HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Blerta Bishaj

Efficient Leap of Faith Security with Host Identity Protocol

Master's Thesis
Espoo, June 24, 2008

Supervisors: D. Sasu Tarkoma, Helsinki University of Technology
Ph.D. Peter Sjödin, The Royal Institute of Technology

Instructor: Miika Komu, M.Sc.(Tech), Helsinki University of Technology

HELSINKI UNIVERSITY
OF TECHNOLOGY
Faculty of Information and Natural
Sciences
Degree Programme of Security and
Mobile Computing

ABSTRACT OF THE
MASTER'S THESIS

Author	Blerta Bishaj	**Date** June 24, 2008 **Pages** 59

Title of thesis	Efficient Leap of Faith Security with Host Identity Protocol	
Professorship	Data Communications Software	**Professorship Code** T-110
Supervisors	D. Sasu Tarkoma Ph.D. Peter Sjödin	
Instructor	M.Sc. Miika Komu	

Host Identity Protocol (HIP) protocol supports secure communication, end-host mobility, as well as end-host multi-homing. Despite the benefits, communication of HIP hosts with non-HIP hosts is deficient in terms of latency. If HIP is to be widely deployed, it needs to coexist smoothly with the normal TCP/IP stack. A sudden shift of all hosts to HIP, a flag day, is simply unrealistic. Users and organizations deploy HIP only if it is fully backwards compatible. Therefore, HIP hosts need to be able to communicate with both HIP as well as non-HIP peers.

HIP-based communication is different from normal TCP/IP communication. End-hosts perform a base exchange and optionally encrypt transport layer data. In order for a HIP host to communicate with an unknown peer, it has to detect whether the peer is HIP-capable. Currently, there is no deployed infrastructure to support retrieval of such information, such as DNS or OpenDHT. Similarly to many other protocols, HIP discovers support at peer hosts based on timeouts which increase latency of communication. User experience degrades as a result.

With the aim of not diminishing user experience, this thesis presents a design and implementation to detect HIP support at peer without timeouts. Our solution is based on Leap of Faith (LoF) security. Other design alternatives are also discussed, and the performance of the solution is measured and analyzed.

Keywords: Host Identity Protocol, Transport Layer, TCP, Leap of Faith Security

Acknowledgements

I wish to thank both my supervisors, Professor (pro tem) Sasu Tarkoma and Associate Professor Peter Sjödin. Many thanks go to my instructor at HIIT, M.Sc. Miika Komu, for giving me the opportunity to work in the InfraHIP project. Miika has been particularly patient and has helped me immensely throughout the thesis project. I have discussed with him many ideas and always received valuable feedback.

A special thank you goes for the NordSecMob programme and all the people behind it. Without NordSecMob, this valuable experience would not have been possible. I really appreciate the opportunity you gave me. In particular, I would like to thank Professor Antti Ylä-Jääski, Planning officer Eija Kujanpää and M.Sc. Laura Takkinen for their help and guidance during the studies.

I cannot thank my beloved family enough for their love and support, especially my mother, who deserves a lot of credit. As do my sister and brother, Monda, and all my close friends. This Master's Thesis is dedicated to my late father, with gratitude and longing.

Helsinki, June 24, 2008

Blerta Bishaj

Contents

Abbreviations

DSA Digital Signature Algorithm

DNS Domain Name System

DoS Denial of Service

ESP Encapsulating Security Payload

HIP Host Identity Protocol

HI Host Identifier

HIPL HIP for Linux

HIT Host Identity Tag

IETF Internet Engineering Task Force

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

IPsec Internet Protocol security

LoF Leap of Faith

LSI Local Scope Identifier

PKI Public Key Infrastructure

RSA Rivest-Shamir-Adleman

SA Security Association

SPI Security Parameter Index

TCP Transport Control Protocol

RR Resource Records

Chapter 1

Introduction

The TCP/IP suite was designed when host characteristics were different from today. As far as security is concerned, there are all sorts of security attacks on the Internet nowadays. This was not the case at first, and was not reflected on TCP/IP because it was originally designed to operate in a relatively trusted environment. Moreover, computers were singly-homed and their location fixed [26] in the early days of Internet, due to lack of WLAN and portable equipment. Hence, it was assumed that IP addresses served both as locators and identifiers for the hosts. TCP connections are bound to source and destination IP addresses. When the IP address of either of the communicating hosts changes, the TCP connection breaks and communication stops.

The HIP protocol has emerged as a solution to meet the new requirements for mobility and security [19, 24]. Essentially, HIP introduces a new layer between the network and transport layers. The HIP layer takes the identity role from the IP address. The separation of the identity and the location of the host facilitates mobility and multi homing [26]. The HIP layer is below the transport layer, hence connections can be bound to the identifiers it provides. As a result, TCP connections are no more bound to IP addresses, allowing the host to change its location, while maintaining transport layer communication. Furthermore, HIP is above the network layer so that the connection can be handled dynamically through any IP address, providing multi homing. The feature that distinguishes HIP is that it embeds security into the stack, allowing layers higher than the HIP layer to operate securely. HIP uses the public key of a cryptographic public/private key pair as the identity of the host. HIP hosts generate the public/private key pairs themselves, and use their cryptographic identities to authenticate to each-other. They negotiate symmetric encryption keys for IPSec using the Diffie-Hellman cryptographic protocol. These keys secure the data communication between them.

There are several ways how HIP end-hosts can authenticate each other and learn the HIs of each other. The Public Key Infrastructure (PKI) is an option, but it has not been widely deployed in the current Internet. Leap of Faith (LoF) is another possible solution, and SSH is a successful example of it. [12] proposes that the LoF approach provides enough security for HIP mobility and shows that it is backwards compatible, because their implementation can fall back on non-HIP communication in the case that the peer does not support HIP.

The fall back in [12] is based on timeouts. When the HIP Initiator has not received a HIP reply during a certain amount of time, the peer is considered not to support HIP, and the Initiator falls back on normal non-HIP communication. However, the problem with this approach is that the fall back time is unacceptable for a normal user with a good quality connection. Therefore, the HIP capability of the peer should be detected faster. Were a normal user able to benefit from the security and mobility HIP provides without making negative trade-offs in user experience, the deployment of HIP would be wider and faster. The efficient coexistence of HIP with the normal TCP/IP stack increases the chances of HIP to be deployed into existing networks.

In this thesis, we explore the detection of HIP capability. Our design is an extension of the opportunistic mode of HIP. The opportunistic mode allows a HIP host to communicate with other HIP hosts whose HITs are unknown because of the missing HIP infrastructure. Our design is based on receiving an explicit negative acknowledgment when the peer does not support HIP, rather than implicit timeouts. The HIP detection with our design offers a much faster fall back on normal non-HIP communication if the peer does not support HIP.

Our design uses a new, unassigned TCP option for the detection of HIP support because most middleboxes support them. The use of a TCP option has the limitation that it does not support other transport protocols, even though HIP is applicable to other transport layer protocols, such as UDP. Our solution is beneficial to applications running on HIP hosts that use TCP to communicate with other hosts, for example, when a user uses a browser to open a web page containing several links to non-HIP sites.

The organization of the thesis is as follows. Chapter 2 introduces topics and information related to this thesis. Following, Chapter 3 goes through several design alternatives and the implementation architecture. Chapter 4 presents the results and analysis of measurements of the solution implementation. Chapter 5 outlines possible future work. Finally, Chapter 6 presents the conclusions of this thesis.

Chapter 2

Background

This chapter provides background information that is required to understand our proposed solution better. First, we take a quick look at the IP and TCP protocols, focusing on the options fields in the IP and TCP header and on the TCP SYN cookies. Then, an overview of the HIP protocol follows, highlighting the aspects that are significant to the goal of the thesis, such as the opportunistic mode of operation. Then, we give an overview of HIP DNS extensions and HIP NAT traversal extensions. Next comes an overview of the HIP firewall as well as a brief introduction of Linux raw sockets and the libipq library.

2.1 Options in the IP header

2.1.1 IPv4 Options

Internet Protocol (IP) [27] handles the transmission of datagrams from host to host over an interconnected system of networks. IP does not provide reliability, flow control or sequencing. An IP datagram contains a header and data. The IP header contains a field named Options. This field encompasses optional additions to the header that are used for control functions, such as timestamps, security, special routing, etc.

The format of an IP header option can be one of two cases. The simpler one consists of a single octet of option type. Figure 2.1 illustrates the End of Option List option, which marks the end of all options, as well as the No Operation option which is occasionally used to mark the boundary between two consecutive options. Both are single octet options.

The other kind of option contains an option-type octet, followed by an option-

Figure 2.1: IPv4 End of Option List and No Operation options

length octet, followed by the option-data octets. The option-length octet indicates the entire length of the three parts of the option. For example, Figure 2.2 displays the Security option. Through this option, hosts can send security, compartmentalization, handling restrictions, and closed user group parameters.

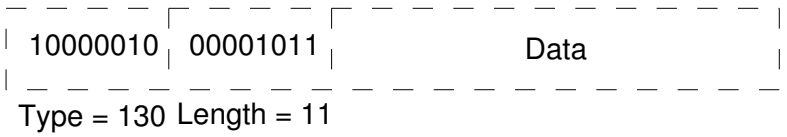

Figure 2.2: IPv4 Security option

2.1.2 IPv6 Options

IPv6 uses a different mechanism for including optional information in a packet [31]. This information is put into separate headers that are placed between the IPv6 header and the upper layer header. There are several types of such extension headers, and an IPv6 packet may contain zero, one or more of them. Each header indicates what header is following. The final one indicates the upper layer protocol. These headers are only examined at the destination, not along the packet path; the Hop-by-Hop Options header is the only exception to this. The IPv6 extension headers carry several fields that are related to the header type, as well as options. The options can be a single octet, or type-length-value options.

Single octet options are illustrated in Figure 2.1 and Figure 2.2, with the IPv4 options of the previous subsection since they are identical to them. Figure 2.3 shows an example of the IPv6 extension headers.

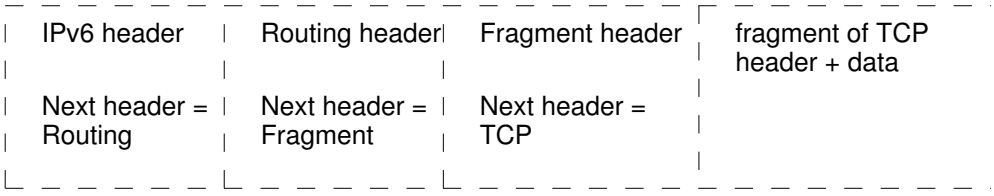

Figure 2.3: IPv6 extension headers

2.2 Options in the TCP header

The TCP protocol [28] is connection-oriented and end-to-end reliable. It is a transport layer protocol and provides reliable communication between pairs of processes in different host computers. The two hosts communicate with each-other exchanging TCP packets, each packet contains a header and data. The TCP header contains the Options field. As the name suggests, this field is optional. If present in the TCP header, the Options field resides at the end of the header. The Options field is a multiple of 8 bits in length, and it is included in the checksum calculation. A number has been assigned for each option kind. However, there are also unassigned numbers, to leave space for future options that might be added to TCP or for experimentations.

There are two categories of TCP options. The first category contains only one octet that indicates the presence of the option kind. There are two special options that belong to the first category. One serves to separate adjacent options, the other one marks the end of all the options in the TCP header. Figure 2.4 illustrates these options in the TCP header. They are the same as the IPV4 options with the same names described in the previous section.

Figure 2.4: TCP End of Option List and No-Operation options

The second category of option contains an octet that indicates the option kind, an octet for the option length, and the octets containing data. The option length accounts for the whole length of the option: option kind, option length, as well as option data. If the end of the final option is not at the end of the current TCP header 32-bit word, then the rest of the 32-bit word is padded. Figure 2.5 shows an option that belongs to the second category.

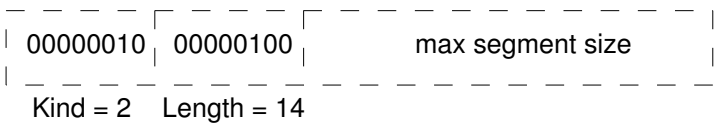

Figure 2.5: TCP Maximum Segment Size option

Communicating peers use options in the TCP header for negotiation of parameters. The peers usually specify them during the handshake. One example of TCP options is the Maximum Segment Size [28], which serves to communicate the maximum segment size that can be received at the hosts that sends this option. The Maximum Segment Size option belongs to the second option category; the option data indicates the allowed maximum segment size. It is demonstrated in [7] that TCP options are widely accepted in the Internet.

2.3 TCP SYN cookies

The TCP protocol is vulnerable to attacks that are based on its specification. One of them is TCP SYN flooding. The attacker attempts to consume the resources at the server, causing DoS eventually. When a TCP client sends a TCP SYN packet, the server makes an entry in a queue for connections in the SYN_RECEIVED state. If the server is attacked with many such packets and does not receive the ACK packets in response to the SYN_ACK packets it has sent, the queue of connections in the SYN_RECEIVED state becomes full eventually. Hence, there is no more space for new connections from legitimate clients [22].

One of the ways to handle this attack is to increase the queue length so that it can accommodate more connections that are in the SYN_RECEIVED state. Another way is to decrease the amount of time that connections in the SYN_-RECEIVED state wait in the queue for the TCP handshake to complete. Still, an attack with a higher number of TCP SYN packets overcomes both these remedies.

