Version 1 Base Exchange . . . 17
2.4 The Layering Model of HIP Native API [23] . . . 18
2.5 The HIT centric namespace model [23] . . . 18
2.6 HIT Generation in HIP version 1 (HIPv1) and HIPv2 . . . 20
2.7 HIP-Based Devices from TOFINO . . . 21
2.8 The Libmod Message Handler Registration in HIPL . . . 23

3.1 HIP Version 2 Base Exchange . . . 29
3.2 HIP Version 2 Diffie-Hellman Negotiation . . . 30
3.3 System Architecture of HIPL Library . . . 32
3.4 Protocol Stack Change from HIPv1/HIPv2 to HIPv2.5 . . . 33
3.5 HIPv2.5 Packet Encapsulation . . . 35
3.6 HIPL Library BEX Process . . . 37
3.7 Application Data Assembling through the Library Input Buffer 38
3.8 HIPv2 Handover . . . 42

4.1 HIPL Library Function Hierarchy . . . 45
4.2 A Client-Side Handover . . . 51

Chapter 1

Introduction

Within 20 years, the Internet has evolved from a small network of research institutions to an universal communication carrier, which connects every little corner of our world. Although the expansion of the Internet was constrained by the dot-com bubble 10 years ago, the Internet surge is still formidable because of the rising of the mobile Internet. In the mobile era, Internet Protocol (IP) faces a great challenge as an IP address serves a dual purpose: it defines both "who" and "where" the host is. Transport layer and application layer utilize IP addresses to identify hosts, while IP layer uses them as locators. For a static host with a single IP, this coupled binding is mostly valid. However, the portable nature of mobile clients leads to frequently changing IP address, which can disrupt upper layer protocols. Unintended changes on IP addresses break media streams on top of Transmission Control Protocol (TCP), and can have adverse effects in contacting of hosts and IP address-base access-control lists. Mobile IP makes an effort to fill this gap by introducing a surrogate IP address to be used as host identifier, but requires extra infrastructure.

In addition to mobility concerns, there are three main problems for the current Internet. First, security protocols in the Internet are challenged by mobile clients. A number of solutions have emerged to secure communications for the Internet, such as Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), but lack NAT traversal support. Second, traversal through NAT middleboxes has become prevalent. Due to the shortage of IPv4 addresses, many hosts are located behind NAT devices and employ private IP addresses. Since NAT devices cannot handle incoming connections without manual pre-configurations, hosting servers behind the NAT is inconvenient. P2P applications are complicated by NATs drastically because they have use tricks to penetrate the NAT boxes and the tricks are not always reliable. Third, the Internet is challenged by IPv6 adoption and

interoperability. The slow progress of IPv6 deployment indicates that IPv4 and IPv6 will coexist for a long term. In order to function properly, applications should support both protocols but the migration involves a lot of effort and costs.

The Host Identity Protocol (HIP) offers a more complete solution for the four problems within a single protocol for legacy applications. HIP introduces unique identifiers for each host. Instead of relying on IP addresses, Applications can use HIP-based identifiers for persistent naming. At the same time, the identifiers are cryptographically protected and can be used for secure authentication. HIP includes a key exchange procedure called the Base Exchange (BEX) to minimize the cryptographic overhead, during which the hosts negotiate symmetric keys to protect the data of the applications.

While HIP supports multiple features within a single protocol, the process of adopting a new protocol is never easy because it is not only determined by the technical aspects of the protocol, but also governed by deployment costs and integration aspects. In the past, HIP has required changes into the Linux kernel in Internet Protocol Security (IPsec) processing, while this has been successfully adopted into the Linux networking stack, Application Programming Interfaces (APIs) supporting HIP have not been adopted yet, the APIs require again kernel changes, thus requiring another deployment hurdle. However, the API for HIP-aware applications is already needed now to facilitate HIP adoption and the situation appears as a "chicken-and-egg" problem.

In this thesis, we examine the design and implementation of HIPv2 and HIP version 2.5 (HIPv2.5) in the HIP for Linux (HIPL) project. HIPv2 brings an agile framework to adopt new cryptographic algorithms and stronger ciphers. To ease the protocol upgrade transition, we design a dual-version HIPL. Meanwhile, our proposed HIPv2.5 aims to extend HIP into a standalone user-space library in contrast to normal daemon-based solutions. The library-based implementation can be integrated into an application when it requires mobility and needs to be aware of the secure identifiers of HIP, e.g., for access control purposes.

1.1 Problem Statement

This thesis aims to address 2 two research problems:

1. HIPv2: the new version of the protocol raises some transition concerns. The standardization process for the HIP version 2 also demands feedback from implementators. In the HIPL project, we explore a seamless

transition solution and examine gaps between HIPv2 draft and implementation. We believe that this is not only beneficial for v1-v2 transition but also if there is a need for HIPv3 at some point.

2. HIPv2.5: considering that HIP has not been deployed widely, in an unpublished paper, the authors have interviewed 19 experts to examine the deployment challenge of HIP [48]. They suggested that a HIP implementation as an application-layer middleware provides better deployability then common network-layer solutions. Based on the result, we propose to build HIP as a standard-alone library to extend the coverage of HIP, which features easier integration and experimentation.

1.2 Structure of the Thesis

The thesis consists of eight chapters. The second chapter is a background study, where we review the concept of HIP, its extensions, and the related projects. In the third chapter, we focus on the design of HIP version 2 and 2.5. This includes dual-version support (version 1 and version 2), agile cryptographic framework, library-based HIP approach and mobility solution. Chapter four and five details the implementation and shows experimentation results to validate the design. Chapter six presents name-based sockets and compares different mobility solutions with our library-based approach as related work. Finally chapter seven discusses further work and chapter eight concludes the thesis.

Chapter 2

Background

2.1 Host Identity Protocol (HIP)

The Internet nowadays has two major global namespaces, IP address and Domain Name System (DNS). The IP address was originally intended for routing packets to certain locations, but then was reused for host identification. This coupled role makes IP address based host identification more difficult in mobile Internet, because IP address based mobile hosts cannot be identified persistently as their IP addresses are prone to change. IP address also lacks any security protection and does not offer any protection against IP spoofing.

HIP [40] is a protocol to address these gaps in the Internet architecture. With the help of HIP, the role of an IP address returns to its origins as a routing and location tag while another new identifier is introduced by HIP to identify hosts globally. The new identifier is also cryptographically protected thus difficult to forge.

HIP is composed of both a control plane and a data plane: the control plane is responsible for establishing a security association between end hosts and generating shared key; the data plane uses the shared key from the control plane for keying materials to protect the application traffic.

2.1.1 Host Identifier (HI)

HIP introduces a new Host Identity namespace [30] for the Internet. Each entity in this namespace is identified with a Host Identifier (HI). In HIP, the Host Identity and Host Identifier are two different concepts. The former is an abstract term and the latter is the realization of the abstract concept. The HIP architecture specification defines the Host Identifier as “a public key used as a name for a Host Identity” [30].

Figure 2.1 illustrates the binding between a HI and its corresponding locators. The binding can be also one-to-many for mobility and multihoming. The HI is the public key part of an asymmetric key-pair of an end host. A HI is statistically unique. The HI also facilitates mutual authentication between two hosts. HIP uses it and the corresponding private key to protect the control and data plane between two hosts.

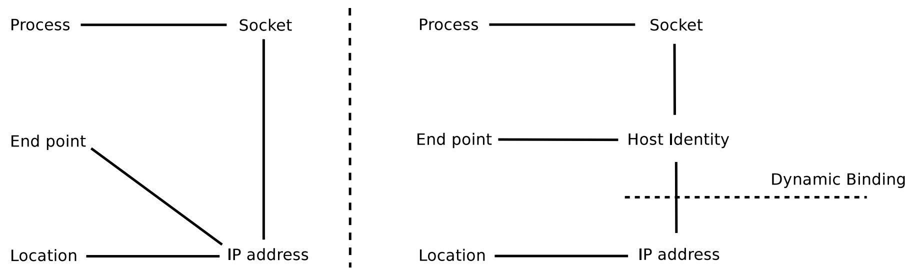

Figure 2.1: The Current Internet Binding Model (left) and the HIP Binding Model [31]

A HI is independent of IP addresses. Combined with its global (statistical) uniqueness, it can be used to identify hosts even in overlapping private address realms. Thus HIP has the potential to revive the end-to-end communication model for the Internet.

Different asymmetric key algorithms have their own presentation formats for a public key and HIP reuses to the DNS Security Extensions (DNSSEC) [38] specification for public-key encoding. The definition of the encoding for RSA keys is specified in RFC-3110 [1] and in RFC-2536 [13] for Digital Signature Algorithm (DSA) key encoding.

2.1.2 Host Identity Tag (HIT)

Embedding a complete HI in each HIP message is inefficient for two reasons: first, the length of the result after encoding it is relatively large, subjecting the control plane for fragmentation issues; second, the length of the result is arbitrary, which makes the format of the protocol more complicated.

For space-efficiency reasons, HIP introduces Host Identity Tag (HIT) for message exchange. A HIT is a fixed length (128-bit, the same as IPv6 address) tag. It is a secure hash result of an encoded HI. Reversing a hash is computationally difficult, and therefore the HIT can be considered as a secure identifier.

2.1.3 HIP Message Format

A HIP message contains a HIP header and its body. The header is 40-byte long and the length of the body is defined in the header. Figure 2.2 presents the structure of a HIP message with the following fields:

- Next Header: the HIP header conforms to the definition of a header in IPv6 that starts with a 1-byte next header field.
- Header Length: this field defines the length of the body for this HIP message.
- Packet Type: this field determines the type of the current HIP message. For instance, the HIP BEX (described in Section 2.1.4) consists of four different type of messages: I1, R1, I2 and R2. In addition to them, HIP also has other types of message such as the UPDATE message and the NOTIFICATION message.
- Version & Reserved: the HIP version number can be inferred from this field. In addition to the version number, several bits are reserved for future extensions.
- Checksum: the Checksum is calculated based on the HIP control message and pseudo IP header as defined in the IPv6 specification [11].
- Controls: the controls field consists of bit flags to handle special situations for different message types.
- Sender HIT and Receiver HIT: the HIT of the sender and the HIT of the receiver.

A HIP message body is composed of several HIP parameters, and each parameter is presented in Type, Length and Value format (TLV). This way, each HIP parameter is self-contained and new parameter can be added easily without changing the parameter processing. This feature makes HIP a flexible protocol to handle optional extensions defined by new parameters.

2.1.4 HIP Base Exchange (BEX)

The process of establishing HIP security association (or HIP association) is called HIP BEX. The BEX consists of four steps and four HIP messages are involved: I1, R1, I2 and R2. The Figure 2.3 shows a BEX between two hosts, which are called the initiator and responder. For demonstration purposes, this figure only includes the most important parameters for the BEX. The steps are as follows:

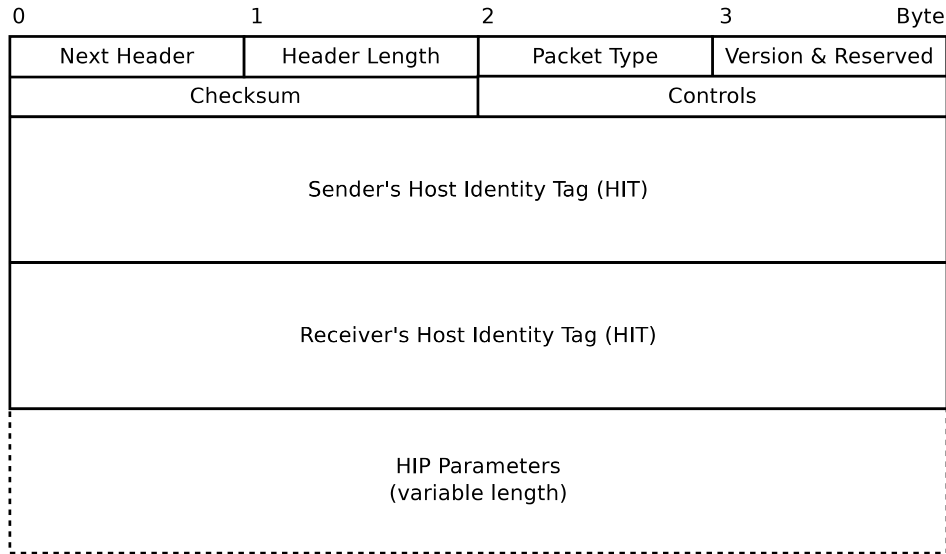

Figure 2.2: HIP Message Format

1. The initiator sends an I1 message, which contains only a HIP header. This message essentially notifies the responder the start of BEX.

2. The responder sends a R1 message to inform the initiator about its HI, a computational puzzle for the initiator to solve (PUZZLE), a Diffie-Hellman public value (DH_VALUE) and a signature of the packet (SIG). The R1 message can be pre-created and sent without any cryptographic processing cost before receiving an I1 message, which allows the responder to remain stateless at this stage and to prevent the Denial of Service (DoS) attacks against HIP control plane.

3. After the initiator receives the R1, several steps are required to process the message: a) the initiator must check the correctness of the signature in the R1. b) the initiator prepares its HI, solves the puzzle (SOLUTION) which requires a certain level of computation effort, generates its Diffie-Hellman public value (DH_VALUE), calculates a message authentication code (MAC) and signature (SIG). c) finally, initiator groups the parameters into an I2 message and sends it to the responder.

4. Upon receiving the I2 message, the responder must also verify the signature, MAC and puzzle solution. At this point, the Diffie-Hellman key exchange is finished and both side generate an identical shared key. The responder then sends a R2 message which contains MAC and

signature (SIG) to notify the initiator on the completion of the BEX.

The two hosts use the shared key generated in the BEX as key material for the protection of the data plane. The base exchange specification in RFC-5201 [40] does not explicitly mandate any specific method for the data plane to support modularity. Implementations can freely choose different algorithms and methods. RFC-5202 [32] illustrates a way to establish Encapsulating Security Payload (ESP) [20] to secure the data plane.

