IP Security Extensions for Explicit Congestion Control Protocols

Nuno Gonçalo de Castro Plácido Salta

March 2008

Master's Thesis in Electrical and Computer Engineering
INESC-Porto Supervisor: Filipe Lameiro Abrantes
FEUP Supervisor: Manuel Pereira Ricardo

(President of the Jury)

FEUP Universidade do Porto
Faculdade de Engenharia

Abstract

This dissertation addresses the current interoperability limitations of end-to-network protocols such as explicit congestion control protocols, with the present IP security extensions, the IPsec. The current stack layout, with the IPsec appearing immediately after the IP network layer restricts the use of such congestion control protocols since the necessary information to the middle nodes would appear inaccessible.

In this project, is made a discussion over design considerations and implementation details on altering IPsec in order to accommodate explicit congestion control protocols such as XCP, eXplicit Control Protocol, without loss of functionality, thereby allowing congestion control to be performed securely in environments characterized by an erosion of trust.

The proposed implementation in based on the Information Sciences Institute's XPC release and in the open source and highly acclaimed operating system, the FreeBSD.

Contents

List of Acronyms

AES Advanced Encryption Standard

AH Authentication Header

AIMD Additive Increase, Multiplicative Decrease

API Application Programming Interface

BSD Berkeley Software Distribution

CBC Cipher Block Chaining

DCCP Datagram Congestion Control Protocol

DES Data Encryption Standard

DNS Domain Name System

DNSSEC Domain Name System Security Extensions

DVB Digital Video Broadcasting

ECN Explicit Congestion Notification

ESP Encapsulating Security Protocol

FPGA Field-programmable Gate Array

HMAC Hash Message Authentication Code

ISI Information Sciences Institute, University of South California

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPsec Internet Protocol Security Extensions

IPv4 Internet Protocol Version 4

IPv6 Internet Protocol Version 6

LDAP Lightweight Directory Access Protocol

LSI Local Scope Identifier

MD5 Message-Digest algorithm 5

NAT Network Address Translation

RR Resource Record

RTT Round-trip Time

SA Security Association

SAD Security Association Database

SHA Secure Hash Algorithm

SPI Security Parameters Index

TCP Transmission Control Protocol

UDP User Datagram Protocol

UMTS Universal Mobile Telecommunications System

VPN Virtual Private Network

WLAN Wireless Local Area Network

XCP eXplicit Control Protocol

List of Figures

List of Tables

Auxiliary Listings

Chapter 1

Introduction

1.1 Overview

In the recent years we have assisted to the emergence of heterogeneous and mobile networks. Lossy and high latency links such WLAN puts to test the current Internet paradigm. As the topologies rapidly change, the long term Internet protocols strive to keep the pace.

Congestion control is fundamental to maintain the stability and efficiency of networks. The Transmission Control Protocol, TCP [1], was created to offer reliable transmissions. The retransmission policy of the original TCP threatened the whole stability of the Internet. By doubling the data rate upon a packet lost, TCP was aggravating the network congestion.

Van Jacobson introduced [2] congestion control mechanisms in TCP. This mechanisms allowed backward compatibility whilst adding an unprecedent stability to the Internet, prolongated for decades.

As the high bandwidth and high latency networks grow, the old congestion control mechanisms start to reach the limits. New congestion control paradigms arise such the explicit congestion control protocols like XCP [3] and RCP [4]. By using feedback from the end nodes, the sender can estimate better the current congestion along the path and adjust its throughput.

One other concern in networking is security. Since 1995 [5] IP provides security services to traffic flow. These services include confidentiality and authentication. For the new IP version, IPv6, the support of this security framework is mandatory, revealing the increase interest on transmission security. With the increase of node mobility, both congestion control and security are important parameters in other to deploy efficient and secure

networks. The interoperability of both should be a matter of interest.