Another way to maintain the service to legitimate clients is to use TCP SYN cookies. They are special values given to the initial TCP sequence number. When the queue of connections in the SYN_RECEIVED state becomes full, the server does not save the connections in this state in the queue any more, but only sends out the SYN_ACK packets with the special sequence numbers. When an ACK packets arrives, the server analyzes the packet and the sequence number, extracting the necessary information for creating the TCP connection.

This way, communication with legitimate clients is maintained, and DoS attack is more difficult to mount.

2.4 General Overview of the Host Identity Protocol

In this section, we take a closer view at HIP [9]. The HIP working group at the Internet Engineering Task Force (IETF), as well as a HIP research group at the Internet Research Task Force (IRTF) standardize it. HIP provides security, mobility and end-host multi homing [12] and supports IPSec-based integrity and confidentiality protection for applications. At the time of writing this thesis, there were three public HIP implementations: HIPL [1]; OpenHIP [21]; HIP for inter.net Project [10].

As mentioned earlier in the introduction, HIP introduces the addition of a new layer between the network and transport layers [30]. In HIP, the Host Identifier (HI) assumes the host identification role from the network layer. There is no central authority that assigns HIs to hosts, but they are unique statistically. The connections of the transport layer are bound to host identifiers instead of to IP addresses. As a result, the effects of mobility and multi homing on transport layer connections are reduced because the transport identities remain the same.

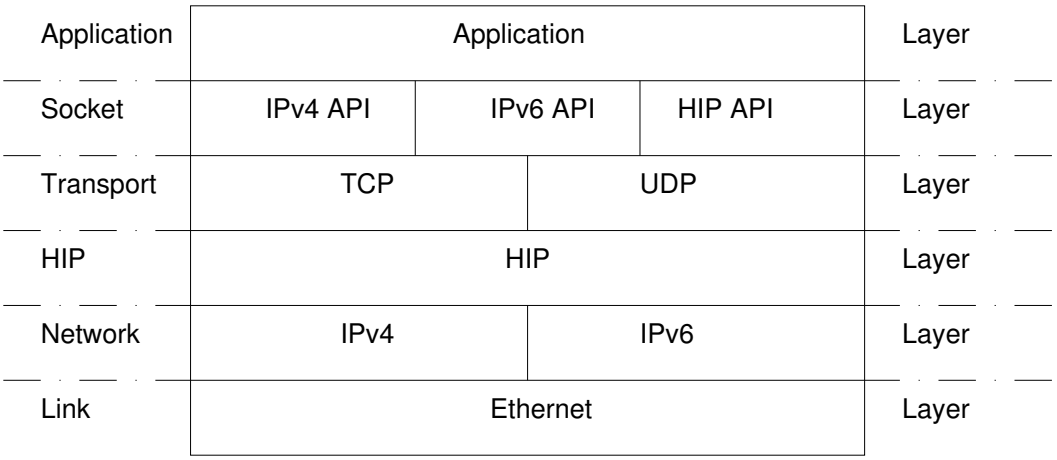

Figure 2.6: HIP layering and naming as presented in [14]

The HI is the public key of a cryptographic public/private key pair [30], and it can be significantly longer than what can fit in the socket structures.

Therefore, shorter representations of HI are used by applications. The Host Identity Tag (HIT) has the length of an IPv6 address, and the Local Scope Identifier (LSI) has the length of an IPv4 address. HIP can work with both IPv4 and IPv6 addresses at the application and the network level. The position HIP takes in the normal stack is depicted in Figure 2.6.

HIP allows applications to exchange encrypted or integrity protected data by using IPSec [23]. Therefore, the communicating peers need to negotiate the secret keys. During the four steps of the base exchange, the two hosts negotiate the IPSec keys and algorithms that are used for protecting the communication. The base exchange is illustrated in Figure 2.7.

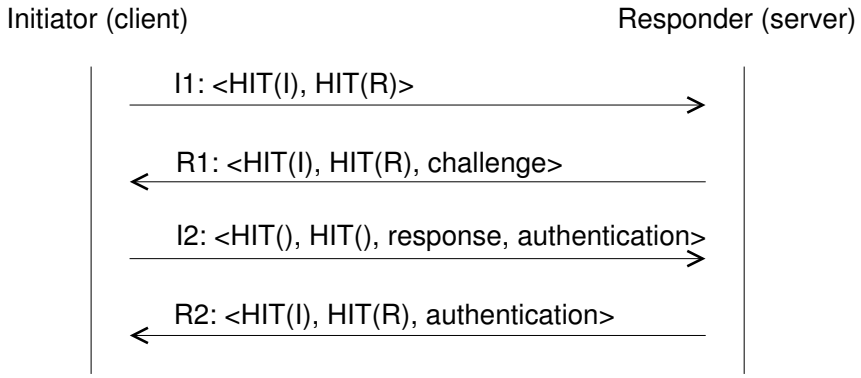

Figure 2.7: HIP base exchange

The packets in the base exchange are not encrypted but are signed with RSA or DSA private keys to allow middlebox inspection [34]. The I1 message is a trigger that indicates that a host wants to communicate using HIP. This message contains the HIT of the peer when it is known. After receiving the I1 packet, the Responder replies with an R1 packet which contains the HITs of the Initiator and of the Responder. The R1 packet also contains a puzzle for the Initiator to solve. The Initiator replies with the I2 packet that contains the solution to the puzzle. The Responder replies with the R2 packet after verifying the solution. The puzzle protects against certain kinds of DoS attacks. During the base exchange, the peers establish a shared secret key for IPSec using Diffie-Hellman.

2.4.1 Opportunistic Mode Implementation Architecture

The opportunistic mode of operation takes place when the Initiator of communication does not know the HIT of the peer and when it operates in envi-

ronments without HIP infrastructure support. In this case, the Initiator puts NULL (all zeros) as the peer HIT in I1 [30]. The Initiator obtains the peer HIT when the Responder sends its HIT in the R1 packet. The opportunistic mode is based on LoF security. In LoF, a host contacts its peer without prior knowledge of the peer identity. The host learns the peer identity in the first contact and stores it to verify the peer identity when recontacting the peer. LoF has been used successfully in protocols such as the Secure Shell (SSH) and the Transport Layer Security (TLS). [12] argues that LoF security is enough for mobility.

There are security issues related to the opportunistic mode. This mode makes HIP vulnerable to replay attacks, since the Responder can reply with R1 packets that contain any HIT. Additionally, the opportunistic base exchange is vulnerable to man-in-the-middle attacks. The reason is that the Initiator does not have the peer HIT. The attacker acting as a man-in-the-middle has to be in the path between the two peers in order to carry out the attack. It replies to the I1 packet of the Initiator with an R1 packet that contains its own identifier. The fall back approach of the opportunistic mode introduces another risk, that of down-negotiation. A man-in-the-middle can drop I1 packets sent by the Initiator and force the communication to fall back on plain TCP/IP. Afterwards, the attacker can eavesdrop on the connection, assuming that it is on the path of communication between the Initiator and the Responder. Alternatively, it can create parallel TCP connections to the Initiator and the Responder.

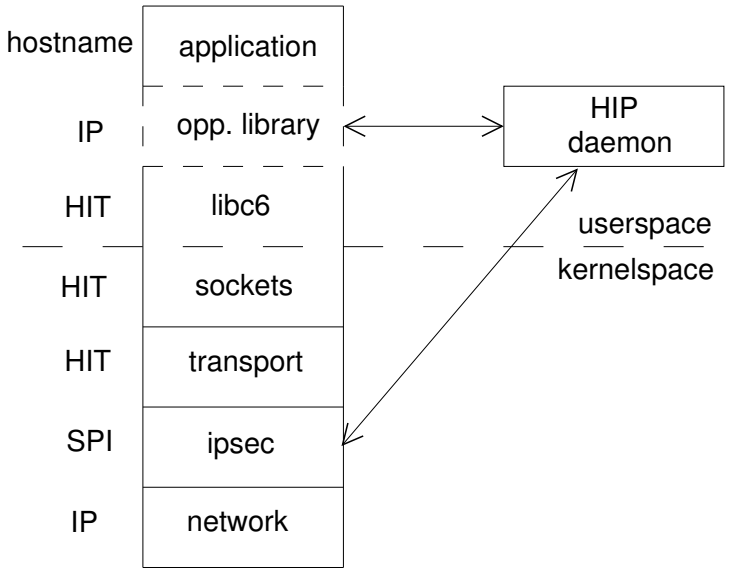

Figure 2.8: Layering and Software Module Organization

In HIPL [1], the opportunistic mode is implemented as a user-space library in Linux. Figure 2.8 depicts the hierarchy of layers in the opportunistic mode and the position of the HIP daemon.

Figure 2.9 visualizes the opportunistic mode of operation from the HIP Initiator point of view using a flow diagram, similarly as in [12]. The steps are marked with numbers.

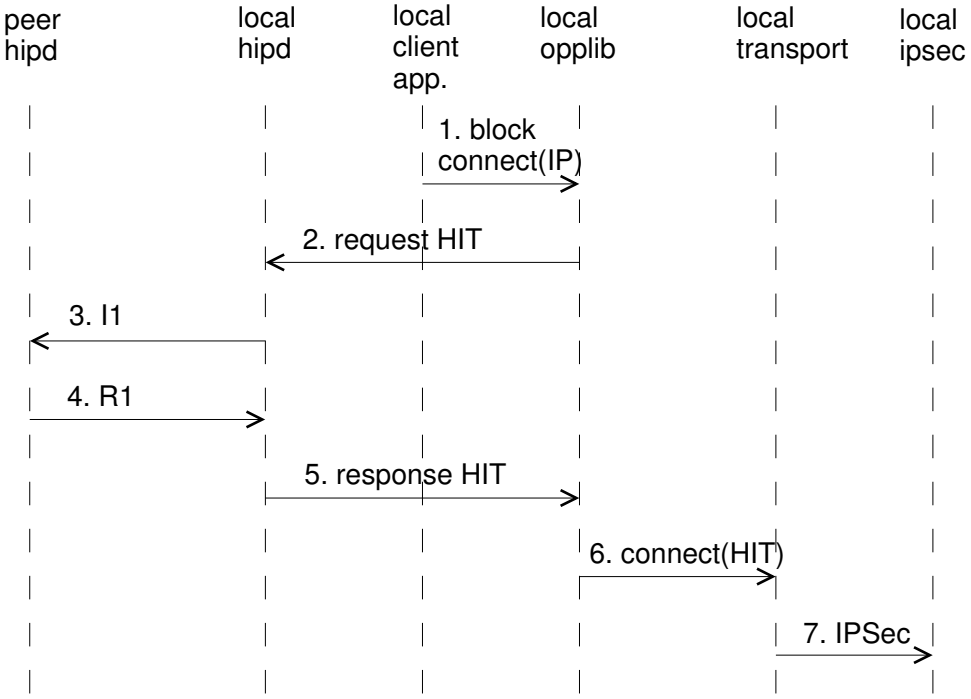

Figure 2.9: Flow Diagram of Opportunistic Base Exchange

In the first step, an application calls a socket function which sends data using IP addresses, such as sendto() or connect(). The opportunistic library intercepts this call, and then, in step 2, queries the HIP daemon for the HIT of the peer. If the local HIPD does not have the peer HIT in store, it initiates an opportunistic base exchange with the peer. In step 3, the HIPD sends the I1 packet of the base exchange to the peer. When the peer responds with the R1 message containing the peer HIT, in step 4, the HIP daemon forwards the peer HIT to the opportunistic library, in step 5. After the local HIP daemon supplies the peer HIT, the library unblocks the application. In the mean time, the HIP daemons at both peers continue with the base exchange. The Responder HIT is now available to the opportunistic library, which creates a socket connection using the peer HIT in step 6. Next, the data proceeds from

the transport layer to the IPSec layer in step 7. Finally, the IPSec layer sends the data to the peer encapsulated within ESP.

2.5 HIP DNS Extensions

Currently, applications that need to communicate with a host translate a domain name into IP addresses. With HIP, the domain name needs to be translated into the HIT of the peer additionally because the transport layer uses HITs for the connections. When the Initiator does not know the HIT of the peer due to the lack of HIP infrastructure, it uses the opportunistic mode. This mode introduces security risks such as the man-in-the-middle attack. To prevent them, the Initiator must first obtain the HIT of the Responder. Therefore, a need arises to translate domain names into HITs. It seems logical to reuse the DNS system for retrieving HITs, but it requires the introduction and implementation of a new Resource Record (RR) to accommodate the information we need to store. The HIP DNS extensions allow recording this information in the DNS.

A DNS RR [18] without any extension has a top level format as illustrated below Figure 2.10.