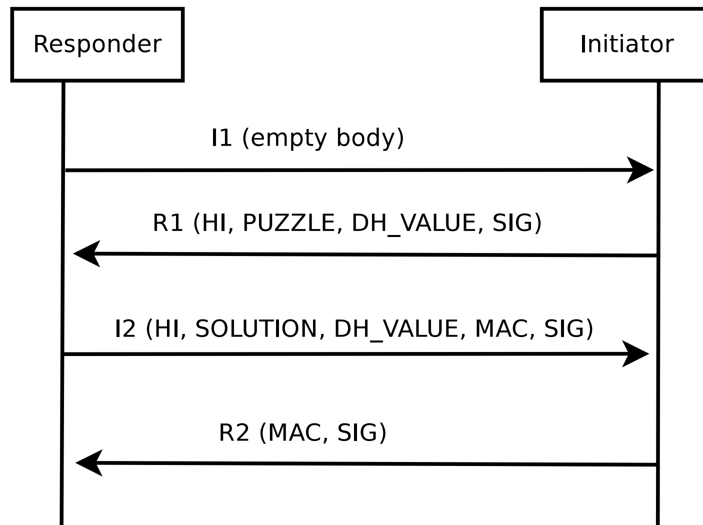

Figure 2.3: HIP Version 1 Base Exchange

2.1.5 HIP Native API

HIP Native API [23, 24] extends Sockets API to support HIP through a new PF_HIP socket family. The API closely follows the design of Sockets API to maximize reusability. Figure 2.4 illustrates the layering architecture of the native API. The HIP layer is located between the transport and network layer. At the socket layer, a new HIP API is provided for applications to access HIP functionality. Each layer uses different namespace identifier, which is presented in figure 2.5. On the top, the User Interface uses the Fully Qualified Domain Name (FQDN) for better human readability. The application layer uses source HIT, destination HIT, source port, destination port and protocol to identify a connection. This model decouples IP address which is originally part of the application layer identifier.

The HIP Native API is mainly implemented in the kernel space and it can be divided into two parts, the HIP module and the HIP socket handler. The

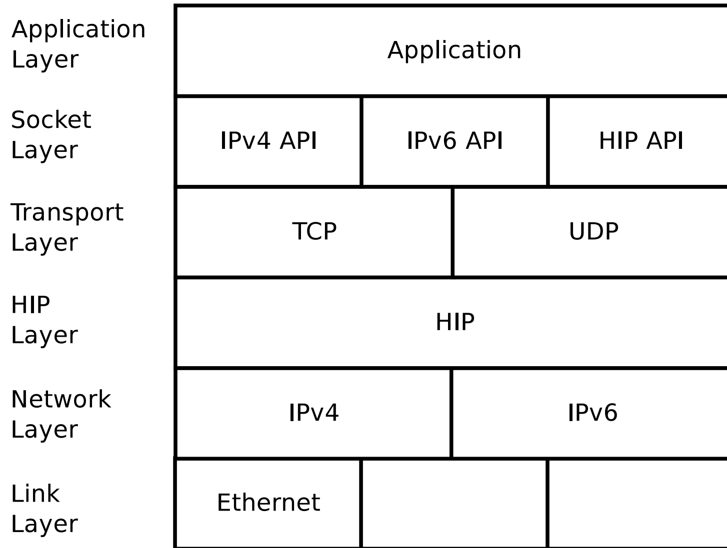

Figure 2.4: The Layering Model of HIP Native API [23]

HIP module is the implementation of the protocol, which handles mechanisms such as BEX and update. The HIP socket handler is registered into the network stack as a handler for the new PF_HIP socket family. This way, the handler becomes a bridge for applications to utilize the functionality provided by the HIP module in the kernel.

HIP Native API also has a fallback option for applications. If the peer host of an application does not support HIP, it can switch back to plain TCP/IP model for compatibility purpose.

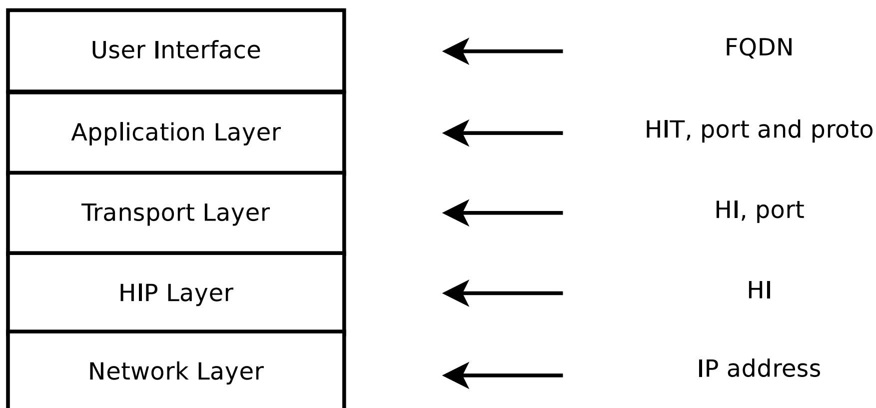

Figure 2.5: The HIT centric namespace model [23]

2.2 HIP Version 2 (HIPv2)

The HIPv1 RFC [40] was published on April 2008. It is based on now obsoleted cryptographic algorithms that addressed by HIPv2 [41] that introduces stronger ciphers. It also provides an agile framework for algorithm negotiation. The HIPv2 RFC was still working in progress during the writing of this thesis.

2.2.1 HIT Generation Mechanism

In the HIPv1, the HIT generation method is based on the generic specification of Overlay Routable Cryptographic Hash Identifiers (ORCHID) [9]. It defines a standardized algorithm to generate hash-based identifiers that are compatible with IPv6 addresses. The proposal also defines an IPv6 prefix (2001:10::/28) obtained from the Internet Assigned Numbers Authority (IANA). HIPv1 conforms to this specification by applying the ORCHID prefix in HITs and truncates the hash digest of a HI to fill in the remaining 100 bits.

In HIPv2, the HIT generation method has been upgraded to meet the requirement of “cryptographic agility”. As a result, ORCHID Generation Algorithm (OGA) is introduced for the generation of a HIT. Figure 2.6 compares the HIT generation mechanisms in HIPv1 and HIPv2:

- A HIT in HIPv1 consists of two parts. The left side consists of the fixed 28-bit prefix as assigned by IANA. For the right 100 bits, HIPv1 first applies SHA-1 [2] to digest a pre-organized structure of a HI, and then truncates the result to fit into 100 bits.

- A HIT in HIPv2 consists of three parts. The first part, that is, the 28-bit prefix is kept unchanged. Then a new 4-bit of OGA field follows, which is marked with gray color in Figure 2.6. The third part is the hash result of a HI, which is reduced to 96 bits.

  HIPv1 only supports SHA-1 as the hash function for HIT generation. It is well-known that a security vulnerability on the collision resistance property of SHA-1 is found by Xiaoyun Wang [49]. Although the vulnerability does not affect the ORCHID specification drastically because the specification relies on the second-preimage resistance property of the hash function, keeping using an algorithm which is extensively examined for security vulnerabilities [6] discredits HIP as a secure protocol. Therefore, HIPv2 decided to add the OGA field for adopting stronger algorithms. Based on this field, HIPv2 hosts can infer hash

algorithm being used rather than iterating through all the possible combinations to find a match. The algorithms can differ but have to be supported by both sides.

The generation of the last part of a HIT in HIPv2 is similar to the generation of the second part in HIPv1, with the only difference that the hash result is truncated to 96 bits since 4 bits is assigned to the OGA field.

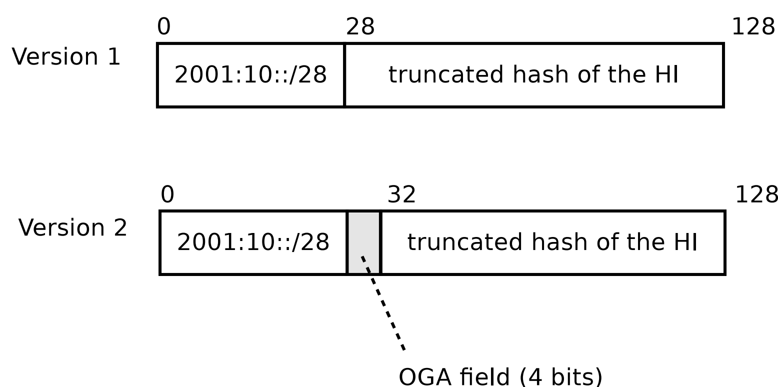

Figure 2.6: HIT Generation in HIPv1 and HIPv2

2.3 Engineering Efforts on HIP

OpenHIP[1] and HIPL[2] [36] are two active projects for the implementation of HIP. OpenHIP is a cross platform solution for Linux, BSD, Mac OS X and Windows while HIPL focuses specifically on the Linux platform. HIP for Internet from Ericsson is not active at the moment.

In the industry, TOFINO[3] offers HIP-based devices to protect Supervisory Control And Data Acquisition (SCADA) and Industrial Process Control System (IPCS). Figure 2.7 demonstrates the product from the company. Boeing also uses HIP as part of their SCADA solution for assembling airplanes [35] and the company plans to utilize HIP to develop secure smart phones[4].

[1] http://www.openhip.org/
[2] https://launchpad.net/hipl
[3] http://www.tofinosecurity.com/
[4] http://www.itnewsafrica.com/2012/04/boeing-to-develop-own-smartphone/

Figure 2.7: HIP-Based Devices from TOFINO

2.3.1 Daemon-based Approach

Both OpenHIP and HIPL utilize a daemon-based approach for HIP implementation. The HIP daemon is a process maintains HIs and state of HIP associations. The daemon is also responsible for translating HITs to locators and mobility management. An advantage for this approach is that the HIP control plane and data plane are clearly separated[5]. Control plane messages are exchanged between daemon processes, while data plane messages flow between user applications, possibly secured by a separate security module such as IPsec.

However, the daemon approach has its own disadvantages: first, the security improvements are transparent to applications. The secure authentication as offered by HIP BEX and data encryption/decryption by IPsec are totally invisible for applications. Second, it is difficult to achieve HIP-related configuration at application granularity. All the applications have to share the same configuration for one host. Third, the daemon process normally requires administrator privilege, which is not always allowed in all environments.

2.3.2 HIPL Modularization Framework

HIP is an extensible protocol. The HIPL implementation also prioritizes flexibility and extensibility, which has led to a modularization framework for protocol implementation called Libmod [19]. The Libmod framework aids the implementation of HIPL in two key areas:

[5]with the exception of the HIP Immediate Carriage and Conveyance of Upper-layer Protocol Signaling (HICCUPS) extension

1. By using the Libmod, HIPL separates protocol core and extensions: Libmod divides a monolithic protocol implementation into modules by their functionality and coordinates their communication with the protocol "core". Modules on top of the Libmod only rely on APIs of this library, which decouples source code dependencies internally.

   The characteristic of this library meets the implementation requirement of a flexible protocol design like HIP. Instead of mixing many HIP extensions together within the protocol core, they can be implemented in different plug-in modules and introduced to the project through a configuration file. In the HIPL project, extensions such as HIP certificates [16] are built as Libmod modules.

2. The Libmod maintains a state machine and a function registry service for a protocol implementation. There are three stages in the Libmod for function registration: initialization, message handling and protocol maintenance.

   In the initialization stage, modules can initialize the necessary states, data structures, check the availability of required features, and register their handlers and so called maintenance functions.

   In the message handling stage, the library coordinates the modules and the protocol core to process a given message. This coordination is based on the handler function registration. The registration service allows functions to be bound to certain predefined criterion with a priority number. When handling a message, the Libmod finds the criterion matching the current message and executes handler function chain registered by this criterion one by one ordered by priority. The criterion consists of the combination of HIP message type and current HIP association state.

   Figure 2.8 illustrates the function registration mechanism. In the figure, square boxes with solid lines stand for the criteria and the round boxes are the registered functions. The figure can be interpreted in two directions: horizontally, with the gray bar including all the criteria HIPL uses for message handling, such as handling of the I1 message when no HIP association exists (the first criterion) and handling of the R1 message when current state is I1-SENT. Vertically, each chain shows registered functions for a single criterion. In the chain 1, four handler functions are registered and three in the Chain 2.

   The maintenance stage also has a similar handler function registration mechanism. The difference between this stage and the previous one is

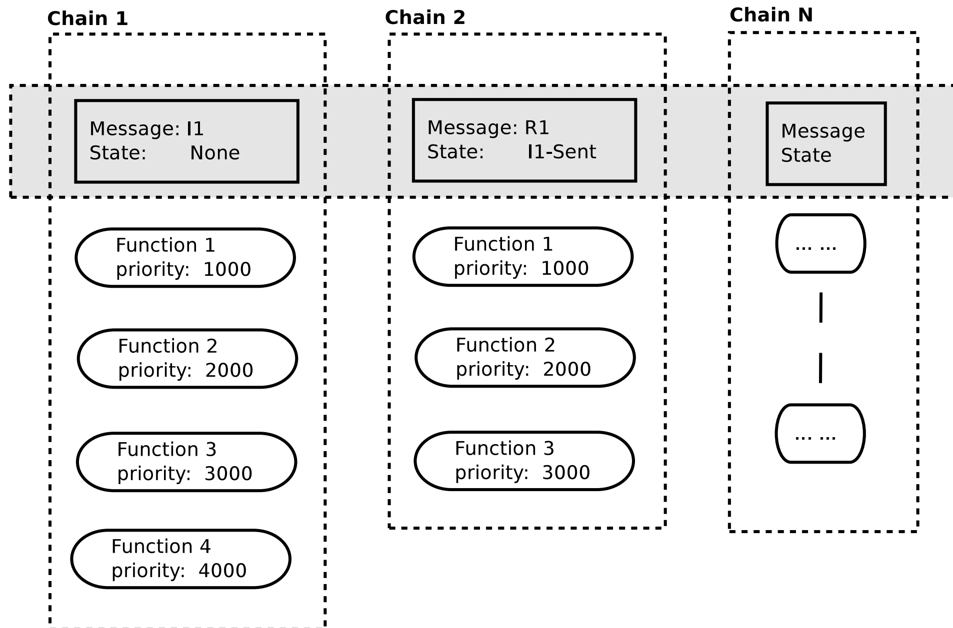

Figure 2.8: The Libmod Message Handler Registration in HIPL

that the maintenance stage is not driven by incoming messages, but by time. Maintenance functions are triggered periodically and are suitable for implementing features requiring to be performed repetitively over time. For instance, the HIPL project uses the maintenance stage to send "heartbeat" messages to associated hosts to detect when they are not reachable anymore.