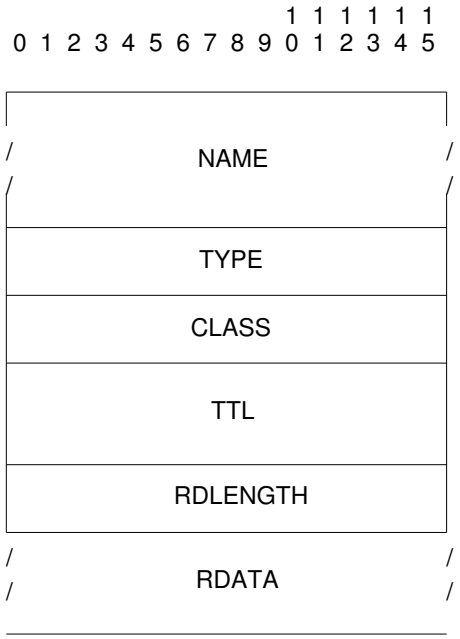

Figure 2.10: DNS RR Format

The DNS RR format contains several fields. NAME is the name of the node for this RR. TYPE consists of two octets indicating the RR type. CLASS also consists of two octets, it indicates the RR class. TTL contains a 32-bit integer that specifies how long to cache an RR before reconsulting the source of the information. A zero value means no caching. RDATA is a field of variable length that describes the source. Its length in octets is specified in the RDLENGTH field.

[25] specifies a new resource record (RR) for the DNS, and its usage with HIP. A HIP host would store its HI, HIT, and the domain names of its Rendezvous Servers (RVS) in the DNS RR. The IP addresses of the host are not kept in the DNS, since they can change frequently due to mobility and DNS propagates changes slowly. Instead, HIP hosts publish the domain name(s) of the RVS in the DNS. Meanwhile, the HIP host updates its set of addresses to the RVS. If the host is not mobile and its IP address/es do not change frequently, it can publish its own data on the HIP DNS instead of the RVS data.

The HIP DNS additional fields are placed in the RDATA field of a HIP RR. Figure 2.11 illustrates the HIP RR storage format.

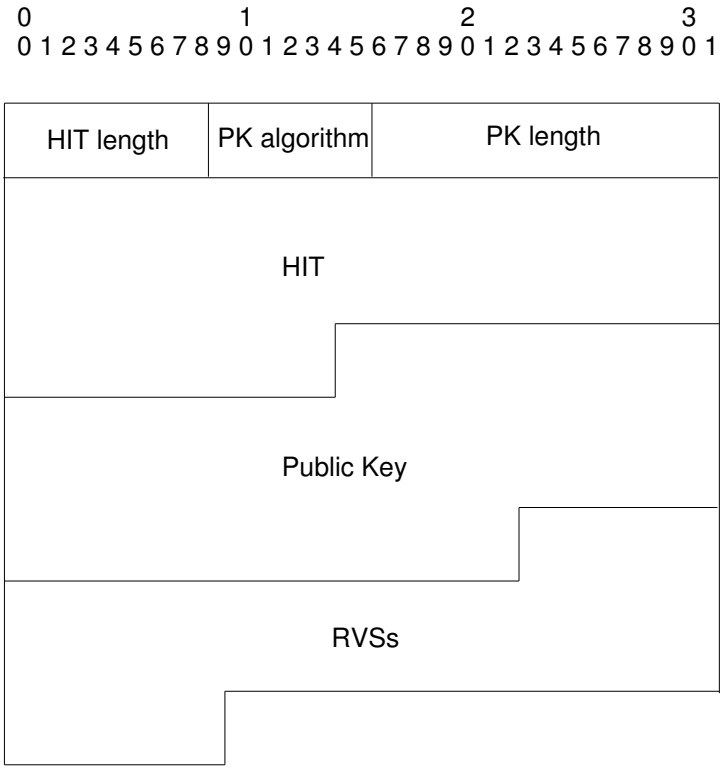

Figure 2.11: HIP RR Storage Format

All fields except the RVSs field are required, the RVSs field is OPTIONAL.

From the security point of view, the DNS is vulnerable to threats described in [8] in the absence of DNSSEC. DNS with HIP extensions is also vulnerable to these threats. It is possible to use HIP to secure communication with DNS servers but it is out of scope of this thesis.

2.6 HIP NAT Traversal Extensions

[15] defines Network Address Translator (NAT) traversal extensions for HIP. The extensions make use of the Interactive Connectivity Establishment (ICE) protocol to discover a path between two HIP hosts that are both behind NATs in the worst case. With these extensions, even legacy applications can communicate with each other behind NATs or firewalls.

A HIP Relay is different from an RVS. The HIP RVSs deal with initial contact and mobility when there is no NAT in the networks. A HIP Relay does the same and solves NAT traversal problems. The HIP Relays can be used in NATted or non-NATted networks. Both RVSs and HIP Relays forward control packets, but the RVS forwards only the I1 packet of the base exchange to the Responder. The remaining control packets, as well as data packets are sent directly between the peers. On the other hand, the HIP Relays relay all HIP control packets because NATs could drop them otherwise.

A HIP host registers with a Traversal Using Relay NAT (TURN) server [11] first and then with the Relay server. A TURN server can be used for relaying data traffic between the peers. The registration with a HIP Relay is shown in Figure 2.12

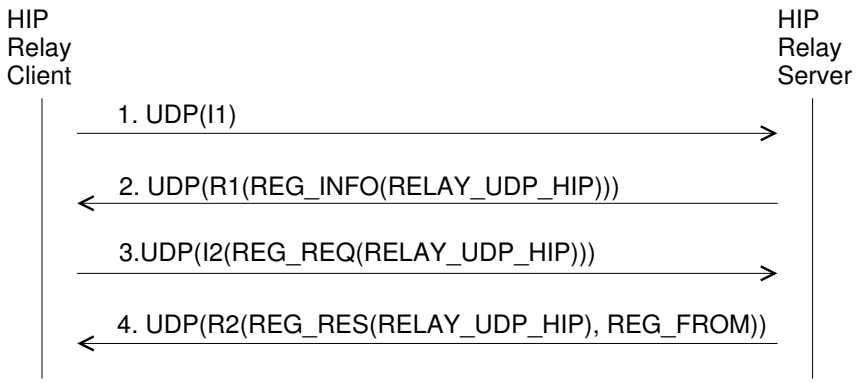

Figure 2.12: Example Registration to a HIP Relay

The registration is a four step process in which the Initiator first initiates the registration by sending an I1 packet. Then, in step 2, the Responder sends an R1 packet with the list of services. The initiator selects the services to register for and sends them in the I2 packet, in step 3. Finally, the Responder acknowledges the registered services with the R2 packet in step 4.

After registration, a HIP Initiator can use the HIP Relay for sending HIP control packets to HIP hosts that have registered with the relay. The relay processes the packets accordingly and passes them over to HIP responders. Instead of four steps, the base exchange would take place in eight steps because the HIP Initiator sends the packets to the relay, which forwards them further to the HIP Responder. The reply from the Responder goes through the relay as well. The processing at the HIP Relay includes changing the source and destination IP addresses and ports, and occasionally adding the RELAY_FROM parameter to packets.

[15] defines a new optional transform parameter type, NAT_TRANSFORM. This parameter is negotiated in the R1 and I2 packets of the base exchange, and it indicates that the host supports the extensions presented in [15]. This parameter can be applied both to the registration with the HIP Relay, and to the base exchange between HIP hosts.

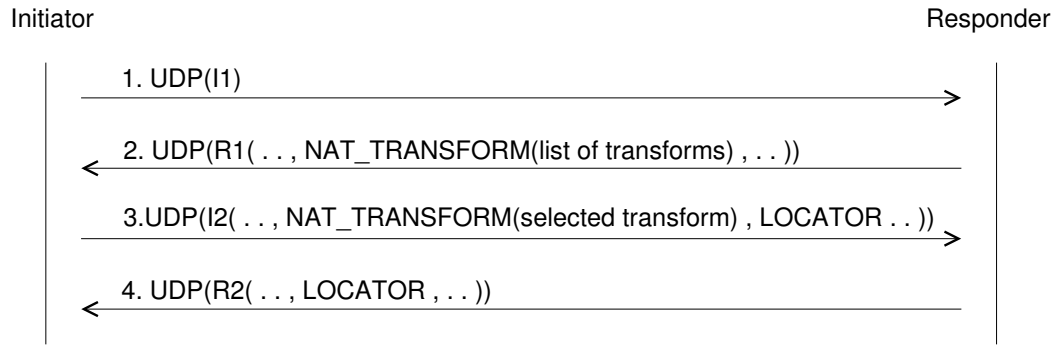

Figure 2.13: Negotiation of NAT Transforms between HIP hosts

Figure 2.13 illustrates the negotiation of the NAT transform type during the base exchange when two hosts have direct communication.

Figure 2.14 shows the types of transformation that are currently supported.

The changes in the base exchange begin in the R1 message in step 2 which contains the list of transforms that the Responder supports inside the NAT_TRANSFORM parameter. The Initiator replies with an I2 packet in step 3 that contains the NAT_TRANSFORM parameter with the transform type the Initiator has chosen from the list the Responder sent.

Transform Type	Purpose
RESERVED	Reserved for future use
ICE–STUN–UDP	UDP encapsulated control and data traffic with ICE–based connectivity tests using STUN messages

Figure 2.14: Locator Transformations

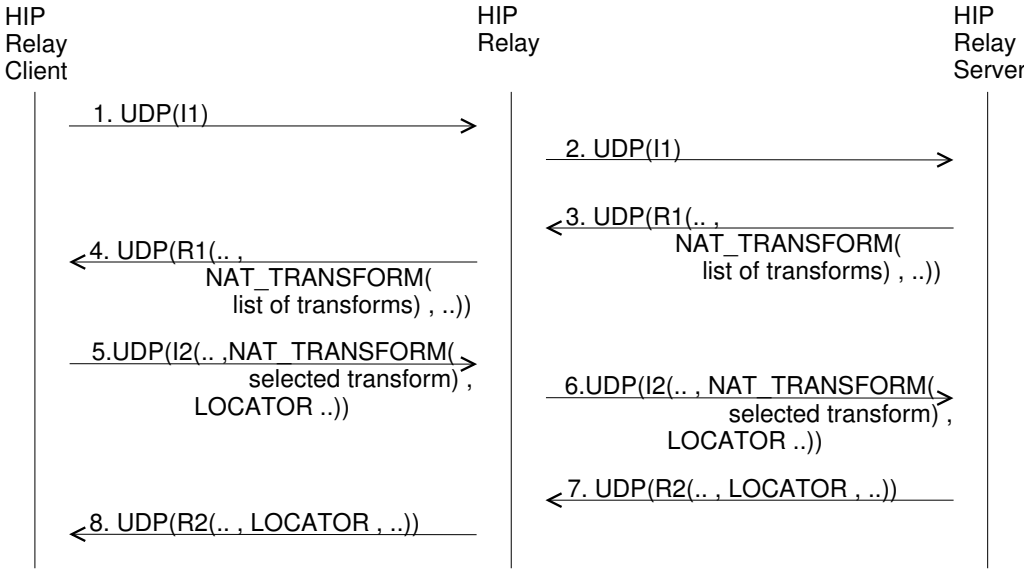

Figure 2.15: Negotiation of NAT Transforms through a HIP Relay

Figure 2.15 illustrates the negotiation of the NAT_TRANSFORM parameter between two hosts when they communicate through a HIP Relay.

After the base exchange, the communicating hosts perform connectivity tests based on ICE to find a direct path for delivery of data traffic. A direct path is preferred to avoid using a TURN server for the data traffic, since the TURN server may become a point of network congestion. Figure 2.13 shows the LOCATOR parameter exchanged in the I2 and R2 packets. This parameter contains a list of all the ICE candidates. ICE candidates are transport addresses that have not been verified yet for reachability using ICE. After the base exchange, the locators are still in an unverified state. The hosts test connectivity using the candidates exchanged during the base exchange. If these tests fail, then a TURN server is used for relaying the data traffic between the

hosts. Apart from the risk of congestion, the TURN server also increases the round-trip delay.

The extensions in [15] do not elaborate on NAT traversal in the HIP opportunistic mode.

2.7 General overview of the HIP firewall

The purpose of this section is to introduce a HIP firewall implementation [34] that is part of the HIPL [1] software bundle. This section also analyzes how the security provided by HIP can contribute to firewalls.

There are three types of firewalls. First, there are firewalls that are not aware of HIP and provide no support for it. Second, there are firewalls that are aware of HIP, but are transparent to end-hosts. Third, there are firewalls aware of HIP that communicate with end-hosts. The initial implementation of the HIP firewall is of the second type. Firewalls are widely used for enforcing security mechanisms. However, current firewalls lack support for HIP traffic [17], [16].

One of the design principles of HIP is to cooperate with middleboxes [30]. HIP provides integrity, confidentiality and data authentication. Still, network entities do have access to plain text HIP association flow identifiers to keep track of HIP connections.

The main function of the HIP firewall implementation is to filter the traffic based on HIs or HITs. A firewall can use HITs conveniently to enforce access control. HIT can serve as the flow identifier for HIP traffic. Middleboxes need to filter ESP data traffic related to the HIP control traffic as well. Therefore, the firewall needs to maintain state of the connection in order to filter all HIP traffic.

HIP uses IPSec ESP to carry its data traffic. Middleboxes can use the IP addresses and the SPIs of peers to identify an ESP flow. Initially, the middlebox can identify the data traffic flow identifiers from the base exchange. The Initiator delivers its initial SPI in the I2 packet and the Responder sends its own initial SPI in the R2 packet. A HIP host can have more than one network address, and the SPI used for a particular connection can change too. Hence, a HIP firewall has to map a HIP association against a varying set of SPIs, as well as against potentially changing IP addresses. However, a HIP host updates its peer about changes in network address and SPIs using udate packets. Middleboxes need to monitor the update packets in order to track the connection properly.

Figure 2.16 illustrates how a firewall keeps state for a HIP connection and

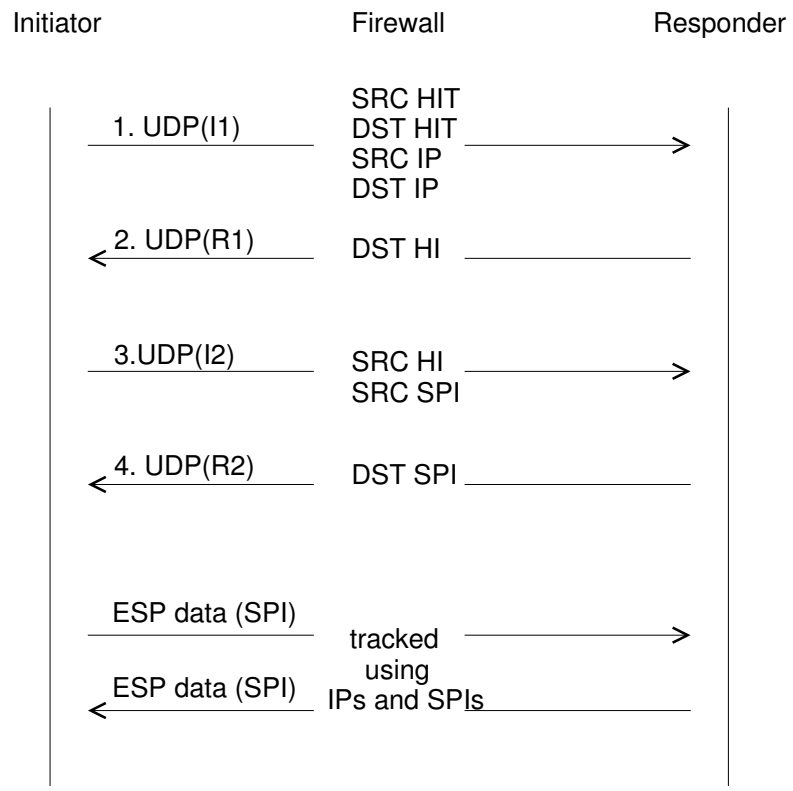

Figure 2.16: Filtering of ESP data traffic at HIP firewall

how it uses it to filter ESP data traffic. This figure does not depict update packets and their processing. During the four steps of the base exchange, the firewall gathers information from the packets. Upon receiving I1 in step 1, the firewall registers the source and destination HITs as well as the source and destination IP addresses. In steps 2 and 3, the firewall registers the destination and source HIs accordingly. In steps 3 and 4, the firewall registers the source and destination SPIs accordingly. When the firewall detects an ESP packet, it extracts the destination IP address from the encapsulating IP packet and the SPI from the ESP header. With this information, the firewall can associate the packet with the HIP connection.

The opportunistic mode of HIP might be troublesome with HIP-enabled firewalls because the firewall cannot determine the destination HIT from the I1 packet of an opportunistic base exchange and it might discard the packet. This problem can be adjusted with firewall rules that support the opportunistic mode.

The opportunistic mode affects stateful filtering as well. The destination HIT

is missing from the I1 packet, but it is used as a flow identifier by firewalls. A firewall that supports the opportunistic mode of HIP can use the destination IP address in the flow identifier, until it intercepts the R1 packet from the Responder and extracts the destination HIT from R1.

HIP signatures are visible to middleboxes. Therefore, middleboxes can authenticate the validity of the control messages by validating the signatures. This distinguishes HIP from other protocols. Traffic authentication means that the firewall can detect invalid packets and keeps state information. This way, middleboxes do not need to go into details of the HIP protocol. Instead, middleboxes only verify the HIP packets.

HIP can encrypt transport layer communication. As a consequence, the HIP firewall cannot intercept transport layer traffic. From the point of view of organizations that want to inspect and filter transport layer traffic, this might be considered a disadvantage. However, the ESP encryption is optional in HIP, and it is possible to use only integrity protection.

The HIP firewall [34] inspects for HIP and ESP traffic. It captures, analyzes, and decides whether to accept or drop incoming or outgoing packets, as well as those being forwarded. The firewall implementation in [34] uses the libipq library to capture packets out of the stack and queue them into userspace [3]. The HIP firewall implements stateless packet filtering, and stateful packet filtering. Stateless filtering, as in all firewalls of such sort, means filtering on the basis of packet properties or the network interfaces.

2.8 Introduction to Linux Raw Sockets

Raw sockets in Linux [2] bypass the kernel stack by allowing packet processing in user space. When creating packets using raw sockets, the programmer is responsible for creating packet headers. Raw sockets can be used for both the IPv4 and IPv6 protocols. As for transport layer protocols, there is no limitation. In fact, raw sockets can be used for new protocols, or protocols that have no user interface, such as ICMP.

With raw sockets, it is possible to create IP packets or trasport layer packets, depending on what socket options are enabled. Specifically, the socket option that specifies whether the programmer creates the IP header or the kernel does is IP_HDRINCL. The following piece of code illustrates the function used to turn this option on and off:

```
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on))
```

In this case, on is an integer that has the value 1. To turn this option off, we call the same function, as illustrated below:

```
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, (char *)&off, sizeof(off))
```

where off is an integer that has the value 0. section A.1 contains raw socket output handling.

In Linux, only processes created by a root user, or having a special capability are able to create and use raw sockets. Additionally, forging the source address works only for IPv4, not IPv6. As for Windows, it lacks support for raw sockets [20]. Windows XP Service Pack 2 (SP2) and Windows Vista restrict the ability to send traffic over raw sockets in several ways. For example, raw sockets cannot send TCP data or spoofed UDP packets. Furthermore, the bind function cannot be called with a raw socket.

2.9 Introduction to the libipq Library

The HIP firewall uses the libipq library [3] for iptables userspace packet queuing. Netfilter [4] supports queuing packets out of the stack into userspace, as well as receiving the packets back into the kernel with a verdict that specifies whether the packet is to be dropped or accepted. Packet modification can also take place before packet re injection back into the kernel.

A kernel module called a queue handler registers with netfilter in order to perform the passing of packets between kernel and userspace. ip_queue is the standard queue handler for IPv4, and ip6_queue is the standard queue handler for IPv6. After loading the ip_queue and/or the ip6_queue modules, iptables rules with the QUEUE target queue IPv4 and/or IPv6 packets for userspace processing, for example:

```
iptables -I INPUT -p 6 -j QUEUE
```

The libipq library is an API for communicating with the ip_queue and ip6_-queue. Following is an example of reading a packet into a buffer with a queue handle:

```
status = ipq_read(handle, buffer, BUFSIZE, 0);
```

The code below illustrates how to obtain a packet message from the data the buffer contains. An application can access the contents of the packet through the ipq_packet_msg_t structure.

```
ipq_packet_msg_t *m = ipq_get_packet(buffer);
```

The following code illustrates further the usage of the library by setting a verdict on a packet. In this case, the verdict is to drop the packet:

```
ipq_set_verdict(handle, m->packet_id, NF_DROP, 0, NULL);
```

section A.2 contains a full example of libipq usage for reading queued packets.

Chapter 3

Implementation Architecture

The problem to solve is to design and implement HIP capability detection to environments without HIP infrastructure. The goal is to make HIP hosts interact at the same time with HIP and non-HIP hosts efficiently. In this chapter, we first present several alternatives we considered, as well as the solution we decided to implement.

3.1 Design Alternatives

This section discusses four different approaches to HIP detection. The design alternatives describe the benefits and drawbacks of each alternative, which helps us justify our selected approach. For each design alternative not chosen as the final solution, we argue why we did not choose it.

The current solution to the HIP detection problem uses timeouts in the opportunistic mode. In the opportunistic mode of HIP, the Initiator sends to the peer an I1 packet that lacks the peer HIT. If there is no reply to I1 within an amount of time, the Initiator assumes that the peer does not support HIP. Timeouts are widely used by many other protocols for detecting protocol support at peer. However, timeouts affect usability since the user has to wait for an amount of time before knowing whether the connection is successful or not.

3.1.1 IP Options

At first, we considered IP options for the detection. The Options field is an optional one in the IPv4 header [27]. The IPv4 specification defines that IP options must be implemented by all IP modules (hosts and gateways). It states that the presence of IP options in datagrams is optional, but the

implementation of receiving datagrams with the Options field is mandatory. However, [32] demonstrates that IPv4 options are not well supported in the Internet. Approximately, half of Internet paths drop IPv4 packets with options according to [32]. Therefore, we decided not to employ IPv4 options as the solution for HIP detection.

We presented a short overview of IPv6 options in subsection 2.1.2. IPv6 options are not widely deployed yet. Therefore, we did not consider them as an eligible solution.

The advantages of detecting HIP through IPv4 or IPv6 are that it would solve the HIP detection problem at the network layer, and we would not need to handle transport layer protocols separately. Even so, if the problem were solved at the network layer, it would mean implementation for both IP versions.

3.1.2 HIP DNS Extensions

HIP DNS RRs are defined in [25]. In this RR, a HIP node stores its HI, HIT, as well as the domain names of its RVSs. The host keeps the RVS updated with its current list of the IP addresses. Instead of the domain names of the RVSs, the RRs can also contain the addresses of the host itself.

This approach is not as vulnerable to man-in-the-middle attacks as the opportunistic mode is. However, it requires that DNS be populated with HITs and that the communication to DNS be secured with, for example, HIP or DNSSEC [29]. Such infrastructure is not yet deployed. Therefore, we did not choose this as a solution.

3.1.3 Establishing Host Identity Protocol Opportunistic Mode with HIT in TCP Option

Another design alternative is [13]. It solves the latency issue by creating a TCP connection to be used in case the peer does not support HIP. This solution is an extension to the HIP opportunistic mode. [13] proposes that instead of the I1 packet, the initiating host sends a TCP packet that contains the local HIT inside a TCP header option. If the peer supports HIP, it extracts the HIT from the TCP options and replies with the R1 message. Otherwise, it replies to the TCP SYN packet with a TCP SYN_ACK packet. This way, the TCP connection is established without additional timeouts.

Unlike IP options, TCP options are generally supported throughout the Internet. [6] presents the results of tests on TCP connections with assigned TCP options (the Timestamp options) or an unassigned TCP option. In both

scenarios, the connection failure rate was only 0.2%.

The first issue with this design alternative is that it is not backward compatible with hosts that use the HIP base exchange. The second issue is that it does not work with NATs. If the Initiator is behind a NAT, the incoming R1 packet will be dropped, because it appears as a connection initiated from outside the NAT.

3.1.4 Optimized TCP Option Approach

The NAT traversal problems of the previous design alternative led us to enhance the opportunistic mode. This design alternative uses a TCP packet with an unassigned option, similarly to the previous alternative. However, the Initiator does not send the TCP packet as part of the base exchange in this case. Before triggering the base exchange, the local host sends a TCP SYN packet with a special option to the peer. If the peer is a HIP host, it replies with a TCP SYN_ACK packet that contains the same option. If the peer is not a HIP host, it replies with a TCP SYN_ACK packet without the option because it does not understand it.

The local host analyzes whether incoming TCP SYN_ACK packets contain the special option or not. If it receives a TCP SYN_ACK packet with the option, it determines that the peer is a HIP host and the local host initiates the HIP base exchange. If the packet does not contain the special option, the local host concludes that the peer is not a HIP host and the local host falls back on non-HIP communication immediately.

This design alternative has the drawback that normal HIP hosts that do not implement this solution appear as non-HIP hosts. Like other non-HIP hosts, legacy HIP hosts do not recognize the new option. These hosts receive first the TCP SYN packet with the option to which the TCP service at the port replies with a TCP SYN_ACK packet lacking the option. This identifies them as non-HIP hosts.

3.1.5 Final Design

We chose the design alternative described in section 3.2 because it was the optimal solution. This design alternative is similar to the one presented in the previous subsection. The local HIP host sends a TCP SYN packet with a special option to the peer. The difference is that, in this case, it also sends the I1 packet to the peer. If the peer is a HIP host, it drops the TCP SYN packet with the option and replies only with an R1 packet. This way, the HIP

base-exchange takes place, and ESP data follow afterwards.

When the peer is not a HIP host, it does not understand the I1 packet and replies only to the TCP SYN packet with a TCP SYN_ACK one excluding the option. The local host filters this packet and concludes that the peer does not support HIP and unblocks the application that initiated the connection. Similarly to the previous design alternative, this approach supports communication with non-HIP hosts without extra timeouts.

The benefit of this solution is that when the peer is a HIP-enabled host, there is no TCP round trip delay for the detection, since the Initiator sends the I1 packet from the beginning. This design alternative has also the advantage that legacy HIP hosts implementing only the base exchange are correctly identified as HIP hosts. In this solution, the Initiator sends the I1 message ahead of the TCP SYN packet with the option, which means that it is receives the I2 packet as a reply earlier than the TCP SYN_ACK lacking the option at the local host. Therefore, HIP hosts that do not support our extension are correctly identified as HIP hosts, in contrast to the previous design alternative.