2.4 HIP NAT Traversal

The NAT traversal in HIPv1 is achieved by an HIP extension [28] which utilizes mechanisms from the Interactive Connectivity Establishment (ICE) [17, 39, 43] protocol for path discovery between two hosts. ICE is an UDP-based NAT solution which uses an offer and answer model and a connectivity checks mechanism to find the best path.

The HIP NAT traversal extension consists of three elements: first, the specification defines User Datagram Protocol (UDP) encapsulation formats for HIP control messages and IPsec ESP packets; second, the specification extends the HIP Rendezvous Server (RVS) to support NAT. The so called HIP Relay Server records registration information for each HIP host behind the NAT and relays their control messages to the host they want to reach;

third, the specification integrates the ICE offer/answer exchange into the HIP BEX and defines the interaction with ICE connectivity checks to discover a working path for the data flow.

The NAT traversal mechanism can be divided into three steps: first, a host behind the NAT registers its public address on a HIP relay server. A keep-alive mechanism is applied to guarantee the aliveness of this public address in the NAT; second, an initiator utilizes the relay server to delegate the whole HIP BEX (with the ICE offer/answer exchange embedded) to a responder. If the BEX succeeds, two parties have established a working path for HIP control messages via the relay server; third, based on the ICE exchange in the BEX, two parties can try to find a better path for data packets.

The ICE protocol internally uses Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) for NAT traversal. For the NAT solution of HIPv1, the ICE, STUN and TURN increases the amount of effort for the implementation. Therefore, in HIPv2 an improved specification [21] extends HIP messages and state machines to solve the NAT problem in a more native way. The new specification reuses HIP messages for connectivity checks and candidate gathering, which are previously handled by the ICE protocol.

Chapter 3

Design

For the HIPL project, HIPv2 and HIPv2.5 features are two independent tasks for different purposes. The former keeps focusing on the daemon-based solution of HIPL albeit it is entirely reusable for the latter. Based on the assessment of the current HIPL implementation and the new HIPv2 draft, we experiment with a dual-version prototype and introduce cryptographic agility of HIPv2 into the project. This way, we can validate and give valuable feedback for the HIP standardization process. Contrary to the HIPv2 design in the project, HIPL Library(HIPv2.5) innovates a library-based prototype that is not defined in any existing specification. We explore the design of this library and build its prototype.

3.1 HIP Version 2

This section presents the design of dual-version HIP and cryptographic agility in the HIPL project.

3.1.1 Dual-version HIP

In order to boost the transition from HIPv1 to HIPv2, we design a dual-version HIP for the HIPL project. The dual-version HIPL allows hosts to handle HIPv1 and HIPv2 at the same time. We believe our design might offer some insight in transitioning to HIPv3 some day if required. This section is divided into two parts: transition analysis and the dual-version design.

3.1.1.1 Analysis on HIPv1 to HIPv2 Transition

As introduced in section 2.2.1, a HIT in HIPv2 contains a 4-bit OGA to make the asymmetric key algorithm and hash function of the HIT explicit.

Although a HIT in version one and a HIT in version two have identical prefix according to the current specification of HIPv2 and ORCHID, they are incompatible because of the 4-bit OGA field. If a HIPv1 host validates a HIT of version two, it will incorrectly consider the OGA field as part of truncated hash result, thus the validation fails[1]. Similarly, the validation on a HIT of version one in a HIPv2 host also fails because the host will wrongly recognize the first 4 bits of hash results as the OGA field. Meanwhile, since the 4 bits OGA between bits 29 and 32 in version one are actually part of the hash result and can be considered as random numbers, thus it is clueless to determine the version of a HIT.

For a HIPv1 or HIPv2 host, this issue has no effect on their operations, because they are supposed to handle a certain version of HIP and to assume that HITs to process always align to the corresponding version. However, if we consider the transition from v1 to v2, this issue becomes significant due to the ambiguity.

In order to support version transition, a host must provides support in two aspects related to HITs: firstly, the host must provide at least one HIT for each version, then other hosts can choose from them based on the HIP version they support; secondly, the host must be capable to handle HITs from different versions. We conclude all possible situations for two parties that are involved in the communication in table 3.1 and find that the ambiguity of HIT formats in v1 and v2 causes transition problems.

In this table, a V1-only initiator stands for a host that is only capable of triggering HIPv1 BEX. A V1-only responder is a host that is only capable of handling HIPv1 messages. Similarly, A V2-only initiator and a V2-only responder are hosts that only support HIPv2. The dual initiator and dual responder are for the purpose of transition and able to handle both HIPv1 and HIPv2. In the table we use asterisks in the cell to mark those problematic situations. All the situations related to the transition from v1 to v2 are described as follow:

1. Cell c2, A dual-version initiator and a V2-only responder: the initiator should contact the responder using HIPv2 since the responder is running HIPv2. However, since the version of the responder cannot be inferred from the HIT of the responder, the BEX cannot be started because initiator is unable to determine which version to select for triggering BEX.

2. Cell c1 is similar to Cell c2.

[1]Strictly speaking, some special hash results can pass the validation, but the probability of getting them is so small that we can ignore those cases.

	1) V1-only responder	2) V2-only responder	3) Dual responder
a) V1-only initiator	v1	no	v1*
b) V2-only initiator	no	v2	v2*
c) Dual initiator	v1*	v2*	v1 or v2*

Table 3.1: HIPv1 and v2 Interoperability Table

3. Cell b3, where a V2-only initiator and a dual-version responder: the initiator has knowledge of two HITs (both v1 and v2) of the responder. The initiator can only start a HIPv2 BEX and expect a HIT of version two as the receiver HIT, but it is impossible for the initiator to choose the correct HIT from the two HITs that are provided by the responder.

4. Cell a3 is similar to Cell b3.

5. Cell c3, A dual-version initiator and a dual-version responder: the initiator has two HITs of the responder and it can start both HIPv1 and HIPv2 BEX, however both requires to identify the version of the HITs of the responder.

The analysis above clearly shows that a dual-version host cannot function properly in all the situations, which means a smooth transition can be hardly achieved. We reflected our worries to the HIP working group and the editor of RFC5201 agreed with us that the ORCHID prefix has to be changed in HIPv2 in order to support detection of the HIP version. Thus, this addresses our compatibility worries and we continue the dual-version design based on this assumption.

3.1.1.2 Dual-version Support

To achieve dual-version support, we first extend the handler function registry service in the modularization framework introduced in section 2.3.2. Previously, the registration service only uses the combination of packet type and current association state to index handler functions. For the dual-version

support, we add the version number as the third dimension to the key index for handler functions. By adding the version number to the criteria, enabled version two related handler functions can be adopted in the HIPL project. It also has two advantages. First, this design guarantees that the behavior of HIPv1 in HIPL is free from the side effects of the dual-version. Second, since HIPv2 is an improvement over HIPv1 and both versions share common concepts, this design maximizes code reuse in HIPL. The version two implementation can reuse most of the functions for version one, and only registers new functions if required by the HIPv2 specification.

In addition to the update of the handler function registry service, we also design the way the version number is decided and handled in the dual-version HIPL. Basically, the host decides it individually for each HIP association. Based on the version of a HIP association, the host selects the corresponding handlers for message processing. The version number for each HIP association is determined by the I1 message. Ultimately, the initiator decides and fixes the version for the lifetime of the HIP association when transmitting the I1. At the initiator side, the version to use for the initiation is determined by the HIT of the responder.

Regarding to security, the dual-version design is potentially prone to a version downgrade attack, which is one kind of man-in-the-middle attack. In this attack, a middleman deceives two hosts to use a lower version association although stronger version is available by manipulating the version of the I1 message. Lowering the version number may expose deprecated security algorithms and known vulnerabilities of the old version which can be levered by the attacker to ease cryptographic analysis or abuse the weaknesses of the old protocol version.

In the BEX, the only message can be manipulated is the I1 message, and other three messages are signed by its sender. Therefore, a middleman can change a HIPv2 I1 message from the initiator to version one for the purpose of downgrade the HIP version between two hosts. We can mitigate this attack by checking the version number when the initiator receives the R1 message from the responder. If the R1 message uses a version that is different from the one used by the initiator for the I1 message, the initiator should stop the BEX.

3.1.2 Agile Cryptographic Framework

HIP utilizes a number of security algorithms to protect its communication, including asymmetric key algorithms for HI, one hash algorithm for HIT generation, Diffie-Hellman algorithms for master key generation and symmetric key algorithms for data encryption. Some of these algorithms are fixed with-

out negotiation between the initiator and responder while some have limited options. Due to the Moore's laws and advancements in cryptographic analysis, several algorithms in HIPv1 can be considered week and vulnerable nowadays. Fortunately, HIPv2 introduces cryptographic agility to overcome this. Cryptographic agility allows protocols to adopt or deprecate security algorithms without changing the structure of the protocol itself.

HIPv2 achieves the cryptographic agility with algorithm negotiation. The protocol has added new parameters to facilitate this process. Figure 3.1 illustrates the BEX in HIPv2 with new parameters marked with bold letters. The new parameters are as follows.

The DH_GROUP_LIST parameter introduces the negotiation of the Diffie-Hellman algorithm for the two parties in the BEX in step 1.

The HIT_SUITE_LIST parameter manages the asymmetric key algorithm of the HI and the hash function to digest a HI to a HIT in step 2.

The HIP_CIPHER parameter determines the encryption algorithm that is in step 3 for data transmission after the BEX. This parameter is upgraded from the HIP_TRANSFORM parameter in HIPv1.

A R2 message in step 4 of HIPv2 is identical in format with a R2 message in HIPv1, which is used to notify the initiator the success of the BEX.

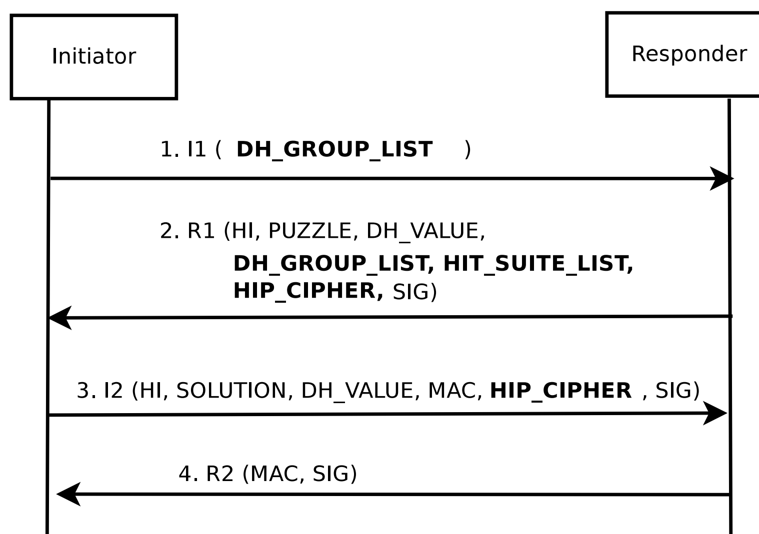

Figure 3.1: HIP Version 2 Base Exchange

In Figure 3.2 we present an example of the Diffie-Hellman negotiation to support the cryptographic agility in the HIPv2. HIPv2 defines each Diffie-Hellman algorithm as supported by the protocol with a type value. New algorithms can be included by assigning a new number. In the I1 message,

the initiator provides its own Diffie-Hellman list (DH_GROUP_LIST). This list contains type numbers of algorithms currently supported by the initiator. In our scenario, the initiator supports Diffie-Hellman algorithms number 3, 4 and 8. Upon receiving the I1 message from the initiator, the responder intersects the list from the initiator with its own list of supported algorithms and selects the strongest algorithm. In figure 3.2, the list of the responder contains algorithm types 3, 7 and 8. Algorithm numbers 3 and 8 match so the responder chooses number 8 because it is stronger than the algorithm number 3. The responder returns its complete Diffie-Hellman list along with the selected algorithm to the initiator in its R1 message. When receiving the R1 message, the initiator matches between its own Diffie-Hellman list and the list from the responder to verify if the responder has selected the strongest algorithm.

The negotiation of the Diffie-Hellman group list is the only case in HIPv2 which requires a more extensive verification of the negotiation result. The purpose of this check is to avoid a downgrade attack for the Diffie-Hellman negotiation. The I1 message is the only message in four base exchange messages without the protection of a signature, which means it can be easily forged by a middleman. The middleman can strip away the strongest Diffie-Hellman algorithms and leave weakest one in. This downgrade attack allows the middleman to conduct a cryptographic analysis on the traffic and to reverse engineer the symmetric keys used for protecting the traffic with less effort. Since the middleman cannot choose a weak algorithm not present in the list, as it would fail the BEX, the downgrade problem only affects hosts which offer weak Diffie-Hellman algorithms. To address this issue, HIPv2 enforces the responder to return full Diffie-Hellman group list for the initiator to validate. Based on the list from the responder and its own list, the initiator conducts the algorithm selection again. If the result calculated by the initiator is not identical with the one returned by the responder, the initiator will restart the BEX process to avoid potential downgrade attack.

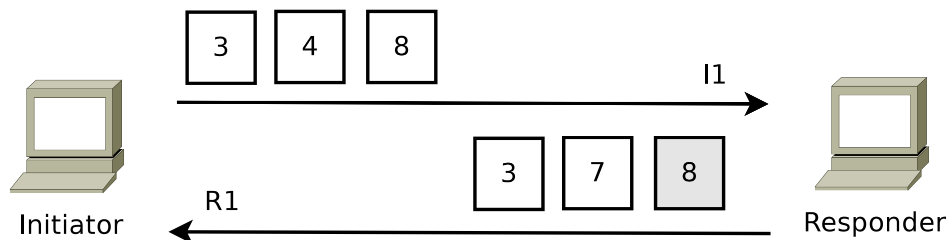

Figure 3.2: HIP Version 2 Diffie-Hellman Negotiation

3.2 HIP Version 2.5

In this section, we presents the library-based HIP approach.

3.2.1 Requirement Specification

HIP for Internet, OpenHIP and HIPL projects implement HIP using a daemon-based approach, where as HIPv2.5 is a library-based solution. HIPv2.5 shifts the logical HIP layer over the transport layer, and it can be used with programmable APIs. In the context of this thesis, We refer to this library as HIPL Library as it is based on the HIPL project. The goal of the library is to meet the following requirements:

- It can reuse the functionality of HIPv2.
- It is an application-layer protocol. The control and data plane share the same transport-layer connection (or flow in the case of UDP).
- The library operates within the process space of its caller and no extra thread or process is needed.
- The library should offer a similar API for both TCP and UDP.
- The library should provide mobility support at least for the client side and recreate the transport layer flow automatically. This includes detection of lost traffic for TCP even during handovers and data consistence guarantee.
- The API should follow the design of the Sockets API to minimize developer training costs and reduce the effort of porting existing Sockets API-based applications, libraries and frameworks.
- The library should not require higher privilege.

The system architecture is illustrated in figure 3.3. The dashed lines symbolized the APIs between two layers. HIPL Library is located above the Sockets API. It uses standard Sockets API and presents its own APIs for the voluntary applications. Applications can use its functionality either indirectly or directly: and application can utilizes an IoC Library wrapper which provides event-driven style APIs or alternatively the application can be integrated directly on top of HIPL Library APIs[2].

[2]IoC Library is stated in details in the Section 3.2.4.

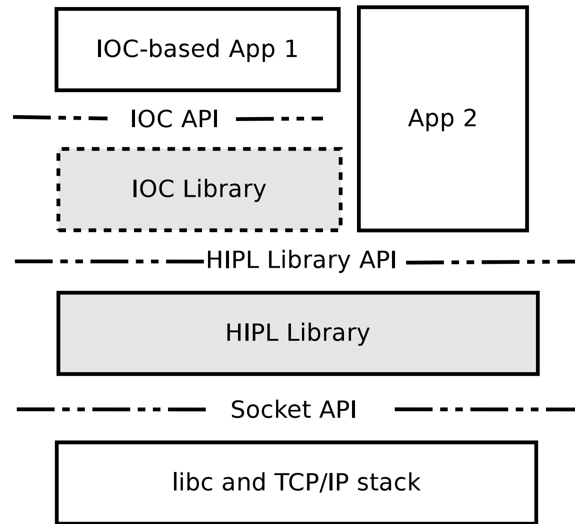

Figure 3.3: System Architecture of HIPL Library

Instead of the coarse interposition approach as Reliable Sockets (ROCKS) described earlier in section 6.1.2, HIPL Library provides new APIs to applications. This way, the applications are aware of the underlying security mechanisms taking place. Figure 3.4 illustrates shifting HIP layer above the transport layer. Thus, HIP control and data plane packets are encapsulated inside TCP or UDP packets. This relocation together with the library-based requirements, introduces many new challenges to the design of the HIPv2.5: first, the port numbers from transport layer need to be managed by the library; second, the library requires a new mechanism to demultiplex HIP control-plane and data-plane messages arriving at the same source port; third, the library-based solution uses single thread and does not run on a dedicated process, which means the library should handle events only when the application passes control to the library (by calling functions of the library); last but not least, the mobility mechanism for the transport layer is, especially for a wrapper style library, entirely different. In the following sub-sections, we present a legacy compatibility analysis and address these challenges one-by-one.

3.2.2 Legacy Compatibility Analysis

The main purpose of the library is to improve the deployability of HIP. If the library can offer better backward compatibility to reduce the migration cost, more applications may be willing to use HIP. We evaluate the compatibility of the library in four cases: 1) client and server that are both use the library,

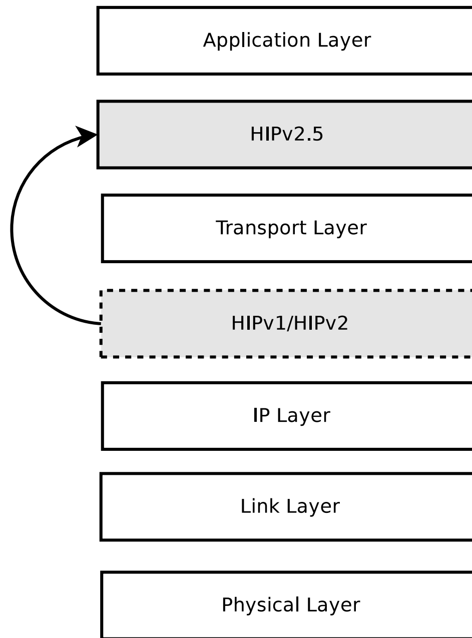

Figure 3.4: Protocol Stack Change from HIPv1/HIPv2 to HIPv2.5

2) legacy client and server that do not use the library, 3) legacy client but server on top of HIPL Library, and 4) client on top of HIPL Library but legacy server. We focus on the last two cases as the first two cases do not have compatibility issue.

The library can support the third case[3]. The server can notice that client did not start a BEX and continue in "by-pass mode" where it abstains from processing and passes all function calls through.

The last case can be further divided into two sub-scenarios: opportunistic BEX and normal BEX. The opportunistic BEX is problematic because the client does not look up the server identity from anywhere, so it cannot know if the server is HIP capable. It could be resolved by timing out in the library and then falling back to normal TCP/IP. A more optimal way with TCP options are described by Bishaj et al [7]. The other sub-scenario of normal BEX is an easier case because the client has resolved the host name of the server using some mechanism and should know based on result whether the server supports HIP or not. A legacy server results in a IP address, so the library should fall back to normal TCP/IP and restrain from all processing[4].

[3]Not yet available in current library prototype
[4]Naturally, the opportunistic mode could be tried here but it is a policy question.

3.2.3 Demultiplexing of the Control and Data Plane

HIPv2.5 control and data plane share faith with the same transport layer end point for a single HIP association. This behavior differs from protocols such as File Transport Protocol (FTP) which uses one port for FTP commands and one port for data transmission. The character stream oriented nature of TCP makes framing between control and data plane messages a bit more challenging. When using TCP, a host can possibly receive: 1) half part of a control message or a user message; 2) several consecutive control or user messages; 3) a mix of condition one and two. When the underlying transport-layer protocol is UDP, the boundary between a control and data plane message is more natural as it is defined by the scope of each UDP message, since an UDP packet contains either a control or a data plane message.

To demultiplex control plane and data plane from the same TCP stream, we have considered two possible solutions. The first alternative is to encapsulate the data-plane message in HIP message body as a HIP parameter. The second alternative is to create a new header for the data-plane message.

Solution one requires a HIP header for each data-plane message. It has several advantages: first, it can conform to the HICCUPS [34], which provides a standard way to encapsulate application data into a HIP message. Second, the HIP message format is a flexible structure and it allows simple future extension. The drawback of this solution is the lowered Maximum Transmission Unit (MTU). A HIP header is already 40 bytes long, and except the packet length field, most of other fields are not necessary for HIPv2.5.

As the library-based model is already incompatible with standard HIP, we chose to create a new header that should be more compact than a full HIP header. The new library header is only four bytes long. Figures 3.5 illustrates the encapsulation of HIPv2.5 datagrams.

For control-plane messages, the four bytes of library header are always zero. The size of the HIP message body can be determined by the HIP header next to the library header. This format conforms to RFC-5770 [28], which encapsulates control messages into UDP for NAT traversal purpose. Thus, the UDP-based Control plane remains compatible with the relay extensions.

For data-plane messages: the four-byte library header stores the size of the data-plane message in the network byte order. Therefore, the maximum size of a data-plane message is 4GB (2 power 32), which should be far more than enough for normal cases. The 4GB size limitation also confirms to the IPv6 Jumbograms specification for MTU larger than 65575 bytes [10]. Once the receiving side receives a library header with a value more than zero, it expects a data-plane message with the length specified by the library header.

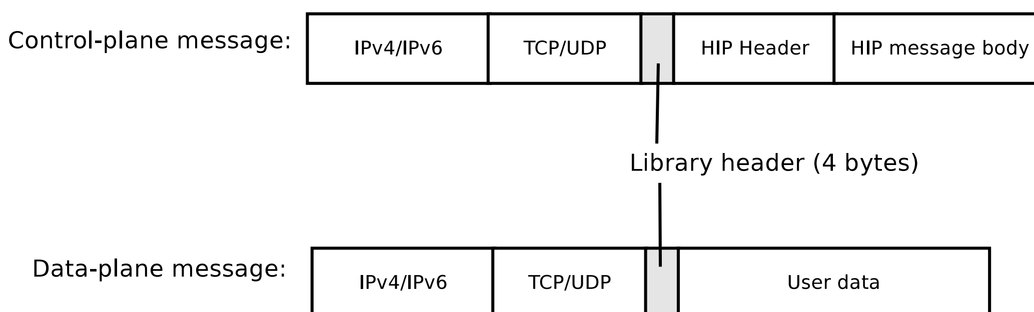

Figure 3.5: HIPv2.5 Packet Encapsulation

3.2.4 API Design Alternatives

For a networking library or framework, there are at least two design alternatives for the API. A common approach, such as adopted in OpenSSL, is to provide a wrapper API around the Sockets calls. Another is called the Inverse of Control (IoC) or which is sometimes referred as event-driven, and a typical example for this breed of libraries is Twisted[5]. The difference between these two designs is the owner of the right of control. In the wrapper case, the application is in control of the flow of execution, whereas in IoC the underlying library takes over. We refer the former one as a wrapper library design, because applications are in the control of themselves and they decide the timing of passing control to the library by calling its functions. Conversely, in an event-driven design, applications register callback functions first, and pass the control to the library. After this, the library determines when to call those registered functions. Typically, event-driven applications employ threading to sustain concurrent operation.

The most imminent obstacle for a wrapper style HIPL Library is that the timing of handling HIP control-plane messages and conducting connectivity check is non-deterministic. A daemon-based solution runs in its own background process, and similarly an event-driven solution also has full control of the process. Both of them can perform connectivity change detection at any time, but this is not the case for HIPL Library as it relies on applications to pass the control to it.

A reason for choosing a wrapper library design in HIPL Library is flexibility. A wrapper library is generic and can be the foundation of an event-driven library. However, the main reason is that porting of existing applications based on the Sockets API is more straightforward with the wrapper API than with event driven.

[5]http://twistedmatrix.com/

In HIPL Library, the BEX is triggered when an application uses the library for data operations (sending or receiving) for the first time. An application acts as a HIP initiator if it sends data. Correspondingly, an application acts in the role of HIP responder if it is receiving data. The library delay data transmission until the potential transport-layer handshake and the BEX are completed between two end points. The BEX is hidden transparently by the library.

Figure 3.6 illustrates the BEX in the library. In the first step, the two end points establish a transport-layer connection by using the API from the library. This step is not necessary for UDP but mandatory for TCP (a three-way handshake). In the second step, the application on the left side requests to send N bytes of user data. The left-side application has been identified as a HIP initiator. Then the library triggers a BEX in steps 3, and it is completed in steps 4, 5 and 6. Once the BEX finishes, the library delivers the N bytes of data as requested by the application.

After the establishment of the HIP association, most of messages between two end points are data-plane messages. Occasionally, an end point sends HIP UPDATE messages to inform location changes or sends a HIP CLOSE message to shutdown the current HIP association. For an UPDATE message, the library updates the new location transparently from the application. For a CLOSE message, the library notifies the application of the termination of the HIP association when the application attempts to continue data operations.

In order to support the mobility, the control plane is also responsible for tracking the connectivity status. The detail of the connectivity detection and mobility handover process at the control plane is presented in the following section.

3.2.5 Mobility

When the library determines that a connectivity change occurs, it should initiate a handover, create a new transport-layer flow and restart data-plane transmission. The library assumes that mobility can only be supported between two associated applications, which means that a connectivity change in the middle of the BEX will not be considered as a valid mobility scenario[6]. As a wrapper library, it can only performed the connectivity status check when it is in control of the flow. If an application has a large interval between function calls, the interval for connectivity status check is also correspondingly long. A connectivity change during this interval also affects

[6]HICCUPS extensions [9] may be more suitable in such scenarios.

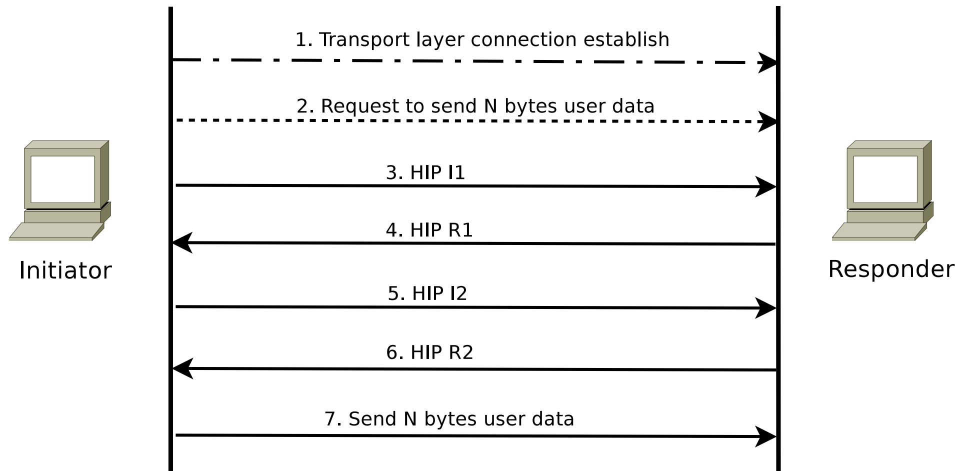

Figure 3.6: HIPL Library BEX Process

the remote side, which has to wait for the new status notification from the application in order to continue data transmission. This is a limitation of the wrapper library design, but it is possible to be fixed by an event-driven library on top of the wrapper library.

The mobility support of the library allows it to react to connectivity change transparently, without assistance of the application. The library overcomes mainly three problems to achieve mobility support: firstly, the library needs to detect connectivity status changes with limited control of the whole application and user level privileges; secondly, the library needs to create a new mechanism to process transport-layer handover; lastly, the library should guarantee the data consistency for TCP after the handover.