3.2 Solution Architecture - I1 and TCP Packet with Option Simultaneously

This section describes implementation details of our solution. The implementation of our changes amounts to 634 lines of code. Figure 3.1 depicts the case when the peer is not HIP capable, while Figure 3.2 depicts the case when the peer is HIP capable. In both cases, there is a client application that initiates the communication. It calls connect(IP), which is blocked by the opportunistic library as shown in step 1 of both figures. The opportunistic library requests the matching HIT from the HIP daemon, as seen in step 2 of both pictures. The assumption is that the local host cannot map the IP to the peer HIT due to lack of an infrastructure for retrieval of such information. In step 3 and 4 of both figures, the local HIP daemon sends an I1 packet to the peer, as well as a TCP SYN packet with the special option. It sends the I1 packet first, and then the TCP one.

If the peer is not a HIP host, it will not reply to the I1 message because it does not understand it. Instead, it replies to the TCP SYN_I1 packet with a normal TCP SYN_ACK one lacking the option, as shown in step 5 of Figure 3.1.

The local firewall filters this packet and detects that it does not contain the special option. Thus, it infers that the peer does not support HIP. Therefore, it sends a message to the local HIP daemon requesting it to unblock the ap-

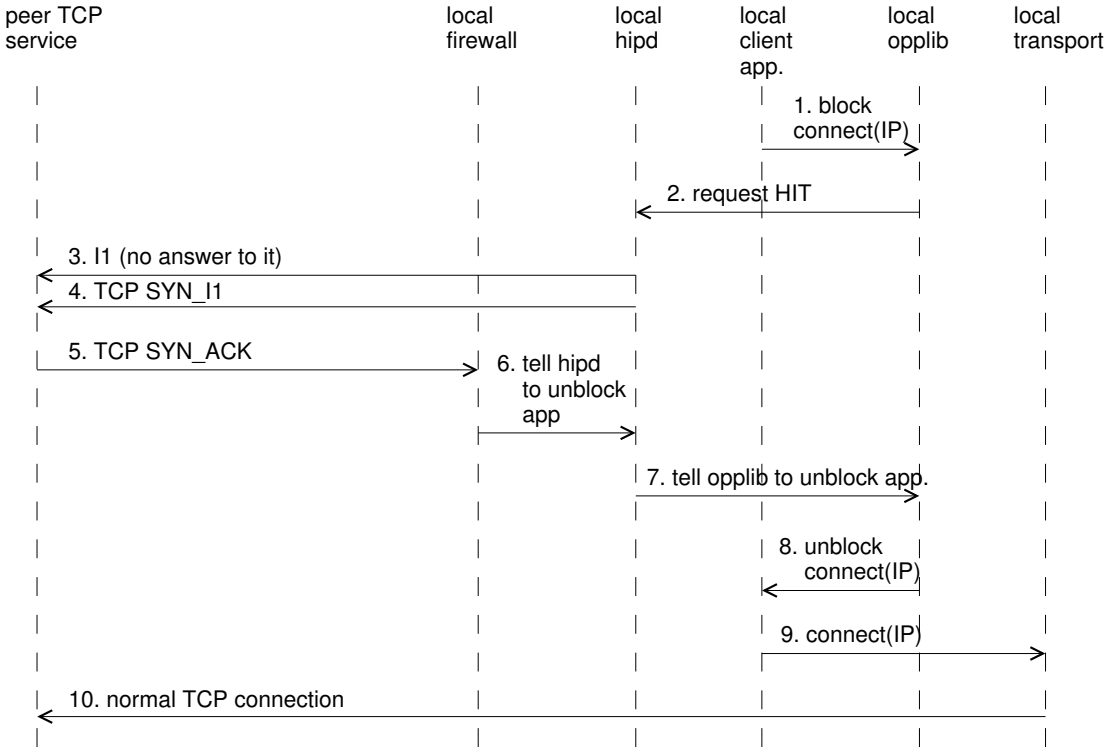

Figure 3.1: Flow Diagram of HIP detection and Fallback with non-HIP peer

plication, in step 6. The local HIP daemon asks the opportunistic library to unblock the application in step 7. The opportunistic library does this in step 8. In step 9, the connect call connect(IP) to the transport layer is unblocked, and the transport layer initiates a TCP connection with the non-HIP peer.

If the peer is a HIP host that supports our solution Figure 3.2, its firewall will drop any incoming TCP SYN_I1 packet. The peer HIP daemon replies to the I1 packet with an R1, in step 5. When the local HIP daemon receives the R1, it sends the HIT of the Responder to the opportunistic library, in step 6, and continues the base exchange with the peer. The opportunistic library makes a connect(HIT) call that shims the connect(IP) call that the application had done in step 7. Afterwards, application data flows from the transport layer to IPSec processing, where it is ESP-encapsulated and finally transmitted to network.

Our solution is valid only for TCP traffic. The reason is that the design needs a service at the peer that listens on a transport port and replies to the packet. We use a special option for the detection in the case of the TCP protocol. However, other transport protocols lack properties for this detection.

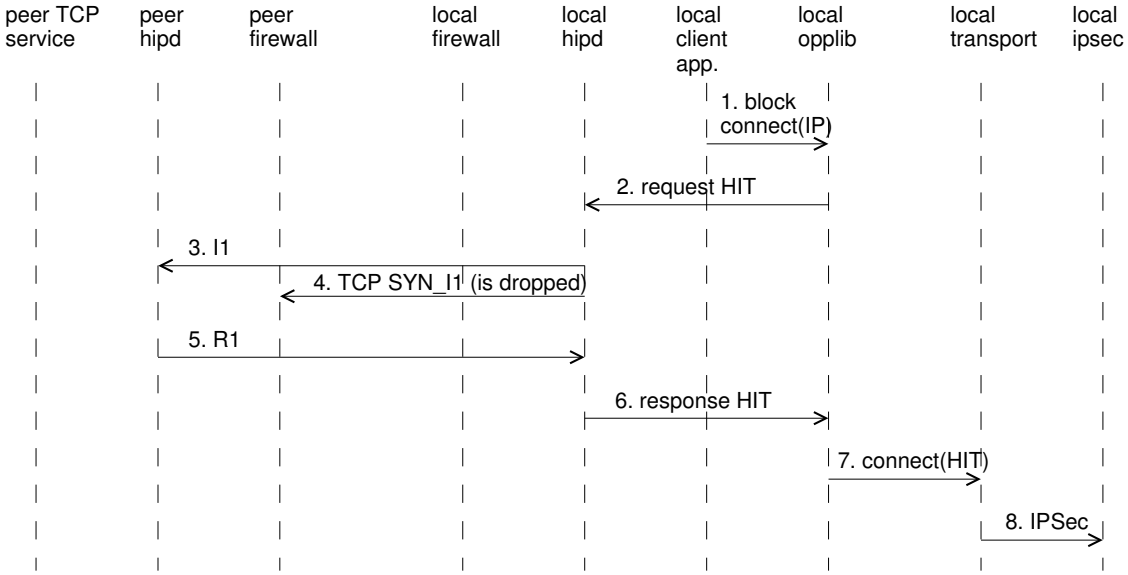

Figure 3.2: Flow Diagram of HIP detection with HIP peer

For example, the UDP protocol does not have a standard handshake for all application protocols, in which to specify options.

IP header options are not well supported in the Internet. Therefore, peer HIP detection cannot be solved at the network layer. Going one layer up to the transport one has the cost that each protocol needs to be tackled individually. However, the TCP protocol accounts for the majority of the Internet traffic and we consider to have solved a major part of the problem.

Chapter 4

Results and Analysis

4.1 Performance measurements

In this section, we present and analyze measurements of several scenarios. The focus of our measurements is the latency and the throughput in non-HIP and in HIP communication when our extension is used. Our solution was implemented as part of the HIPL project, release 1.0.4.

We tested latency in HTTP transfers, TCP throughput using Iperf version 2.0.2 [5] (the TCP/UDP Bandwidth Measurement Tool), and TCP handshake latency at the application layer. For the measurements, we used two computers. They were connected to each-other with an isolated switch. Both machines were running in multiuser mode and each had a 100 Mbit NIC. Following are the configuration details of both machines.

Initiator:
Intel(R) Pentium(R) 4 CPU 3.00GHz
Linux info 2.6.22.5.hipl #1 SMP PREEMPT Sun Nov 18 21:24:29 CET 2007 i686 GNU/Linux
Debian version 4.0
RAM capacity - 1034916 B
NIC - 0b:02.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8139/8139C/8139C+ (rev 10)

Responder:
Intel(R) Pentium(R) 4 CPU 3.00GHz
Linux blerta-pc 2.6.22.5.hipl #1 SMP PREEMPT Fri Feb 8 10:20:16 EET 2008 i686 GNU/Linux
Debian version 4.0
RAM capacity - 1034144 B

NIC - 02:01.0 Ethernet controller: Intel Corporation 82547EI Gigabit Ethernet Controller

We measured various aspects of the overhead caused by the extension we implemented. The results of tests and measurements for each examined aspect lie in a separate subsection. For each subsection, there are these communication scenarios: plain TCP/IP, standard HIP, HIP negative detection, and HIP positive detection. Plain TCP/IP means that there is non-HIP communication. Standard HIP means HIP communication between HIP hosts without HIP detection. HIP negative detection is the scenario in which the Initiator is a HIP host and supports our solution. It attempts communication with a non-HIP host and detects lack of HIP support at the peer. Therefore, there is TCP/IP communication afterwards. HIP positive detection refers to the scenario in which both the Initiator and the Responder are HIP hosts supporting our extension. The Initiator detects the HIP support at the peer and communicates with it using HIP.

Whenever the hosts in our tests use HIP, they use RSA keys for encryption. The public RSA keys are 216 bytes long for both machines. We repeated the measurements 40 times for each scenario and then calculated the average values and the standard deviation. Standard deviation values are shown in each bar of the charts in gray color. The data in the all the charts are depicted in logarithmic scale to bring out the differences in the values better.

4.1.1 HTTP Transfers

We analyzed HTTP transfers in the following cases: plain TCP/IP, standard HIP implementation, HIP negative detection, and HIP positive detection. To take the measurements, we tcpdumped all the traffic between the two hosts in every test of each scenario. Afterwards, we calculated the average and the standard deviation of the difference in time between the last and the first packets of each output of tcpdumped HTTP transfer. The y axes of the charts display the latency in milliseconds. The data was grouped into two charts to distinguish better between HIP and non-HIP communication.

The chart on the left shows the measurement data for the cases when peers do not communicate in a HIP way. The chart on the right displays data for HIP communication. The y axis in each of these charts displays the latency of the communication as observed by tcpdump Figure 4.1.

In the first chart, both bars display data for non-HIP communication. The bar for HIP negative detection shows a higher latency value due to detection of lack of HIP support at the peer. The time overhead can be broken down into additional processing, as well as round-trip delay of the TCP response

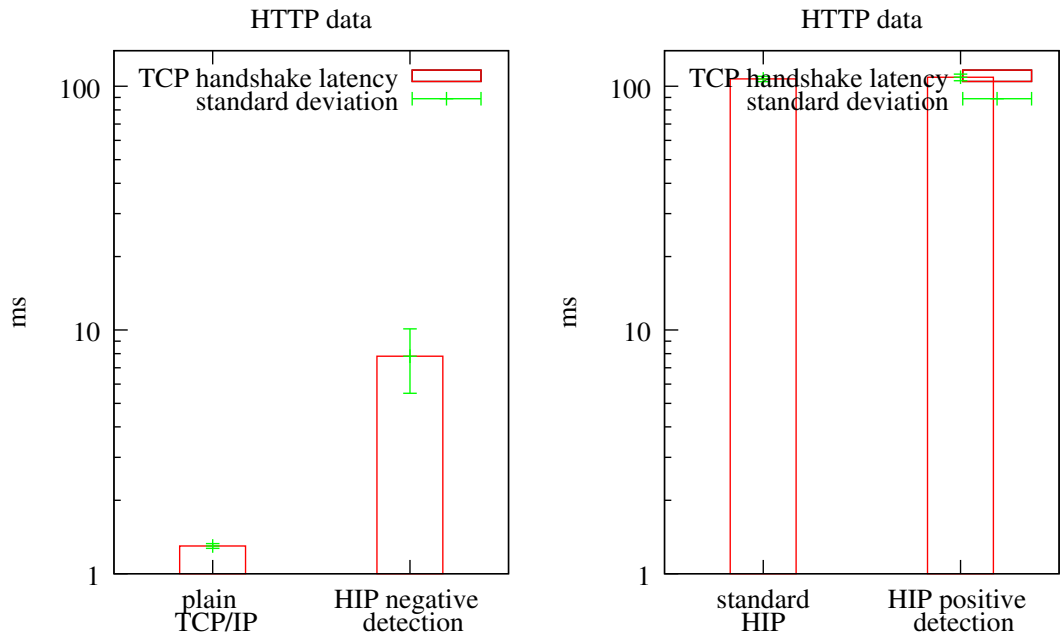

Figure 4.1: HTTP

from the peer. The additional time for HIP negative detection as compared to plain TCP/IP is 6.5 ms.

The second chart shows the values for HIP communication. The bar on the right shows a higher value since it contains the time consumed for HIP positive detection. The difference of the values is 1.7 ms.

We notice a smaller difference of values in the chart on the right. This was to be expected because HIP negative detection includes a TCP round trip delay. In HIP positive detection, HIP communication is initiated in parallel with HIP detection and introduces no extra round trip delay.