3.2.5.1 Buffer Management and Data Consistence

The library maintains two buffers for each TCP or UDP end point. Typically, the network stack of a OS has also its own buffers. In order to distinguish between the two terms, we refer to the input and output buffers in the library as "library input and output buffers" and call the input and output buffers managed by the network stack as "kernel input and output queues".

The library input buffer is used to demultiplex the control from the data plane for a streaming protocol such as TCP. The library appends the receiving data to the library input buffer and then follows the parsing mechanism as described in the section 3.2.3 to distinguish between control and data messages. If a control or data message spans two or more TCP packets, the library input buffer can detect it by checking the library header in the

payload, and waits for the remaining part of the message.

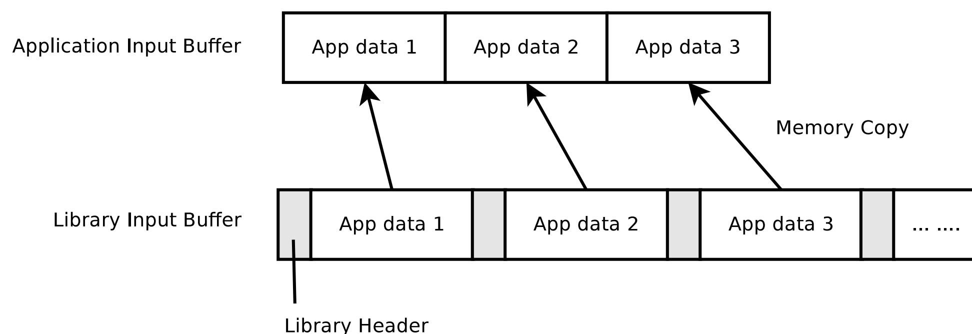

Figure 3.7: Application Data Assembling through the Library Input Buffer

The library input buffer can also assemble consecutive data-plane messages to make the whole receiving process more efficient. As illustrated in figure 3.7 (the gray parts in the library input buffer is the library header), if there are several data-plane messages residing in the library input buffer side by side and the application offers abundant application buffer to hold them, the library can expedite the receiving process by striping of all the library headers and aggregating application data into the application buffer in one function call. In the figure, the library delivers the application data one to three from three consecutive TCP packets to the application in one function call. This manipulation is valid because the application has the same character-oriented assumption on the receiving data.

On the other hand, the sending side of the library needs to overcome another issue of potential data loss in the kernel output queue. The library is based on the Sockets API and if TCP is used, the socket library will inform the application that data is sent as long as there is enough size to hold sending data in the kernel output queue. It is the responsibility of the network stack to deliver those data in the kernel output queue but when the host locator changes, the network stack does not assure the data remaining in the kernel output queue to be delivered. For this reason the library includes a output buffer.

The library output buffer stores undelivered application data. To be more exact, the term "undelivered" here refers to the data that has not received by the remote side. Then the data in the kernel output queue will be considered as undelivered and cached in the library output buffer. The buffer guarantees that data not received by the remote side will not be dropped if a connectivity change occurs.

In the context of the library output buffer, the sending process of the library can be divided into 4 steps: firstly, the application initiates a sending

action by calling function of the library and provides the data to be sent; secondly, the library gains control of the flow and checks the delivery status of existing data in the library output buffer. If some data is already successfully delivered, the library removes it from the output buffer. Otherwise, the library starts a timer for the handover. If the data remains undelivered until the timer expires, the library triggers the handover; Thirdly, the library appends the new data as provided by the application to the end of the output buffer and the data is then tracked by the library; lastly, the library uses a standard Sockets API call to deliver the new data.

The library output buffer plays a significant role for mobility support. In the following two sections, we explain the connectivity status detection and handover mechanism.

3.2.5.2 Connectivity Status Detection

Section 3.2.4 explained the timing of connectivity status detection in a wrapper style library design such as HIPL Library. This section explains the detection method. As a user-level library, HIPL Library can only use user-level APIs, which drastically limits the range of mechanisms it can utilized for connectivity status detection.

The library relies on the relationship between its output buffer and the kernel output queue for connectivity status detection. Although retrieving the content of a kernel output queue is not supported by most of network stack implementations, querying the data size in the queue is allowed, which can be a good hint for the library to detect a change in connectivity. Assuming that two applications A and B are using the library to communicate with each other, the connectivity status detection in the host A includes following steps:

1. Application A calls a sending function in the API to transmit N bytes data to application B. The library appends the data to the output buffer and updates its counter.

2. The library calls a function in the Sockets API to send data to application B. The N bytes of data is stored in the kernel output queue and tracked by the network stack.

3. When the library obtains control from the application for the next time, it first checks the size of the kernel output queue. If the queue size decreases by K bytes, this K bytes of data is delivered by the network stack and the library removes it from the output buffer. Otherwise, the buffer remains unchanged.

4. If the out buffer is not empty and the library cannot remove data from it for a certain period of time, the library consider the current transport-layer connection is broken and it performs a handover.

The detection mechanism is efficient and suitable for applications that has a frequent request and reply pattern (for instance, Telnet) as it can track connectivity status with only small performance overhead. However, it also has two drawbacks. First, an "download"-oriented application that receives data constantly but seldom has data to send, hardly benefits from this mechanism because the utilization of the library output buffer is too low to trigger the detection mechanism. Second, the precision of the timer to trigger handover greatly influences the handover performance. The library should offer a reasonable default time but allow the application to readjust it according to its use scenario. If the time value is too long, the user determines that the connection is dead. If it is too small, the library may trigger handover on a functioning but temporarily congested connection.

The fixed timeout could be improved by collecting cross-layer link statistics dynamically, and use this as an active feedback loop to readjust the timeout value. For instance, TCP congestion control estimates the average Round Trip Time (RTT) as an indication of the connectivity status. Since this library is in a prototyping phase, we keep the design simple and use a fix timer. In the future, an adaptive timer mechanism can be added to the library without affecting its current functionality.

In addition to monitor the size of the kernel output queue for connectivity status detection, there are two alternative solutions. First, The library can track the information of all network interfaces in a host and evaluate whether a new change deprecates current transport-layer connection, thus requiring a handover. The library can perform the interface tracking during the execution of any API. A second solution is the IoC version of the library. In this version, the library is in control of the program flow and schedules events, thus the library can perform connectivity status check in a periodical manner.

3.2.5.3 Handover

The handover support in the library is constrained to the client side because we assume that servers are always standard-alone machines and rarely relocate while mobile devices usually act as clients. This does not mean that the library is unsuitable for multihoming servers or P2P networking. The client-side handover just means that the initiating side is responsible for triggering the handover to avoid the communication disconnection between two hosts. The client side handover mechanism can be divided into

three stages, transport-layer migration, update message exchange and buffer retransmission.

The core of the handover is to renew the transport-layer end point. If the library detects a connectivity change, it assumes that the previous end point is broken and creates a new one to substitute the old. The library uses the new end point to communicate with the same remote side, to update its new information and finally to retransmit the remaining data in the library output buffer. Figure 3.8 illustrates the process of handover which consists of three HIP UPDATE messages adopted for the library from RFC5206 [33].

First, the initiating side detects a connectivity change, creates a new transport layer end point and triggers the next stage which sends a HIP UPDATE message to the responder. The new end point is created without specifying a local address thus the network stack can choose the most suitable address for handover. We assumes that the responder is able to accept multiple simultaneous transport-layer sessions, otherwise the handover cannot continue.

Second, the responder receives the first UPDATE message on the new transport-layer end point. From the message header, it determines that it is a handover request from an known initiator. The responder processes the packet and replies back the second UPDATE message.

Third, the receiving of the second UPDATE message in the initiator side indicates that the responder can be reached through the new transport-layer end point and it has accepted the handover request. Then the initiator replies with the third UPDATE message as a confirmation and retransmits data in its library output buffer. If the output buffer of the responder is not empty, the responder also starts buffer retransmission at this point. After that, both parties can discard the previous transport-layer end points and transition to the new pair of end points.

The insecure handover process is validated by the secure UPDATE messages that are protected by MAC and signed with the public key of the originator. The message authentication is crucial because otherwise a malicious application can hijack or mount a DoS attack on a session by injecting a forged UPDATE message to the responder side.

Three parameters are involved in the update message exchange. They are Sequence (SEQ) parameter, ACK parameter and Synchronization (SYNC) parameter. The SEQ and ACK parameters are standard parameters for a HIP UPDATE message and they are used for the purpose of reliability and ordering of the control plane. For TCP mode in the library, these two parameters are redundant because TCP is already a reliable protocol with guarantees of ordering. However, they are needed for the update message exchange with UDP mode to guarantee the two properties for the control

plane. The library retains these two parameters to conform to the HIP standards and to unify the handling of TCP and UDP in the library. In addition, the two parameters are secured with public-key signatures, which is not supported by TCP.

The SYNC parameter is an extension defined by the library to support buffer retransmission. The parameter contains two values: the Number of bytes Sent (NSENT) and the Number of bytes Received (NRECV). The values represent the number of bytes sent and received as observed by the sender of the parameter[7]. Both of the values are currently 64 bits which exceed TCP window size and is probably enough for UDP-based applications. The values are exchanged to synchronize two connected parties, so that they can determine what part of the buffer needs to be retransmitted.

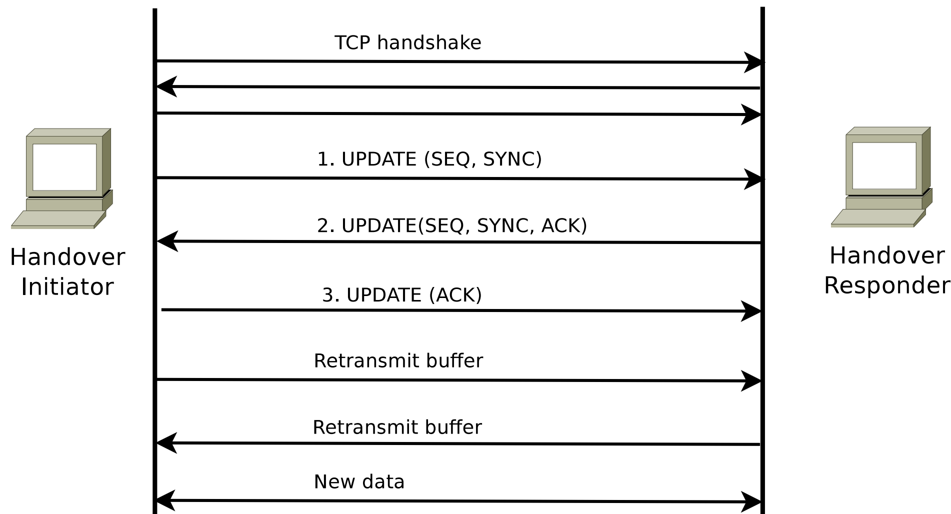

Figure 3.8: HIPv2 Handover

The SYNC exchange is designed to ensure correct data delivery during handovers because vanilla TCP does not support "migration" from one connection to another, let alone UDP. It is obvious that all data left in the library output buffer during handover should be re-transmitted. However, as explained in section 3.2.5.2, an handover can occur on a temporarily congested path, and the new transport layer end point created for the handover process continues using the same path. In this case, the previous connection is not broken and chances are that unsent data in the corresponding kernel output queue will be delivered by the network stack while our handover pro-

[7]In the future, the values could also be used to implement forward/rewind functionality for media streams.

cess. Without the SYNC exchange, the library may re-transmit the same part of data, which causes the remote side to receive duplicated data.

3.2.6 Comparison between Sockets and HIPL Library API

Our library API is distinct from Sockets API in three aspects. First, the library has a "hipl_" prefix for all the function calls. Second, the initialization can be done without explicit DNS resolution as the HIPL Library API supports host names. Third, our library does not support file descriptors but has a similar concept - the new descriptors cannot be used in read/write calls unlike the socket descriptors[8].

[8] The implementation section details the new descriptor used in our library.

Chapter 4

Implementation

In this chapter, we present the implementation of HIPL Library, which includes the function hierarchy design, data structures, solutions to support the *select()* function and handover implementation.

4.1 Overview

The HIPL Library offers APIs mimicking the Sockets APIs that has several alternative functions for the same purpose. For instance, the *send()*, *sendto()* and *sendmsg()* functions are all for sending of data, but are optimized for different use scenarios. Our library follows this style but internally uses a function hierarchy to build those APIs.

The hierarchy is illustrated in figure 4.1: at the bottom part of the figure there are four core internal functions: *buffered_sendmsg()*, *buffered_recvmsg()*, *hipl_sendmsg_internal()* and *hipl_recvmsg_internal()*, which implement the core features and form the basis for all public APIs. The first two functions handle the library input and output buffers during the data transmission while the last two, *hipl_sendmsg_internal()* and *hipl_recvmsg_internal()* are responsible of the BEX and handover detection. The top layer consists of public APIs, and their main task is sanity checking. The purpose of this hierarchy is to modularize the implementation.

A pair of local and remote end points for HIP association in the library is referred as a HIPL Socket (HSOCK). Similar to the socket descriptor, each HSOCK has its own identifier. This identifier can be used to access the public APIs of the library. However, contrary to a socket descriptor, this identifier is only a reference number to a particular HSOCK and it does not support any file related operations. The identifier acts as a bridge between applications and the library, hiding all internal complexities and shielding

the internal data structures from inadvert modification.

When the library is used in the TCP mode, several additional functions have to be supported. For instance, the *hipl_connect()* function is for establishing TCP connection. The *hipl_listen()* function transitions a HSOCK into listening mode. The *hipl_accept()* function accepts a new TCP connection from its client and peals off a new HSOCK from the old one.

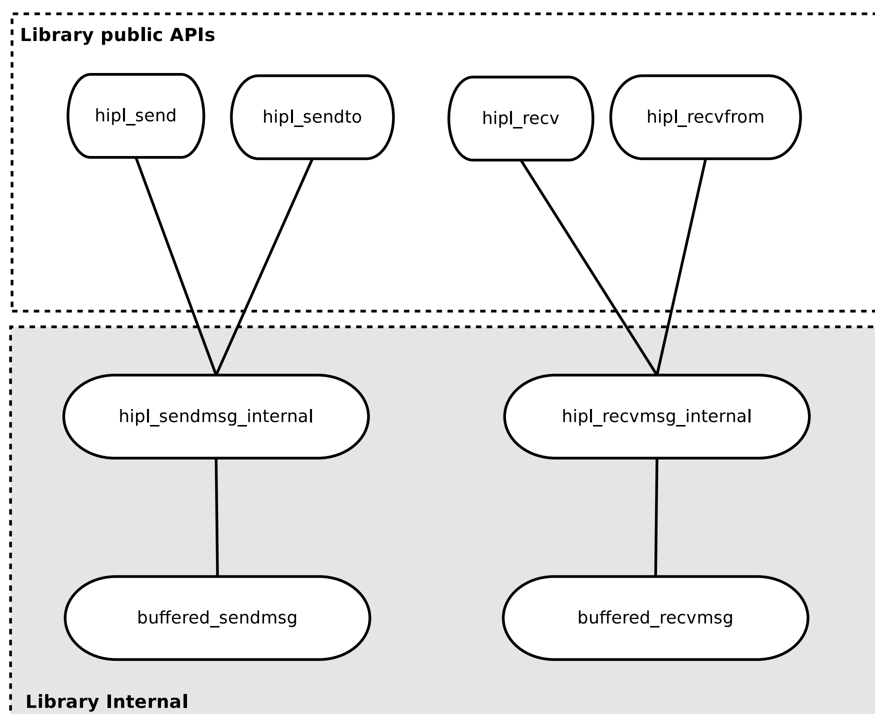

Figure 4.1: HIPL Library Function Hierarchy

Internally, the library maintains a hash table of HSOCK-related structures, where the identifier of HSOCK as the hash key. Table 4.1 lists the members of a hash value and describes their purposes.

4.2 Select Support

All Portable Operating System Interface (POSIX) compatible systems support synchronous I/O demultiplexing via functions such as the *select()*. The *select()* function allows user to monitor a list of file descriptors, and get notification from the OS when one or more descriptors are ready for further reading or writing operations. To support the *select()* and other similar calls, the library needs to solve two problems.

Name	Description
sid	The identifier of the HSOCK. Currently the library supports no more than 65535 (16-bit) HSOCKs simultaneously for a single application.
ha	The HIP association status
peer_locator	The current IP address of the associated application
peer_hit	The HIT of the associated application. It is obtained when the *hipl_connect()* function is called. For UDP, the availability of this information is delayed to the first data retrieval operation.
sock_fd	The transport-layer end point currently in use
ho_sock_fd	The transport layer end point that is created for the handover process. When a handover is completed, the value of the filed *sock_fd* is replaced by this value.
sendbuf	The library output buffer, which is a circular queue implementation for fast data processing
recvbuf	The library input buffer, which is a queue implementation for fast data processing
stat_ts_send_sz	The time stamp of last successful data delivery. This field is used for connectivity status detection together with the field *cfg_unsent_to*.
stat_recv_sz	A 64-bit field which records the total number of the bytes received by this HSOCK
stat_send_sz	A 64-bit field which records the total number of the bytes delivered to the associated application by this HSOCK
cfg_unsent_to	The timer threshold to trigger a handover process. If data in the library output buffer is unsent until the timer expires, the library will trigger a handover.

Table 4.1: Members in the HSOCK Data Structure

First, TCP handshake consists three messages and the BEX procedure consists of four messages. Handling all of these packets in one call of the *select()* function is impossible because it returns when the first packet is available. For instance, Assuming that a responder is waiting for an I1 message from a initiator. It first uses *select()* to check whether its channel is readable. If yes, it processes the I1 message and then sends a R1 message to the initiator. At this point, it expects an I2 message from the initiator. However, the responder must not read the channel immediately, because there is a high chance that the I2 message is not yet delivered by the initiator and calling a read function on the channel will block the whole process.

Second, the handover process creates a new socket to recover the communication with the server. The server peels off a new socket using the *accept()* function. The library needs to find a mechanism to handle this new socket at the library level.

In the following two subsections, we describe solutions for the two problems.

4.2.1 Delayed BEX

The library provides a *hipl_connect()* function to establish a transport-layer. It is mandatory for TCP and optional for UDP. The BEX is triggered either during this function call, or during the first data related operation (read or write). For UDP, the latter is supported by the library, because *hipl_connect()* function is optional.

For TCP, the connection phase also has a blocking issue at the responder side because it requires the library to handle both the TCP handshake and BEX together in *hipl_accept()* function. Thus, the wrapper function *hipl_accept()* could still block despite its has internal *accept()* call has returned a new socket descriptor successfully.

In the implementation of the library, we decided to delay triggering of the BEX until the applications sends the first data bytes in order to keep TCP and UDP symmetric and support *select* function. The *hipl_connect()* and *hipl_accept()* are used only for the purpose to create a transport-layer connection. This decision also simplifies the implementation logic of the library, since *hipl_sendmsg_internal()* is the only one place to trigger BEX.

4.2.2 On Blocking Operations

Delayed BEX solves the blocking problem of handling the TCP connection establishment and BEX together after a *select()* call. However, the handover

and BEX procedures still have the the problem of handling multiple messages within a single *select()* call.

To support the *select()* function, the library provides an non-blocking option to configure its behavior. The library can block all the way until the BEX or handover finishes. Alternatively, The library can support non-blocking operation during the BEX or handover process, thereby supporting asynchronous calls.

The library divides a long process that involves handling multiple messages to several short operations each of which only handles one message. For each short operations, the library returns a status code to the application and the long process can be resumed at a later time. Using Figure 3.6 as an example, the BEX process at the responder side includes 6 steps which involves three *select()* calls:

1. The responder call the *select()* function for the first time to monitor the socket, and it receives a notification that the socket is available to read.

2. The responder calls a reading function to retrieve the data. The call is handled by the library and it first reads the I1 message. It responds with a R1 message to the initiator, saves state and returns control flow to the caller with a notification that the library is processing the BEX.

3. The responder calls the *select()* function for the second time to continue monitoring the socket and receives another notification for availability.

4. The responder continues with the library read operation again. The library retrieves the I2 message and responds with a R2 message to the initiator. Finally the library returns a notification to the caller that the BEX is finished.

5. The responder continues monitoring the socket and retrieves the third notification for read availability.

6. Finally, the responder receives application data from the initiator.

The non-blocking mode returns status of the current ongoing process (BEX or handover) using different status codes. Table 4.2 lists all of the codes, their scenarios and descriptions. Non-blocking applications can check the return code of the library for the progress of the BEX or handover process.

Code	Description	Scenarios
139000	EWAITBEX The initiator or responder is processing BEX.	1) I1 is sent by the initiator. 2) Initiator is waiting for R1. 3) Initiator is waiting for R2. 4) Responder is waiting for I1. 5) Responder is waiting for I2.
139001	EBEXESTABLISHED The HIP association is established.	1) Initiator has processed R2. 2) Responder has processed I2.
139100	EHOCOMPLETED The handover is completed at the client side.	Handover, client side has processed the second UPDATE message.
139101	EHOSOCKSWAPPED The handover is completed at the server side.	Handover, server side has processed the first UPDATE message. The current HSOCK is discarded by the library because it is only for handover purpose.

Table 4.2: Non-blocking Status Code

4.3 Handover

To demonstrate the handover process, Figure 4.2 shows an example scenario of a client-side handover in detail. The operations marked with solid line are executed by the application while the operations marked with dotted line are internal operations of the library. The handover consists of nine steps from 1 to 9 and each step is further divided into sub-steps marked by alphabetic letters.

- In steps 1 and 2, the server side is set up and ready to serve incoming connections. The server application first uses *hipl_listen()* to switch its HSOCK to the listening mode (step 1-a). Then it is blocked in the *hipl_accept()* function and it is waiting for incoming connections (step 2-a). Internally, in steps 1-b and 2-b, the library utilizes Sockets functions *listen()* and *accept()* to operate the socket.

- In step 3-a, the client application calls *hipl_connect()* to reach the server. The library calls the Sockets function *connect()* to establish a transport-layer connection in step 3-b. At this point, the client side only initiates a normal three-way handshake.

- In step 4, the sever side accepts the new TCP connection, then the server application calls the *hipl_recvfrom()* for receiving data from the client side.

- In step 5-a, the client application executes its first data transmission via *hipl_sendto()* function. Since there is no existing HIP association between the two applications, in step 5-b, the library at the client side triggers a BEX procedure with the server side and they establish a HIP association. In step 5-c, the library delivers the data to the server side.

- Between steps 5 and 6, the client relocates and switches to a new IP address.

- In step 6-a, the client application requests a data sending again which fails due to the relocation. However, since the handover timer does not expire and the kernel output queue are not full, the sending operation returns successfully. The client application keeps sending until the handover timer expires. The library starts the handover process via the three UPDATE messages in step 6-b. After that both sides retransmit undelivered data from their buffers in steps 6-c and 6-d.

- After the handover, both the client and server have successfully migrated to a new transport-layer endpoint and the communication channel is recovered. In steps 7, 8-a and 8-b, the server application receives data again from the client application.
- In step 9, the client side terminates the connection with the *hipl_close()* function.

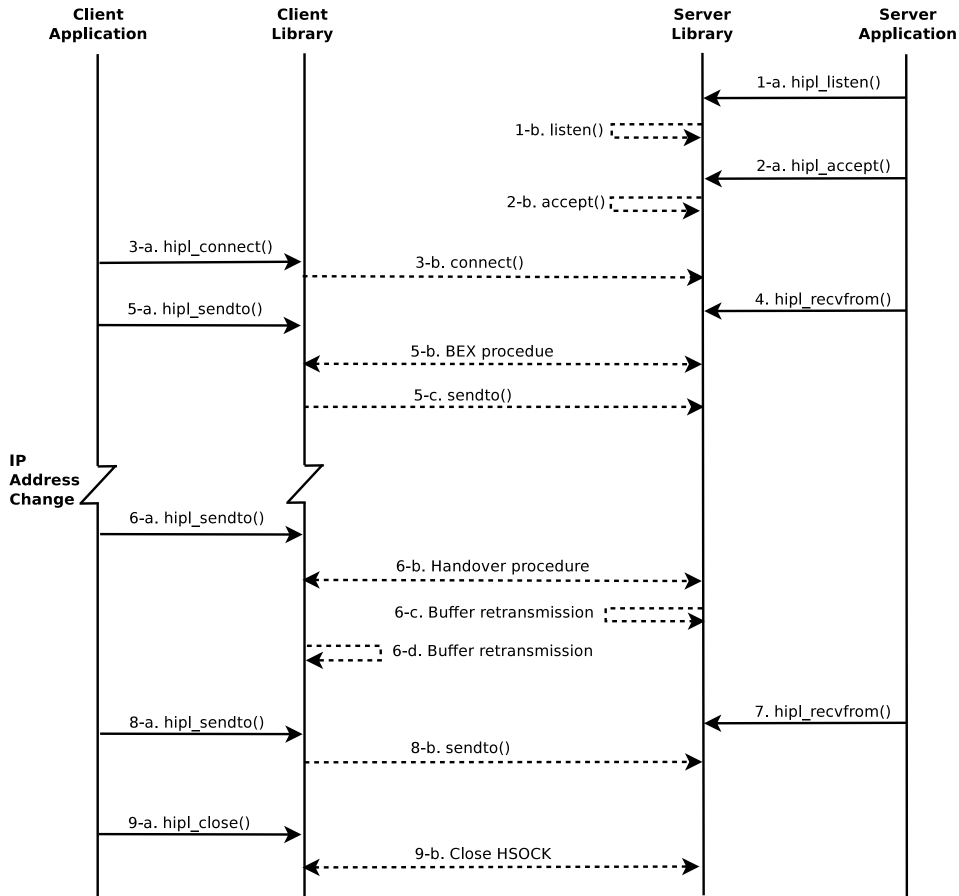

Figure 4.2: A Client-Side Handover

Chapter 5

Experimentation

To validate the prototype, we have implemented two applications to test the library from different perspectives. The first application is a system tester that was adopted into the HIPL project to facilitate rapid testing for developers. This software does not require administrative privileges and it is based on the same code as the daemon which makes it very useful for system testing. The second program is a HIP version netcat[1]. Written from scratch, Netcat is a generic-purpose tool that can be used for sending any kind of data, albeit typically it is used for echoing of UNIX standard or keyboard input.

5.1 System-test Application

To ensure the quality of the software, developers in the HIPL project have implemented many unit tests that are automatically executed after each source code commit on the main branch. However, the unit tests only verify code at the function level and system testing to verify the interaction between different function execution traces is missing.

HIPL Library fulfills this gap. The library itself uses the same modularization framework as the HIPL daemon process. The same function chains (except for the functions for IPsec, which require root privilege) are also reused by the library. Thus, processing of BEX in the library and the daemon can be considered to be almost identical. If the BEX fails in the library, it should also fail in the daemon, thereby making the library a useful system-test platform.

Utilizing the API of the library, we built a single-threaded program. The test program includes BEX on top of TCP and UDP. As the HIPL com-

[1] http://nc110.sourceforge.net/

munity preferred a singled-threaded and single process design, the program interleavingly acts as client and server and uses the library API to establish HIP association with itself through the loopback interface.

The library is based on tokens that resemble socket descriptors but cannot be used as a replacement in standard Sockets API calls. Thus wrapper functions need to be utilized in all socket calls, including *select()*[2]. The wrapper function for *connect()* has another implementation limitation as it is currently supports only non-blocking operations. If the application bypasses the abstraction tokens and utilizes the socket descriptors directly, the handovers of the library are ineffective.

The test program is merged to the main branch of the HIPL project and has been reviewed by the community. It is also compiled and executed for the Scratchbox[3] Advanced RISC Machine (ARM) cross-compilation environment for the Maemo[4] 5 platform, which demonstrates its potential for mobile devices.

5.2 Hipnetcat

The Hipnetcat is a character transfer application supporting both client and server functionality based on the library. The software is not only a guide for developers who want to use the library, but also a diagnosis tool for the developers of the library. The software can be both TCP and UDP-based. The synopsis of it is described in appendix B. We also provide two example runs to demonstrate the usage of hipnetcat, one for TCP and one for UDP in the appendix.