4.1.2 TCP Throughput

We made the measurements for TCP throughput using the Iperf tool. Iperf [5] measures the throughput of TCP and UDP. It also reports datagram loss, bandwidth, and jitter. These scenarios are analyzed with Iperf: plain TCP/IP, standard HIP implementation, HIP negative detection, and HIP positive detection. The scenarios are further separated into two charts based on whether there is HIP-based communication or not. The y axis in each of these charts displays the throughput of the communication in Mbits/sec. The TCP

throughput values were obtained from the Iperf output. Figure 4.2 shows the average values and the standard deviation.

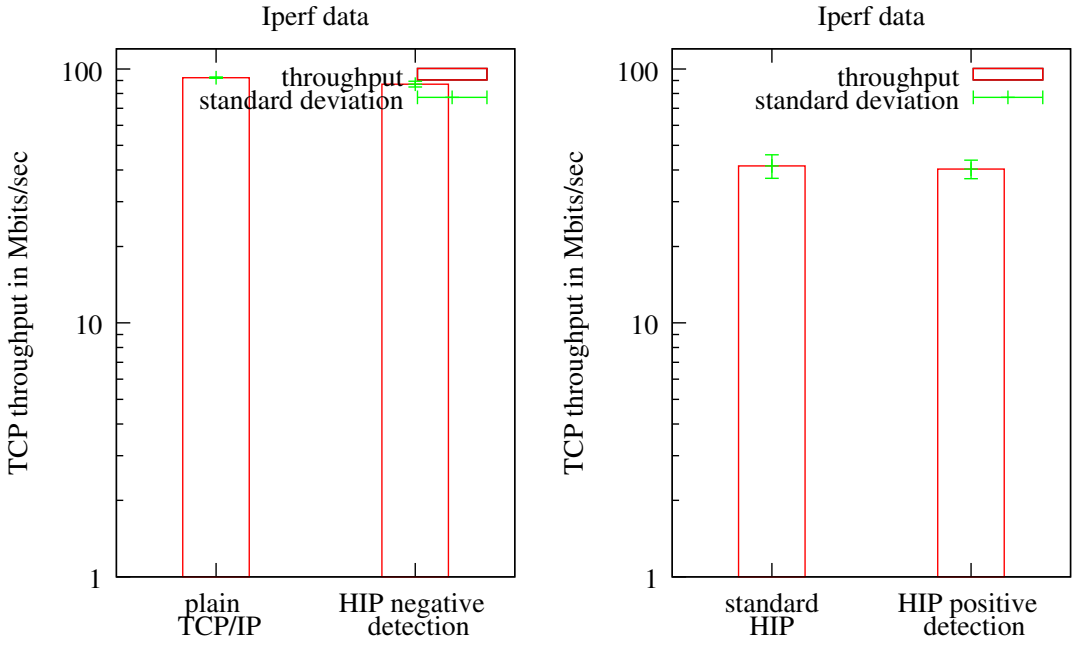

Figure 4.2: Iperf

As can be seen from comparing the charts in Figure 4.2, IPSec encryption in general reduces the throughput of the communication. The bars on the second chart have values that are lower than those in the first chart. Additionally, we notice that the throughput values when HIP detection takes place are smaller than when it does not. Namely, the bars on the right in each chart of Figure 4.2 have lower values than the respective bars on the left. This is expected since detection introduces processing overhead as well as TCP round trip delay in the case of HIP negative detection.

The chart on the left depicts the cases for non-HIP communication. The difference between the two throughput values is 5.25 Mbits/sec.

The chart on the right illustrates the case when HIP communication takes place, causing a smaller throughput value for both bars because of IPSec. When HIP positive detection takes place, the TCP throughput decreases even further. The actual difference in throughput is 1.17 Mbits/sec.

4.1.3 TCP Handshake Latency at the Application Layer

We measured latency as observed at the application layer and tcpdump latency. The measurements were in the following scenarios: plain TCP/IP, standard HIP implementation, HIP negative detection, and HIP positive detection. The y axes of the charts display the latency in milliseconds.

The first figure, Figure 4.3, contains a comparative view of the overall latency at the application layer and the latency at the network layer calculated with tcpdump. The purpose of this figure is to display the additional processing that takes place between the transport and the application layer.

In Figure 4.3, the red bars show the latency measured from the tcpdump of the communication. We measured it as the time difference between the timestamp of the last and the timestamp of the first tcpdumped packets. The green bars indicate the latency as seen at the application layer. We calculated this value as the sum of the latency for the creation of the connection with the latency of data exchange, both measured at the application layer.

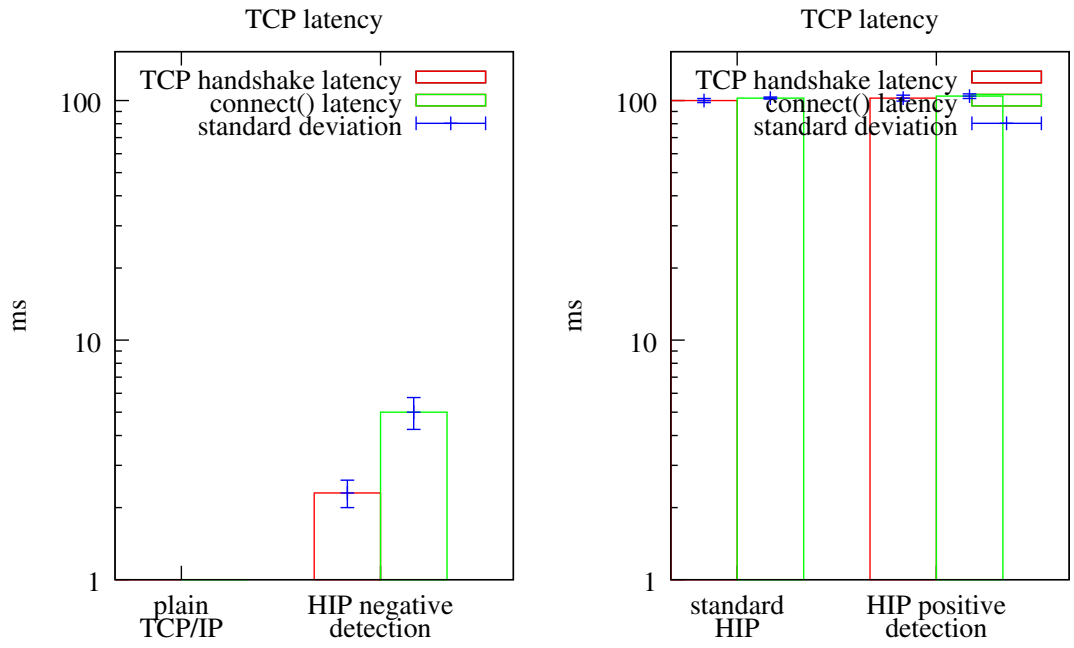

Figure 4.3: TCP handshake time

Since tcpdump measures latency at lower layers, we expect application layer latency values to be higher due to layer processing overhead and context switches. The values were measured at the Initiator.

The chart on the left Figure 4.3 shows a very small value when the commu-

nication is plain TCP/IP. The difference between the values of the two bars is smaller than a millisecond. When HIP negative detection takes place, this difference is 2.7 ms.

The chart on the right Figure 4.3 shows a difference of 2.4 ms in the case of HIP communication. When HIP positive detection takes place, the difference of the values is 2 ms.

As shown in Figure 4.3, processing above the network layer does not cause considerable latency. The differences in latency between the transport layer, and the application layer timers appear to be very close to each other. Indeed, this value is very small for plain TCP/IP. Starting from HIP negative detection, these differences in milliseconds are 2.7 ms, 2.5 ms, and 2 ms.

Further, we notice that HIP positive detection takes less time than HIP negative detection. This result is consistent with the results from the previous tests. The explanation for this is the same as the one we have already outlined in the previous subsections. Basically, HIP positive detection takes more time than HIP negative detection in our solution.

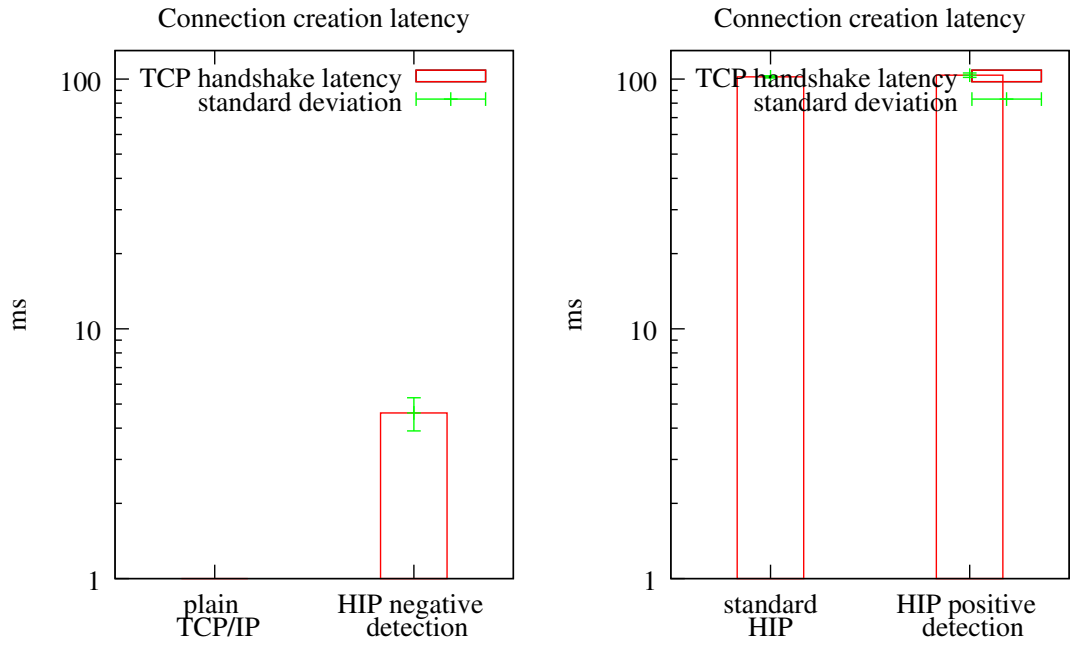

Figure 4.4: TCP socket creation latency

The purpose of the two charts in Figure 4.4 is to use the evaluation of the latency at the application layer for a better estimation of the cost of HIP positive and negative detection. In subsection 4.1.1, we calculated the processing

overhead in relation to the output of tcpdump. This time, we make an estimation from the point of view of the application layer. For the following figure, we used only the time used by the application to create a connection to the peer socket. Whenever HIP communication is attempted and/or established, HIP detection takes place before the creation of the connection to the socket. Therefore, we have the HIP detection time included in the measurements. Additionally, we do not take into account the timing of the data exchange in the connections since what is measured is only the time of connection creation. This makes our estimations of the overhead of HIP detection more accurate. The reason is that adding data exchange measurements contributes to making the calculations more inaccurate.

The first chart in Figure 4.4 displays the values we measured for non-HIP communication. The TCP connection creation takes more time when HIP negative detection takes place. The difference in time is 4.4 ms.

In the second chart, HIP-encrypted communication increases the values of latency significantly due to IPSec processing. The reason is that even the packets that create the TCP connection are encrypted. The difference in time for HIP positive detection is 1.6 ms.

This result also shows that the time difference for HIP positive detection in the chart on the right is higher than the time difference for HIP negative detection in the chart on the left. Still, both values seem to be relatively small.

Chapter 5

Future Work

This chapter analyzes the compatibility of our HIP extension with various parts of the HIP infrastructure. Here, we also investigate into how our solution can be extended further to support other transport layer protocols.

5.1 Protocol Analysis

5.1.1 Security Analysis

Our solution extends the opportunistic mode of the HIPL implementation. As a result, our design is also vulnerable to man-in-the-middle and replay attacks of the opportunistic mode, because an attacker that is in the path between the hosts can capture the I1 packet and send its own R1 packet as a response. Our solution also inherits the risk of replay attacks from the opportunistic mode since the Responder can reply with R1 packets that contain any HIT.

On-Path attacks

On-path attacks require the attacker to be in the path of communication between the hosts. Therefore, they are not easy to implement. If the peer is not a HIP host, dropping the I1 packet sent by the Initiator has no effect, since the Responder would not understand it or respond to it in any case. Dropping the TCP SYN_I1 packet causes DoS attack. When an attacker drops the TCP SYN_I1 packet sent to a HIP host that implements our extension, it has no effect because the Responder discards the packet anyway. If the I1 packet is discarded, down negotiation takes place because our solution inherits the timeout mechanism from the opportunistic mode. After not receiving either

a TCP or an R1 response from the peer for an amount of time, the Initiator identifies the peer as a non-HIP host and falls back on plain TCP/IP communication. To cause down negotiation, a man-in-the-middle needs to drop only the I1 packet when the peer is a legacy HIP host because legacy HIP hosts do not drop TCP SYN_I1 packets. If down negotiation is successful, the attacker can either eavesdrop on the connection or impersonate the Responder by sending a TCP SYN_ACK packet without the option or forging the HIT in R1.

Off-Path attacks

In our solution, the Initiator examines incoming TCP packets to conclude about HIP support at the peer. If an off-path attacker sends spoofed TCP packets, it might compromise the HIP communications. This introduces two security risks that are explained in the following paragraphs.

First, an off-path attacker can flood any HIP host that implements our solution with TCP SYN_ACK packets because these packets require more processing at the HIP firewall. This way, the netlink buffers can become congested causing the HIP firewall application to fail eventually. To boost the HIP firewall performance, and to minimize the consequences of such attacks, we use threads to avoid the congestion of the buffer queue. Additionally, attackers can use spoofed TCP SYN_ACK packets to overpopulate the HIP daemon state tables with data about IP addresses that the local host has not even tried to connect with. An overpopulated database might become a burden for the memory. To protect against this, the daemon implementation should keep track of the IP addresses where it sends opportunistic I1 packets and discard R1 packets from unknown IP addresses.

Second, off-path attackers sending spoofed TCP SYN_ACK packets that contain the source IP addresses of HIP hosts can compromise peer support information in the HIP daemon. This causes down-negotiation because the HIP daemon wrongly identifies the HIP-hosts at those IP addresses as non-HIP hosts.

These attacks are difficult to implement in practice. Additionally, an attacker cannot launch these attacks from outside a network if the network is protected with a firewall, because the TCP SYN_ACK packets would not be allowed to pass.

TCP SYN cookies is a mechanism to protect against TCP SYN flooding attacks aimed at exhausting the queue resources of TCP servers, section 2.3. This defense consists in assigning special values to the sequence numbers of the TCP SYN_ACK packets instead of creating and adding a new entry to

the SYN queue of the TCP server. When TCP ACK packets arrive, the TCP server checks the validity of the next required sequence number in the packet. If it is valid, the server builds the SYN queue entry based on the received packet, and then creates the TCP connection.

Even though SYN cookies are generally used to protect TCP SYN packet receivers, we can use the same mechanism to protect TCP SYN_ACK packet receivers instead. The following describes future work how the mechanism used in TCP SYN cookies can be used to protect HIP Initiators that support our extension. It is not implemented in our extension. The HIP Initiator uses TCP SYN_ACK packets as indicators of lack of peer HIP support. Therefore, the HIP Initiator needs protection from bogus TCP SYN_ACK packets. To avoid such attacks, the Initiator can attach special values to the sequence numbers of the TCP SYN_I1 packets that it sends out. When a TCP SYN_ACK packet arrives, the Initiator checks its validity by checking the next expected sequence number in it. This defense can be combined with the HIP Initiator remembering TCP SYN_I1 packets sent out recently. This way, the HIP Initiator could discard directly TCP SYN_ACK packets for which there is no matching, recent entry of an outgoing TCP SYN_I1 packet. Additionally, the HIP Initiator can make checks before changing the database with information for other hosts. For example, the Initiator can check that when a TCP SYN_-ACK packet arrives from a peer, there is a pending request for connection to that peer. Moreover, the HIP host can disallow down-grading of peer HIP support to plain TCP/IP.

5.1.2 Compatibility with RVS

In this section, we assume the deployment of the HIP DNS infrastructure. The HIP DNS RRs contain the HI, the HIT, and the domain names of the RVSs of the HIP host or the address of the peer itself. Assuming that there is no attack on the DNS, an Initiator can find out whether a peer supports HIP from the reply to its HIP DNS query. The following discussion assumes that both the Initiator and the Responder have publicly reachable addresses. Otherwise, neither the RVS forwarding of I1 packets to the Responder, nor sending TCP SYN packets from the Initiator to the Responder is possible due to NAT restrictions [33].

When the Initiator finds the peer HIP data in the DNS, it knows that the peer supports HIP. The Initiator sends the I1 and the TCP SYN_I1 packet to the Responder or to the RVS depending on the result of the HIP DNS query. If the packets are sent directly to the Receiver, it will drop the TCP SYN_I1 packet and reply to I1. Otherwise, if the packets are sent to the RVS, the

RVS forwards only the I1 packet to the Responder. This does not harm HIP detection because the Initiator knows already that the peer supports HIP.

It is possible that the HIP Initiator does not find HIP related identifier information in the DNS for the peer, but only the peer IP. In this case, the Initiator can use the opportunistic mode with our extension when communicating directly with the peer.

5.1.3 NAT traversal

In order to work, our solution needs to be able to be compatible with NATs. [15] describes HIP extensions for NAT traversal, but very little is said about NAT traversal in the opportunistic mode. HIP extensions for NAT traversal have to be further extended to support the opportunistic mode, as well as our solution.

Peer HIT is known

Let us assume that we have two hosts that support standard HIP. Both hosts are behind NATs, and we have the HIP Relay infrastructure in place. If the Initiator has the peer HIT, then it can communicate with the peer through the HIP Relay and later through the TURN server or directly with each other after successful ICE connectivity tests. Both hosts have initially registered with the TURN server and with the HIP Relay. This is the case that HIP extensions for NAT traversal have already handled and solved.

NAT traversal and the Opportunistic Mode

Let us now examine the opportunistic mode of the current HIP implementation with the NAT traversal infrastructure. We assume there are two HIP hosts supporting the standard HIP positioned behind NATs. The Initiator does not know the HIT of the Responder and cannot initiate communication through the relay in the currently standardized way. The Initiator has to obtain the peer IP address in advance. Additionally, if the HIP Relay serves multiple Responders using the same address, the Initiator needs to obtain the peer port number. The Initiator cannot start a connection with the Responder directly, since the peer is behind NAT. Therefore, HIP NAT traversal needs to support a base exchange that starts with an opportunistic I1 packet that contains the IP address and port through which to contact the peer. Then, the HIP Relay needs to check if a HIP host with that data has already registered with the relay. If the other peer has not already registered, the HIP relay is

unable to initiate a connection with the peer from outside the NAT. If the peer is registered, the relay should forward the I1 packet after removing the IP-port parameter from it and after replacing the empty peer HIT with the one it has mapped using the IP-port parameter. This way, the communication appears as a normal base exchange to the Responder.

The HIP Relay should be extended to support the HIP opportunistic mode as described in the previous paragraph. These changes should be part of the relay extensions to support our solution because we use the opportunistic I1 packet as well. Figure 5.1 illustrates how the HIP Relay can be extended to support the opportunistic base exchange.

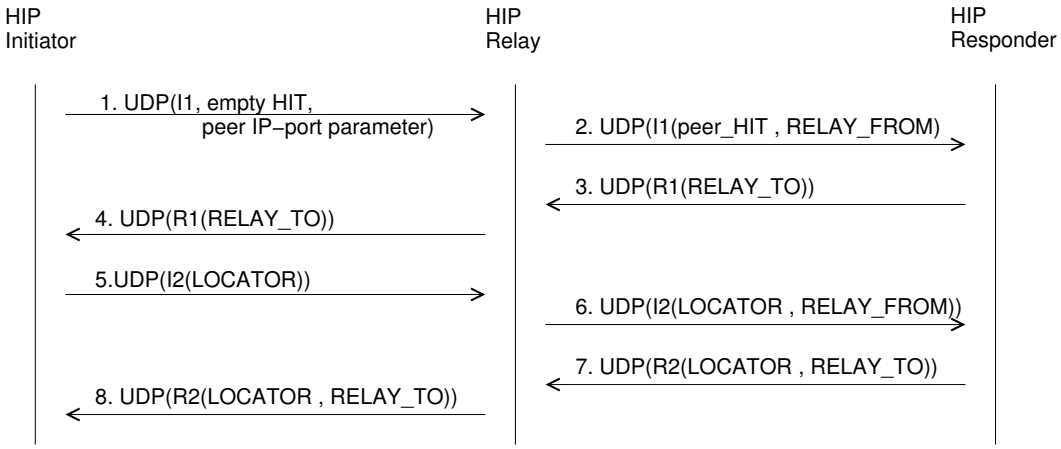

Figure 5.1: Opportunistic Base Exchange via a HIP Relay

The issue with this extension to HIP NAT traversal for supporting the opportunistic mode is to solve where the Initiator finds the IP address and the port of the peer. Additionally, there is a security issue with sending the IP address and the port of the peer unencrypted in the I1 packet to the HIP Relay. There is privacy loss and risk of DoS attacks on the peer.

NAT traversal and our Extension

Let us consider next how our extension to HIPL should work with the HIP relay. In our solution, the Initiator sends out an I1 packet, as well as a TCP packet for the detection. The I1 packet initiates the base exchange immediately in case the peer is HIP-enabled. When the peer does not support HIP, the reply to the TCP packet indicates the lack of HIP support.

When the peer does not support HIP, there is no reason for it to register with the HIP Relay. Therefore, a HIP host cannot communicate with a non-HIP

host through the HIP relay. In the previous paragraph, we argued that the initial TCP packet needed to reach the peer only when the peer did not support HIP. As a result, we conclude that the HIP Relay does not need to relay TCP packets in order to fit our extension.

Let us examine the case when both the Initiator and the Responder support our extension and the HIP Relay is as currently standardized. The Initiator sends an I1 packet and a TCP SYN_I1 packet through the relay. Since the relay does not currently relay TCP packets, only the I1 packet reaches the Responder after being forwarded by the HIP relay. The processing of opportunistic I1 packets at the HIP Relay should be done as previously defined in the previous subsection. The hosts continue with the rest of the packets of the base exchange as standardized in [15].

Let us now suppose that we have two HIP hosts behind NATs, the Initiator supports our extension and the Responder is a legacy HIP host, and the HIP Relay is extended to support the opportunistic mode. In our extension, a HIP host attempts both HIP and non-HIP communication to an unknown host. Therefore, the Initiator sends out both an opportunistic I1 packet, as well as a TCP SYN_I1 packet. The HIP Relay forwards the I1 packet to the peer with the necessary changes depending on whether it contains the peer HIT or the IP-port parameter, and drops the TCP packet. The TCP packet never reaches the standard HIP peer, which is correctly identified as a HIP host because the I1 packet triggers HIP communication.

5.2 Support only for TCP

Our solution has the drawback that it works only for the TCP protocol. Possible future work would be to do the same for other protocols, even though it would not be easy. Not all transport layer protocols have properties to sustain HIP detection. For example, UDP does not have a handshake mechanism in which to negotiate for HIP detection. However, the opportunistic mode can be used with UDP-based communication but without the performance improvements of our extension.

Chapter 6

Conclusion

HIP provides network layer security and supports end-host mobility and multi homing. Therefore, hosts on the Internet would benefit from it. However, they are not expected to adopt HIP all at the same time. An additional deployment obstacle is the lack of HIP infrastructure to look up HIP name information. Until there is adequate name infrastructure, HIP has to operate without it.

Before HIP becomes widely deployed, it has to be backwards compatible in an efficient manner. Current HIP implementation detects peer HIP support using timeouts, but the latency is an obstacle to normal user experience. In this thesis, we extend the HIP implementation of the HIPL project to overcome the deployment obstacles.

Our solution uses TCP options to detect peer HIP support. TCP options are widely supported in the Internet. With this design alternative, our solution is beneficial only to TCP traffic. However, most of the traffic in Internet is TCP traffic and we argue that we have tackled the most imminent problem. The opportunistic HIP mode based on timeouts is still applicable to other transport layer protocols, such as UDP, but without the performance improvements shown in this thesis.

We conducted tests on the performance of our solution measuring the latency and throughput. The results showed that our extension to HIP did not cause additional performance problems to the implementation. On the other hand, user experience seems to benefit as the latency is significantly reduced.

We have analyzed the security implications of our solution in different scenarios. Furthermore, we reviewed the compatibility of our solution with current specifications and drafts, such as the HIP DNS infrastructure, and the HIP NAT traversal specification. As a result, we provided some possible guidelines for future work.

The most important benefit of our solution is the implementation of efficient detection of HIP support at an unknown peer. This detection replaces the timeout mechanism, and improves user experience. Another benefit of our solution is that it maintains backwards compatibility with HIP hosts that do not support our extension. HIP hosts that implement our extension are able to detect HIP support at legacy HIP hosts. Thus, our extension does not have to be deployed at servers.

Bibliography

[1] *HIPL: HIP for Linux.* `http://infrahip.hiit.fi/index.php?index=hipl`.

[2] *Linux IPv4 raw sockets.* Linux Programmer's Manual, SOCK_RAW.

[3] *Linux Userspace Packet Queuing.* Linux Programmer's Manual, libipq.

[4] *Netfilter.* The netfilter.org project.

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

[6] Mark Allman Alberto Medina and Sally Floyd. Measuring interactions between transport protocols and middleboxes, 2004. ICSI Center for Internet Research.

[7] Mark Allman Alberto Medina and Sally Floyd. *Measuring the Evolution of Transport Protocols in the Internet.* **ACM!**, April 2005. SIGCOMM CCR Volume 35 Issue 2.

[8] R. Austein D. Atkins. *RFC 3833: Threat Analysis of the Domain Name System (DNS).* Internet Engineering Task Force, August 2004. `http://www.ietf.org/rfc/rfc3833.txt`.

[9] Andrei Gurtov. *Host Identity Protocol (HIP): Towards the Secure Mobile Internet.* Wiley, June 2008.

[10] HIP for inter.net Project. *hip4inter.net.* `http://www.hip4inter.net/`.

[11] P. Matthews J. Rosenberg, R. Mahy. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun), January 2008. [Internet Draft] `http://tools.ietf.org/html/draft-ietf-behave-turn-06`.

[12] Miika Komu and Janne Lindqvist. Leap-of-faith security is enough for mobility, March 2008. Unpublished manuscript.