The client part of the hipnetcat captures data from the standard input and relays it to the server. The server can be another hipnetcat application or any application built on top of the library. Hipnetcat echoes responses from the server to the standard output of the client.

If hipnetcat is running as a server, it reads data both from the standard input and from the client. For the data from the standard input, the software then sends it to the client. For the data from the client, hipnetcat echoes it to the standard output.

Hipnetcat utilizes HIPL Library to support mobility. The handover is transparent at the client side. For the server side, hipnetcat uses the *select()* function to monitor multiple sockets, thereby being able to accept the handover request from the client and recover their communications.

[2]which remains to be unimplemented

[3]http://scratchbox.org/

[4]http://maemo.org/

Chapter 6

Related Work

6.1 Mobility

In this section, we describe various mobility solutions and compare the difference between each of them and our library approach.

6.1.1 Mobile IP

Mobile IP [37] can be considered as a predecessor of HIP, laying a foundation for improvement in HIP. Mobile IP uses a pair of IP addresses, one acts as a static identifier and the other acts as locator. HIP has adopted a similar approach of splitting the locator and identifier roles, but uses a public key from an asymmetric key pair as identifier to provide built-in security mechanisms. Compared to Mobile IP, which operates on the network layer, our library approach is located on top of transport layer and does not requires extra infrastructure support or configurations.

6.1.2 Reliable Sockets (ROCKS)

ROCKS [52] is a library-based solution for mobility. ROCKS is based on interposition and it intercepts function calls to Socket APIs to add its own mobility mechanisms. ROCKS supports TCP and it introduces a new protocol for its control plane based on Diffie-Hellman exchange. Moreover, ROCKS utilizes internally another UDP-based socket for the control plane.

Although the HIPL Library is also based on the Sockets API, it provides new APIs instead of relying on the library interposition. The interposition in ROCKS could cause integration problems if the application also utilizes the integration mechanism. This kind of problems are tricky and hard to debug.

The HIPL Library cannot achieve zero-modification migration as ROCKS does, but it avoids compatibility issues in library interposition.

The control and data plane in the HIPL Library are coupled, meaning that they share faith. In contrast, ROCKS control and data plane do not share faith, which has some negative consequences. First, firewalls may drop its UDP-based control plane. Second, NAT penetration needs to be done twice, one for each plane. Regarding to security, HIPL Library conforms to HIP which is standardized, while the new protocol introduced by ROCKS still needs validation.

6.1.3 Multi-Path TCP

Multi-Path TCP (MPTCP) [45] takes advantage of multiple available paths to support multihoming and load balancing. From the mobility point of view, Multi-Path TCP (MPTCP) is ideal if multiple paths are available because it can detect a connectivity failure and switch to a better path. However, the problem of intermittent failures on a single path are not in the scope of MPTCP, thus it does not support mobility.

An implementation of MPTCP for Linux[1] is located in the kernel space and thus it requires a new kernel image. Once the customized kernel is ready and MPTCP is configured properly, no modification is required for applications. In the future, MPTCP may be available in vanilla Linux kernels.

MPTCP can be considered as a supplementary solution for HIPL Library. If the library is used with a MPTCP-enabled stack, it can solve the mobility issue for a single path while MPTCP offers a more optimal solution for multihoming. Therefore, the integration of the library and MPTCP has engaged interests from MPTCP community.

6.1.4 DTLS Mobility

The DTLS mobility [46] is a library-based solution for the mobility of the DTLS protocol [42]. To support mobility, the authors add several extensions on the DTLS protocol, such as Connection Identifier (CID) to distinguish between connections, utilization of heart beat extension for handover, peer validation and path MTU discovery.

The DTLS mobility is bound to the OpenSSL APIs, which requires some effort to migrate a Socket-based application but is effortless for applications that already use OpenSSL. Unlike HIPL Library, which supports both TCP and UDP, DTLS mobility only offers mobility support for datagram packets.

[1] http://mptcp.info.ucl.ac.be/

6.1.5 SSH and TLS Resilient Connections

Koponen et al present a session layer mobility solution for Secure Shell (SSH) Transport Layer Protocol [50, 51] and TLS [12] called resilient connections [25]. Their work extends current SSH and TLS by adding a synchronization process for resilient connections. It uses OpenSSH and PureTLS as implementation targets.

The resilient connections are only for the SSH and TLS protocols at the session layer, while HIPL Library provides more general APIs for all transport and application layer protocols. Regarding to mobility, the resilient connections and HIPL Library share many common properties, such as re-creating a new transport-layer end points for the handover, buffering application data, combining control plane and data plane in a single transport layer end point and providing mechanisms for buffer migration.

6.1.6 TESLA

The Transparent Extensible Session-Layer Framework (TESLA) [44] is an all-in-one solution that includes support for mobility, encryption, application-layer routing and connection multiplexing. TESLA adds a new flow-based abstraction on top of the socket, and provides higher level APIs for applications. The handling of a flow in TESLA involves a chain of handlers. This chain enhances the flexibility of the protocol and new functions can be adopted separately by including a new handler.

Similar to HIPL Library, the mobility support in TESLA is also achieved by using buffers to preserve the data. TESLA APIs are event driven. In contrast, HIPL Library offers a Sockets API oriented interface and event-driven support is expected to be offered by another library written on top of HIPL Library.

6.1.7 Other Transport Layer Mobility Solutions

In a survey paper on transport layer mobility solutions [4], the authors have analyzed several approaches, including MSOCKS [29] and Migrate [47]. MSOCKS is a proxy-based solution, and it requires changes to TCP in the networking stack. Migrate TCP also demands changes to the networking stack. It uses DNS to record the location and to facilitate handovers. The survey also presents several other approaches and compares them according to various criteria. A comparison between layer two and layer three mobility based on inter-operatable socket and Mobile IPv6 is presented by Kimura[22].

6.2 Name-based Sockets

The Name-Based Sockets API [18] improves the current Sockets API by leveraging DNS-based, symbolic names as host identifiers. we have adopted a similar approach for the API of HIPL Library as it inputs FQDN names (in addition to HITs and IP addresses) and thus can hide all the details related to addressing. The Name-Based Sockets API introduces a new name exchange protocol and a new address family (AF_NAME). A benefit of the name-based solution is that IP address migration is smoother without session interruption because applications relies on more stable names instead of IP addresses that changes due to relocation [5].

Chapter 7

Future Work

Some unimplemented features of HIPv2 in the HIPL project and some ideas for improving HIPL Library are described in this chapter.

7.1 HIPv2

For HIPv2 support in HIPL, the dual-version and cryptographic agility framework have laid a solid foundation for implementing the remaining, which includes mainly four features.

First, HIPv2 expands the static Key Derivation Function (KDF) mechanism designed for HIPv1. It is decided by the Diffie-Hellman negotiation result in HIPv2. Second, the new OGA definition was not published in the ORCHID specification during the time of thesis writing. Once the new specification is finalized, the support of the OGA bits for HITs in version 2 can be developed for HIPL implementation. Third, the parameters SEQ and ACK are mandatory for an UPDATE message in HIPv2, while they are optional for HIPv1. The UPDATE message handling for version 2 should be upgraded accordingly. Last but not least, instead of SHA-1, HIPv2 recommends the more future-proof SHA-256 hash algorithm for enhanced security. This algorithm should be adopted into HIPL implementation.

7.2 HIPv2.5

The library prototype demonstrates the feasibility of a library-based solution. As the current library is a prototype, it has room for improvement in different aspects.

7.2.1 Protection of the Data Plane

The data plane in the prototype is operated in clear text. Since the BEX establishes a master key for the HIP association, the library can adopt a symmetric key algorithm to protect the data plane. One viable option is the libSRTP[1] library. This library is an implementation of the Secure Real-time Transport Protocol (SRTP) [15, 27] which is also adopted by Real-Time Communication in WEB-browsers (RTCWEB) as part of its security solution [3, 8]. Another alternative is to employ the userspace IPsec implementation available in the HIPL project.

The data plane security can also be offered by the lower-layer HIP if we consider the interoperability between our library and normal HIP solutions. When the library detects a HIT being used, it could resort to security as offered by the lower-layer HIP solution. This way, the library can support cross-layer optimization to minimize the processing costs of redundant security.

7.2.2 Integrate HIPL Library to Existing Applications

Integrating the library to existing applications is the best way to demonstrate the advantages of the library. Many applications have been discussed as candidates for integration. The HIPL team at the Aalto university decided to choose the rtorrent[2] that is a bit-torrent client based on a terminal interface. The integration experiences will also provide feedback for further HIPL Library development.

7.2.3 Adaptive Handover Timer

As described in section 3.2.5.2, the library currently uses a static timer to trigger the handover process. A better solution is to dynamically detect the connectivity status according to certain properties such as the RTT of the application data and then to create an adaptive timer, which can provide a more accurate handover threshold for different network situations.

7.2.4 Inverse of Control (IoC)

In section 3.2, Inverse of Control (IoC) is proposed as a supplementary library on top of HIPL Library which provides event-driven APIs. An IoC-based solution provides a simpler higher level interface than the Sockets API. It can

[1] http://srtp.sourceforge.net/srtp.html
[2] http://libtorrent.rakshasa.no/

also overcome some obstacles in the current library related to connectivity change detection because event-driven style applications delegate right of the control to the library, thus the library can schedule connectivity status detection in a more convenient way.

7.2.5 Concurrent execution on the Library API

It is common for network applications to utilize the multi-process or multi-thread strategy for the management of multiple transport layer sessions. However, as a prototype, the library is not yet ready for a multi-process or multi-thread environment. The library is built on top of the HIPL daemon and reuses its code to expedites the development of the prototype. As the HIPL daemon only requires a single daemon process, it is developed without the consideration for multi-process or multi-thread. Inheriting from the daemon implementation, the library also lacks concurrent protection. However, excluding the code from the daemon implementation, the code of the library itself are mostly ready for concurrent environment, except the re-entrance protection for the HSOCK hash table. If a proper concurrent mechanism is applied to the HIPL daemon and the HSOCK hash table in the future, the development of application can have more alternative ways for the integration of the library.

7.2.6 HIP RVS Incompatibility

The HIP RVS is an extension [26] to solve the mobility problem for two simultaneously moving HIP hosts. This extension introduces an rendezvous server to record HIT-to-IP mappings of a host and to relay HIP messages. The design of RVS conflicts with HIPL Library in the case of TCP. First, it does not support TCP. Second, another problem is that the responder connects directly back to the initiator. For TCP, this means that the initiator needs to have a listening socket to wait for another new connection from the responder, which is not supported by the library. If the library runs in UDP mode, the RVS support can be developed because UDP does not require a new socket for a new connection.

7.2.7 Built-in NAT Support

The library header in the HIPL Library conforms to the standard HIP NAT solutions which are described in Section 2.4. The library can provide a built-in NAT solution which utilizes the same specifications.

7.2.8 TCP User Timeout Option

The TCP user timeout option [14] is a specification to customize the timeout limit for unacknowledged TCP data. The option can be used to prolong the lifetime of a TCP session in the presence of unreliable, lossy networks. It can also expedite the handling time of unacknowledged data by reducing the its value. The timeout limit is negotiable and the option has to be supported by both end-points. If the timeout is exceeded in a end-point, it will close the connection. The user timeout option is available in Linux from kernel 2.6.37[3].

7.2.9 Library API Improvement

As mentioned in section 5.1, for the current library prototype, a wrapper version of the *select()* function is missing and the *hipl_connect()* lacks support for non-blocking operations. This limitation forces applications to directly operate on socket descriptors, which complicates the development of the application. Therefore, the library should provide dedicate APIs for them and hide implementation details from applications.

In addition, the library currently only provides two APIs for reading and sending data which are *hipl_sendto()* and *hipl_recvfrom()*. To meet the requirements of different use scenarios, the library should provide more alternative APIs. For instance, it should offer *hipl_send()* and *hipl_recv()* for TCP mode, and *hipl_sendmsg()* and *hipl_recvmsg()* for more general usage. The latter two APIs can be built as the foundation of other data operation APIs for better modularization.

7.2.10 Opportunistic Mode and Fallback Mode

Currently, the hipnetcat software can handle multiple server identities as parameters and it tries all HIT to locator combinations of them to find a suitable mapping to reach the server. However, the software cannot handle the case that only locators are provided because the library does not support opportunistic mode. If the library support this mode, the hipnetcat software can be smarter and more flexible for input parameters.

In addition, as described in section 3.2.2, the library should support the fallback mode, which allows it to support legacy cases and to improve its deployability.

[3]Kernel 2.6.37 changelogs: http://kernelnewbies.org/Linux_2_6_37

7.2.11 IPv6 Support

Currently, the library prototype only supports IPv4 and temporarily disables the access of IPv6-related functionality. When an application calls *hipl_socket()* with socket family AF_INET6, the library returns error to prevent the operation. The issue is caused by the code reused by our prototype from the HIPL project. The IPv6 and IPv4 packet processing are slight different in the HIPL, thereby causing incompatibility in the library implementation. If the issue is fixed in the HIPL project in the future, the library should be able to support IPv6.

Chapter 8

Conclusion

HIP is a feature-rich protocol that supports security, mobility, IPv4 and IPv6 inter-operation, NAT traversal and other extensions. HIP requires a shim layer that is logically located between the transport and network layers in the networking stack. This thesis presents our contributions on two different areas of HIP, HIPv2 and HIPv2.5.

HIPv2 is the next generation of HIP. It is under the standardization process and mainly focuses on security improvements. The new version introduces cryptographic agility, eliminates vulnerable security algorithms such as SHA-1, and adds stronger algorithms such as Ellipse Curve DSA and Ellipse Curve Diffie-Hellman. Our efforts on HIPv2 can be divided into two parts.

First, we analyze the transition issues from version one to version two and propose a dual-version solution. We identified a transition issue in the way HIPv2 achieves cryptographic agility. Based on our feedback, the HIP working group in the IETF agreed that the IPv6 prefix needs to be changed in HIPv2. To support also version agility, we have extended HIP for Linux implementation and our contribution has been adopted into the official source code.

Then, we validate and implement the new cryptographic agility mechanism for the HIPL project to support the standardization process of HIPv2. The agility mechanism also includes a re-confirmation mechanism to prevent down-grade attacks where a man-in-the-middle attacker can enforce weaker key algorithms.

On the other hand, our novel research proposal for HIPv2.5 is not yet included in the standardization process. HIPv2.5 is a library-based solution that moves HIP on top of the transport layer and drastically differs from current network-layer solutions. Compared to the network-based solutions, a library-based solution operating at the application layer has a number of

advantages: first, applications gain a direct control of HIP and can customize according to their requirements; second, the stand-alone library does not need any additional HIP software to be installed and does not require administrative privileges.

The library prototype supports several features. It can demultiplex control and data plane within a single transport layer session. It buffers data during handover to prevent TCP data loss and detects handovers automatically based on undelivered data.

We demonstrate the library using two applications, the first one was adopted to the official HIP for Linux software and its purpose is to facilitate system-level testing internally for the developers. The second one is a simple client-server application that aids developers to integrate the library into their own projects.

We hope the HIPv2.5 concept facilitates HIP deployment as it is easier to adopt libraries into standard Linux distributions. The librarized HIP offers a stand-alone platform to develop and customize HIP-aware applications. For instance, the library can be used to realize public-keys based authentication, authorization and accounting for users and services. It also supports application-layer mobility and multihoming to be used for disruption-free multimedia streaming. NAT traversal is implementable and HIP offers a convenient namespace to identify hosts securely in private address realms. In the future, we believe the library could offer these features in unified and bug-free implementation instead of application-specific hacks.

Bibliography

[1] 3rd, D. E. RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS). RFC 3110, Internet Engineering Task Force, May 2001.

[2] 3rd, D. E., and Jones, P. US Secure Hash Algorithm 1 (SHA1). RFC 3174, Internet Engineering Task Force, Sept. 2001.

[3] Alvestrand, H. Overview: Real Time Protocols for Brower-based Applications draft-ietf-rtcweb-overview-04. Internet-draft, IETF Network Working Group, June 2012.

[4] Atiquzzaman, M., and Reaz, A. Survey and classification of transport layer mobility management schemes. In *Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on* (2005), vol. 4, IEEE, pp. 2109–2115.

[5] B. Carpenter, R. A., and Flinck, H. Host Identity Protocol. RFC 5887, Internet Engineering Task Force, May 2010.

[6] Bellovin, S., and Rescorla, E. Deploying a new hash algorithm, 2005.

[7] Bishaj, B. Efficient Leap of Faith Security with Host Identity Protocol.

[8] C. Holmberg, S. H., and Eriksson, G. Web Real-Time Communication Use-cases and Requirements draft-ietf-rtcweb-use-cases-and-requirements-09. Internet-draft, IETF RTCWEB Working Group, June 2012.

[9] Camarillo, G., and Melen, J. Host Identity Protocol (HIP) Immediate Carriage and Conveyance of Upper-Layer Protocol Signaling (HICCUPS). RFC 6078, Internet Engineering Task Force, Jan. 2011.

[10] D. Borman, S. D., and Hinden, R. IPv6 Jumbograms. RFC 2675, Internet Engineering Task Force, Aug. 1999.

[11] Deering, S., and Hinden, R. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Internet Engineering Task Force, Dec. 1998.

[12] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force Network Working Group, Jan. 1999.

[13] Eastlake, D. DSA KEYs and SIGs in the Domain Name System (DNS). RFC 2536, Internet Engineering Task Force, Mar. 1999.

[14] Eggert, L., and Gont, F. TCP User Timeout Option. RFC 5482, Internet Engineering Task Force, Mar. 2009.

[15] H. Tschofenig, M. S., and Muenz, F. Using SRTP transport format with HIP. Internet-draft, HIP Research Group (HIPRG), Oct. 2006.

[16] Heer, T., and Varjonen, S. Host Identity Protocol Certificates. RFC 6253, Internet Engineering Task Force, May 2011.

[17] J. Rosenberg, R. Mahy, P. M., and Wing, D. Session Traversal Utilities for NAT (STUN). RFC 5389, Internet Engineering Task Force, Oct. 2008.

[18] J. Ubillos, M. Xu, Z. M., and Vogt, C. Name-Based Sockets Architecture. Internet-draft, Internet Engineering Task Force, Sept. 2010.

[19] Just, T. Adaptable and Flexible Protocols and Protocol Implementations. Master's thesis, RWTH Aachen University, Germany, 2010.

[20] Kent, S. IP Encapsulating Security Payload (ESP). RFC 4303, Internet Engineering Task Force, Dec. 2005.

[21] Keranen, A., and Melen, J. Ative NAT Traversal Mode for the Host Identity Protocol draft-ietf-hip-native-nat-traversal-03. Tech. rep.

[22] Kimura, B., and dos Santos Moreira, E. Mobility at the application level. In *INFOCOM Workshops 2009, IEEE* (2009), IEEE, pp. 1–2.

[23] Komu, M. Application programming interfaces for the host identity protocol. *Master's thesis, Helsinki University of Technology* (2004).

[24] Komu, M., and Henderson, T. Basic Socket Interface Extensions for the Host Identity Protocol (HIP). RFC 6317, Internet Engineering Task Force, July 2011.

[25] Koponen, T., Eronen, P., Särelä, M., et al. Resilient Connections for SSH and TLS. In *Proceedings of the Annual Technical Conference on USENIX* (2006), vol. 6.

[26] Laganier, J., and Eggert, L. Host Identity Protocol (HIP) Rendezvous Extension. RFC 5204, Internet Engineering Task Force, Apr. 2008.

[27] M. Baugher, D. McGrew, M. N. E. C., and Norrman, K. The Secure Real-time Transport Protocol (SRTP). RFC 3711, Internet Engineering Task Force, Mar. 2004.

[28] M. Komu, T. Henderson, H. T. J. M., and A. Keranen, E. Basic Host Identity Protocol (HIP) Extensions for Traversal of Network Address Translators. RFC 5770, Internet Engineering Task Force, Apr. 2010.

[29] Maltz, D., and Bhagwat, P. MSOCKS: An architecture for transport layer mobility. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (1998), vol. 3, IEEE, pp. 1037–1045.

[30] Moskowitz, R., and Nikander, P. Host Identity Protocol (HIP) Architecture. RFC 4423, Internet Engineering Task Force, May 2006.

[31] Nikander, P., Ylitalo, J., and Wall, J. Integrating security, mobility, and multi-homing in a HIP way. In *Network and Distributed Systems Security Symp. NDSS03* (2003).

[32] P. Jokela, R. M., and Nikander, P. Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP). RFC 5202, Internet Engineering Task Force, Apr. 2008.

[33] P. Nikander, T. Henderson, E. C. V., and Arkko, J. End-Host Mobility and Multihoming with the Host Identity Protocol. RFC 5206, Internet Engineering Task Force, Apr. 2008.

[34] P. Nikander, J. L., and Dupont, F. An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID). RFC 4843, Internet Engineering Task Force, Apr. 2007.

[35] Paine, R. *Beyond HIP: The End to Hacking As We Know It.* BookSurge Publishing, 2009.

[36] Pathak, A., Komu, M., and Gurtov, A. Host Identity Protocol for Linux. In *Linux Journal* (Nov. 2009).

[37] Perkins, C. Mobile networking through mobile IP. *Internet Computing, IEEE 2*, 1 (1998), 58–69.

[38] R. Arends, R. Austein, M. L. D. M., and Rose, S. Resource Records for the DNS Security Extensions. RFC 4034, Internet Engineering Task Force, Mar. 2005.

[39] R. Mahy, P. M., and Rosenberg, J. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, Internet Engineering Task Force, Apr. 2010.

[40] R. Moskowitz, P. Nikander, P. J., and Henderson, T. Host Identity Protocol. RFC 5201, Internet Engineering Task Force, Apr. 2008.

[41] R. Moskowitz, T. Heer, P. J., and Henderson, T. Host Identity Protocol Version 2 (HIPv2). RFC-bis-08 5201, Internet Engineering Task Force, Mar. 2012.

[42] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security Version 1.2. RFC 6347, Internet Engineering Task Force, Jan. 2012.

[43] Rosenberg, J. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, Internet Engineering Task Force, Apr. 2010.

[44] Salz, J. *TESLA: A transparent, extensible session-layer framework for end-to-end network services.* PhD thesis, Citeseer, 2002.

[45] Scharf, M., and Ford, A. MPTCP Application Interface Considerations. Internet-draft, Internet Engineering Task Force, Feb. 2012.

[46] Seggelmann, R., Tüxen, M., and Rathgeb, E. P. DTLS mobility. In *Proceedings of the 13th international conference on Distributed Computing and Networking* (Berlin, Heidelberg, 2012), ICDCN'12, Springer-Verlag, pp. 443–457.

[47] Snoeren, A., and Balakrishnan, H. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking* (2000), ACM, pp. 155–166.

[48] Tapio Levä, Miika Komu, A. K., and Luukkainen, S. Adoption of General-purpose Communication Protocols: the Case of Host Identity Protocol. Submitted to Special Issue on Future Internet Testbeds, Elsevier COMCOM Journal, 2012.

[49] Wang, X., Yin, Y., and Yu, H. Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005* (2005), Springer, pp. 17–36.

[50] Ylonen, T., and C. Lonvick, E. The Secure Shell (SSH) Protocol Architecture. RFC 4251, Internet Engineering Task Force Network Working Group, Jan. 2006.

[51] Ylonen, T., and C. Lonvick, E. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, Internet Engineering Task Force Network Working Group, Jan. 2006.

[52] Zandy, V., and Miller, B. Reliable network connections. In *Proceedings of the 8th annual International Conference on Mobile Computing and Networking* (2002), ACM, pp. 95–106.

Appendix A

HIPL Library API

```
typedef uint16_t hipl_sock_id;
```

This *hipl_sock_id* type is used to identify a HIPL socket, similarly as with socket file descriptors. For the locator parameter, the API employs a text format instead of the binary-format sockaddr structure.

A.1 Initialize the Library

```
void hipl_lib_init(void)
```

This function initializes library related data structures. It must be called before using any other function calls of the API.

A.2 Creating a Socket

```
hipl_sock_id hipl_socket(int domain, int type, int protocol)
```

- domain: IPv4 or IPv6
- type: SOCK_STREAM or SOCK_DGRAM
- protocol: TCP or UDP

A.3 Binding a Socket

```
int hipl_bind(hipl_sock_id hipl_socket,
              char *address,
              uint16_t port)
```

A.4 Setting Up a Listening Socket

```
int hipl_listen(hipl_sock_id hipl_socket, int backlog)
```

A.5 Accepting a New Connection

```
hipl_sock_id hipl_accept(hipl_sock_id hipl_socket)
```

The call blocks by default until the TCP three-way handshake is completed.

A.6 Connecting to a Remote Server

```
int hipl_connect(hipl_sock_id hipl_socket,
                 char *address,
                 uint16_t port)
```

The call blocks by default until the connection is established.

A.7 Sending Data

```
int hipl_sendto(hipl_sock_id hipl_socket,
                const void *data,
                size_t size,
                int flag,
                char *address,
                uint16_t port)
```

If the BEX is not established, the call triggers the BEX before sending out the data. The call blocks and not support the *select()* function by default.

A.8 Receiving Data

```
int hipl_recvfrom(hipl_sock_id hipl_socket,
                  void *buffer,
                  size_t buffer_size,
                  int flag,
                  char *address,
                  uint16_t port)
```

If the BEX is not established, the call waits for the BEX before receiving data. The call blocks and not support the *select()* function by default.

A.9 Closing a Socket

```
int hipl_close(hipl_sock_id hipl_socket)
```

This call initiates shutdown procedures for both the control and data plane to terminate them gracefully.

Appendix B

Hipnetcat

This chapter presents the synopsis of hipnetcat and description of each parameter. It also includes two examples to demonstrate the usage of the software.

B.1 Usage

The usage of hipnetcat is as follows:

```
hipnetcat [-hlt] [-p source_port] [-s source_ip_address]
          [-d dest_port] [server_identifier[s]]
```

- *-h*: Print usage of the hipnetcat
- *-l*: Switch the hipnetcat to server mode and wait for incoming connections. In this mode, the *-d* parameter and the *peer_identifier* parameter are disabled. If this option is not specified, the hipnetcat acts as a client.
- *-t*: Turn on TCP transport. If this option is not specified, hipnetcat runs on top of the UDP.
- *-p*: Hipnetcat should use the source port number as specified by the *source_port* parameter instead of an ephemeral port number.
- *-s*: Hipnetcat should use the source IP address as specified by the *source_ip_address* parameter instead of a wildcard.
- *-d*: Hipnetcat running as a client should connect to the remote port as specified by the *dest_port* parameter. If this option is not given, the hipnetcat uses port 10500 by default.

- *server_identifier[s]*: The IP, domain name and HIT are all valid identifiers for a server. Users can provide multiple identifiers for hipnetcat and it will attempt all combination to reach the server. Hipnetcat associates regular IP addresses as locators when a HIT is given.

B.2 Example Runs

We provide two example runs to demonstrate the usage of the hipnetcat, one for TCP and one for UDP.

In the first example, the hipnetcat client is running on top of TCP and connecting to the hipnetcat server. The server is listening the address 127.0.0.1 and port 22300, and the client is connecting from 127.0.0.1 port 22345 with HIT 2001:1c:809e:244a:c33:78fb:45e3:d132.

- At the server side:

```
hipnetcat -l -t -s 127.0.0.1 -p 22300
```

- At the client side:

```
hipnetcat -t -s 127.0.0.1 -p 22345 -d 22300 \
          127.0.0.1 2001:1c:809e:244a:c33:78fb:45e3:d132
```

For the second example, both sides are running on the UDP mode and the server is listening on port 22300 on wildcard IP address. The client side uses an ephemeral local port to connect to the server.

- At the server side:

```
hipnetcat -l -p 22300
```

- At the client side:

```
hipnetcat -s 127.0.0.1 -d 22300 \
          127.0.0.1 2001:1c:809e:244a:c33:78fb:45e3:d132
```