[13] J. Lindqvist. Establishing host identity protocol opportunistic mode with tcp option, March 2006. [Internet Draft] `http://www.tools.ietf.org/html/draft-lindqvist-hip-opportunistic-01`.

[14] J. Kangasharju M. Komu, S. Tarkoma and A. Gurtov. Applying a cryptographic namespace to applications, 2005.

[15] P. Matthews M. Komu, T. Henderson, H. Tschofenig, A. Keraenen, J. Melen, and M. Bagnul. Basic hip extensions for traversal of network address translators and firewalls, February 2008. [Internet Draft] `http://www.ietf.org/internet-drafts/draft-ietf-hip-nat-traversal-03.txt`.

[16] L. Eggert M. Stiemerling, J. Quittek. *Middlebox Traversal of HIP Communication*. IRTF Host Identity Protocol (HIP) Research Group, November 2004. Workshop on HIP and Related Architectures.

[17] L. Eggert M. Stiemerling, J. Quittek. Middlebox traversal issues of host identity protocol (hip) communication, July 2005. [Internet Draft] `http://tools.ietf.org/html/draft-stiemerling-hip-nat-05`.

[18] P. Mockapetris. *RFC 1035: DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION*. Internet Engineering Task Force, November 1987. `http://www.ietf.org/rfc/rfc1035.txt`.

[19] Robert Moskowitz and Pekka Nikander. *Host Identity Protocol (HIP) Architecture*. Internet Engineering Task Force, May 2006. RFC 4423.

[20] Microsoft Developer Network (msdn). *TCP/IP Raw Sockets*. Microsoft Corporation. `http://msdn.microsoft.com/en-us/library/ms740548.aspx`.

[21] The OpenHIP project. *The OpenHIP project*. `http://www.openhip.org/`.

[22] Shunsuke Oshima and Takuo Nakashima. Performance evaluation for linux under syn flooding attacks. In *ICICIC '07: Proceedings of the Second International Conference on Innovative Computing, Informatio and Control*, page 132, Washington, DC, USA, 2007. IEEE Computer Society.

[23] R. Moskowitz P. Jokela and P. Nikander. *RFC 5202: Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)*. Internet Engineering Task Force, April 2008. `http://www.ietf.org/rfc/rfc5202.txt`.

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

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

[26] Jukka Ylitalo Pekka Nikander and Jorma Wall. Integrating security, mobility, and multi-homing in a hip way, February 2003.

[27] Jon Postel. *RFC 791: Internet Protocol.* Internet Engineering Task Force, September 1981. `http://www.ietf.org/rfc/rfc791.txt`.

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

[29] M. Larson R. Arends, R. Austein, D. Massey, and S. Rose. *RFC 4033: DNS Security Introduction and Requirements.* Internet Engineering Task Force, March 2005. `http://www.ietf.org/rfc/rfc4033.txt`.

[30] P. Nikander R. Moskowitz, Ed. P. Jokela, and T. Henderson. *RFC 5201: Host Identity Protocol.* Internet Engineering Task Force, April 2008. `http://www.ietf.org/rfc/rfc5201.txt`.

[31] R. Hinden S. Deering. *RFC 2460: Internet Protocol, Version 6 (IPv6).* Internet Engineering Task Force, December 1998. `http://www.ietf.org/rfc/rfc2460.txt`.

[32] Rodrigo Fonseca George Porter Randy H. Katz Scott Shenker. Ip options are not an option, December 2005. Technical Report No. UCB/EECS-2005-24.

[33] Lauri Silvennoinen. Legacy network address translator traversal using the host identity protocol, October 2007.

[34] Essi Vehmersalo. Host identity protocol enabled firewall, September 2005.

Appendix A

Application Code Examples

A.1 Sending a raw TCP packet

```
/**
 * adds the i1 option to a packet if required
 * adds the default HIT after the i1 option (if i1 option should be added)
 * and sends it off with the correct checksum

 * trafficType - 4 or 6 - standing for ipv4 and ipv6
 */

/**
 * Sends a TCP packet through a raw socket.
 *
 * @param *ptr pointer to an integer that indicates
 *  the type of traffic: 4 - ipv4; 6 - ipv6.
 * @param *ptr


 * @return nothing, this function loops forever,
 *  until the firewall is stopped.
 */

int send_tcp_packet(void *hdr,
    int   newSize,
    int   trafficType,
    int   sockfd,
```

```
    int   addOption,
    int   addHIT){

int    on = 1, i, j, err = 0, off = 0;
int    hdr_size, newHdr_size, twoHdrsSize;
char  *packet;
char  *bytes =(char*)hdr;
struct sockaddr_in  sin_addr;
struct sockaddr_in6 sin6_addr;
struct in_addr  dstAddr;
struct in6_addr dst6Addr;
struct tcphdr *tcphdr;
struct tcphdr *newTcphdr;
struct ip * iphdr;
struct ip * newIphdr;
struct ip6_hdr * ip6_hdr;
struct ip6_hdr * newIp6_hdr;
struct pseudo_hdr  *pseudo;
struct pseudo6_hdr *pseudo6;
void  *pointer;
struct in6_addr *defaultHit = HIP_MALLOC(sizeof(char)*16, 0);
char   newHdr [newSize + 4*addOption + (sizeof(struct in6_addr))*addHIT];
char  *HITbytes;

if(addOption)
newSize = newSize + 4;
if(addHIT)
newSize = newSize + sizeof(struct in6_addr);

//initializing the headers and setting socket settings
if(trafficType == 4){
//get the ip header
iphdr = (struct ip *)hdr;
//get the tcp header
hdr_size = (iphdr->ip_hl * 4);
tcphdr = ((struct tcphdr *) (((char *) iphdr) + hdr_size));
//socket settings
sin_addr.sin_family = AF_INET;
sin_addr.sin_port   = htons(tcphdr->dest);
sin_addr.sin_addr   = iphdr->ip_dst;
}
else if(trafficType == 6){
```

```
//get the ip header
ip6_hdr = (struct ip6_hdr *)hdr;
//get the tcp header
hdr_size = sizeof(struct ip6_hdr);
tcphdr = ((struct tcphdr *) (((char *) ip6_hdr) + hdr_size));
//socket settings
sin6_addr.sin6_family = AF_INET6;
sin6_addr.sin6_port   = htons(tcphdr->dest);
sin6_addr.sin6_addr   = ip6_hdr->ip6_dst;
}

//measuring the size of ip and tcp headers (no options)
twoHdrsSize = hdr_size + 4*5;

//copy the ip header and the tcp header without the options
memcpy(&newHdr[0], &bytes[0], twoHdrsSize);

//get the default hit
if(addHIT){
hip_get_default_hit(defaultHit);
HITbytes = (char*)defaultHit;
}

//add the i1 option and copy the old options
//add the HIT if required,
if(tcphdr->doff == 5){//there are no previous options
if(addOption){
newHdr[twoHdrsSize]     = (char)HIP_OPTION_KIND;
newHdr[twoHdrsSize + 1] = (char)2;
newHdr[twoHdrsSize + 2] = (char)1;
newHdr[twoHdrsSize + 3] = (char)1;
if(addHIT){
//put the default hit
memcpy(&newHdr[twoHdrsSize + 4], &HITbytes[0], 16);
}
}
else{
if(addHIT){
//put the default hit
memcpy(&newHdr[twoHdrsSize], &HITbytes[0], 16);
}
}
```

```
}
else{//there are previous options
if(addOption){
newHdr[twoHdrsSize]     = (char)HIP_OPTION_KIND;
newHdr[twoHdrsSize + 1] = (char)2;
newHdr[twoHdrsSize + 2] = (char)1;
newHdr[twoHdrsSize + 3] = (char)1;

//if the HIT is to be sent, the
//other options are not important
if(addHIT){
//put the default hit
memcpy(&newHdr[twoHdrsSize + 4], &HITbytes[0], 16);
}
else
memcpy(&newHdr[twoHdrsSize + 4], &bytes[twoHdrsSize], 4*(tcphdr->doff-5));
}
else
{
//if the HIT is to be sent, the
//other options are not important
if(addHIT){
//put the default hit
memcpy(&newHdr[twoHdrsSize], &HITbytes[0], 16);
}
else
memcpy(&newHdr[twoHdrsSize], &bytes[twoHdrsSize], 4*(tcphdr->doff-5));
}
}

pointer = &newHdr[0];

//get pointers to the new packet
if(trafficType == 4){
//get the ip header
newIphdr = (struct ip *)pointer;
//get the tcp header
newHdr_size = (iphdr->ip_hl * 4);
newTcphdr = ((struct tcphdr *) (((char *) newIphdr) + newHdr_size));
}
else if(trafficType == 6){
//get the ip header
```

```
newIp6_hdr = (struct ip6_hdr *)pointer;
//get the tcp header
newHdr_size = (newIp6_hdr->ip6_ctlun.ip6_un1.ip6_un1_plen * 4);
newTcphdr = ((struct tcphdr *) (((char *) newIp6_hdr) + newHdr_size));
}

//change the values of the checksum and the tcp header length(+1)
newTcphdr->check = 0;
if(addOption)
newTcphdr->doff = newTcphdr->doff + 1;
if(addHIT)
newTcphdr->doff = newTcphdr->doff + 4;//16 bytes HIT - 4 more words

//the checksum
if(trafficType == 4){
pseudo = (struct pseudo_hdr *) ((u8*)newTcphdr - sizeof(struct pseudo_hdr));

pseudo->s_addr = newIphdr->ip_src.s_addr;
pseudo->d_addr = newIphdr->ip_dst.s_addr;
pseudo->zer0    = 0;
pseudo->protocol = IPPROTO_TCP;
pseudo->length  = htons(sizeof(struct tcphdr) + 4*(newTcphdr->doff-5) + 0);

newTcphdr->check = in_cksum((unsigned short *)pseudo, sizeof(struct tcphdr) +
4*(newTcphdr->doff-5) + sizeof(struct pseudo_hdr) + 0);
}
else if(trafficType == 6){
pseudo6 = (struct pseudo6_hdr *) ((u8*)newTcphdr - sizeof(struct pseudo6_hdr));

pseudo6->s_addr = newIp6_hdr->ip6_src;
pseudo6->d_addr = newIp6_hdr->ip6_dst;
pseudo6->zer0    = 0;
pseudo6->protocol = IPPROTO_TCP;
pseudo6->length  = htons(sizeof(struct tcphdr) + 4*(newTcphdr->doff-5) + 0);

newTcphdr->check = in_cksum((unsigned short *)pseudo6, sizeof(struct tcphdr) +
4*(newTcphdr->doff-5) + sizeof(struct pseudo6_hdr) + 0);
}

//replace the pseudo header bytes with the correct ones
memcpy(&newHdr[0], &bytes[0], hdr_size);
```

```
if(setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0 ){
HIP_DEBUG("Error setting an option to raw socket\n");
return;
}

//finally send through the socket
err = sendto(sockfd, &newHdr[0], newSize, 0, (struct sockaddr *)&sin_addr, sizeo

out_err:
if(defaultHit)
HIP_FREE(defaultHit);

setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, (char *)&off, sizeof(off));

return err;
}
```

A.2 Reading packets with ipq

```
/**
 * Analyzes packets.

 * @param *ptr pointer to an integer that indicates
 *  the type of traffic: 4 - ipv4; 6 - ipv6.
 * @return nothing, this function loops forever,
 *  until the firewall is stopped.
 */
int hip_fw_handle_packet(char *buf, struct ipq_handle *hndl, int ip_version, hip
{
// assume DROP
int verdict = 0;

// same buffer memory as for packets before -> re-init
memset(buf, 0, BUFSIZE);

/* waits for queue messages to arrive from ip_queue and
 * copies them into a supplied buffer */
if (ipq_read(hndl, buf, BUFSIZE, 0) < 0)
```

```
{
HIP_PERROR("ipq_read failed: ");
// TODO this error needs to be handled seperately -> die(hndl)?
goto out_err;
}

/* queued messages may be a packet messages or an error messages */
switch (ipq_message_type(buf))
{
case NLMSG_ERROR:
HIP_ERROR("Received error message (%d): %s\n", ipq_get_msgerr(buf), ipq_errstr()
goto out_err;
break;
case IPQM_PACKET:
HIP_DEBUG("Received ipqm packet\n");
// no goto -> go on with processing the message below
break;
default:
HIP_DEBUG("Unsupported libipq packet\n");
goto out_err;
break;
}

// set up firewall context
if (hip_fw_init_context(ctx, buf, ip_version))
goto out_err;

HIP_DEBUG("packet hook=%d, packet type=%d\n", ctx->ipq_packet->hook, ctx->packet

// match context with rules
if (hip_fw_handler[ctx->ipq_packet->hook][ctx->packet_type]) {
verdict = (hip_fw_handler[ctx->ipq_packet->hook][ctx->packet_type])(ctx);
} else {
HIP_DEBUG("Ignoring, no handler for hook (%d) with type (%d)\n");
}

 out_err:
if (verdict) {
HIP_DEBUG("=== Verdict: allow packet ===\n");
allow_packet(hndl, ctx->ipq_packet->packet_id);
} else {
HIP_DEBUG("=== Verdict: drop packet ===\n");
```

```
drop_packet(hndl, ctx->ipq_packet->packet_id);
}

// nothing to clean up here as we re-use buf, hndl and ctx

return 0;
}
```