1.2 Objectives

The goal of this project is to integrate a new breed of congestion control mechanisms into the standard IP Security Extensions, IPsec. Having identified key requirements and currently unsolved issues in the interoperability between congestion control protocols, the objectives span features in aiding the future standardization and deployment of congestion control protocols, namely:

- **FreeBSD stack review** Gain knowledge of the current TCP/IP stack implementation. Identify and understand the several functions that comprise the Transport and Network layers as well as the IPsec function. Also take into account the code *convention* and best practices.
- **IPsec restrictions** Locate the current limitations in the IPsec routines and elaborate a solution to overcome those restrictions to allow the integration of XCP within IPsec.
- **XCP (eXplicit Control Protocol) integration** Deploy the proposed solution for both IP versions and validate through testing the several IPsec services.

1.3 Document Structure

The outline of this dissertation is as follow: chapter 2 makes a review of the current IP security extension and the eXplicit Control Potrocol. Also is done some insight into FreeBSD TCP/IP implementation. Chapters 3 and 4 cover the proposed XCP integration with IPsec. Chapter 3 makes a detailed description on the implementation in IPv4 whilst the chapter 5 describes the implementation for IPv6. The chapter 5 covers the results obtained in the use of the implementation showing several scenarios for the diverse possible combinations of the IP Security servicies. Finally, the Chapter 6 makes conclusions upon the work validation and the obtained results.

Chapter 2

Technical Background

2.1 IP Security

IPSec [6] is a Security Architecture for the Internet Protocol developed by the IETF to offer interoperable security services at the IP network layer, both IPv4 and IPv6 and thus enclosing all the upper layers. The standard was initial specified in 1995 [5] and was updated and extended ever since.

The IPsec does not define specific functionalities and algorithms but instead provides a flexible framework to allow extensible integration of new developments. IPsec addresses critical security concerns such as authentication, confidentiality, data integrity and anti-replay protection. Allowing endpoints to negotiate asymmetrically the use of algorithms and associated parameters enables IPsec to be scalable and meet the user requirements, while interoperability between legacy equipment is guaranteed through the specification of a minimal set of encryption algorithms that all IPsec implementations must recognize.

These factors, allied to standardization, contributed to the widespread deployment of IPsec, rapidly becoming the $defacto$ network layer security solution, especially on VPN tunnels.

The support of the IPsec protocol suite is mandatory on the IPv6 implementation.

2.1.1 IPsec operation

The IPsec functionalities relies on two main protocols: the **Authentication Header (AH)** [7] and **Encapsulating Security Payload (ESP)** [8]. Both protocols are described in their respective RFC and they are performed on IP packets in order to provide data confidentiality and authentication. The support on the IPsec implementation of ESP is mandatory

and the support of AH is optional - a change from the previous specifications where AH was also mandatory. A list of the services provided by each protocol are shown in table 2.1.

Protocol	Authentication	Integrity	Confidentiality	Anti-Replay
AH	Yes	Yes	No	Yes
ESP	Optional	Optional	Yes	Yes

Table 2.1: Services provided by ESP and AH.

These two protocols can be applied individually or in combination. Note that ESP can itself provide all security services. Both protocols can be used on two distinct modes: **Transport mode** and **Tunnel mode**. The transport mode is usually used on end-to-end communications, and thus ESP and AH provide protection for upper layer protocols. The tunnel mode is used on network-to-network communications, like VPNs. Because in tunnel mode the IP packets are encapsulated within another IP packet, the ESP and AH offer all protections to the Network layer.

2.1.2 Security Associations

A Security Association (SA) is a one-way communication session between hosts that allows the interchange of security services to the traffic carried by it. AH and ESP requires that both end nodes agree on a key, on the algorithm used for authentication and/or encryption, among other parameters. Theses parameters are stored in an SA database (SAD) in both users. Using SA allows decoupling the key management from the security mechanisms.

Since an SA is a simplex channel, a communication session between two hosts will require two SA, one for each direction. Each SA has a Security Parameter Index, a number that allows hosts to identify the current SA in use. The sender chooses the SA based on the source and destination addresses and sends along with the packet the correspondent SPI. The receiver uses the latter to locate the appropriate SA.

Security services are associated to an SA by the use of AH or ESP but not both. If both AH and ESP protection are applied to a traffic stream, then two SAs must be created.

Listing 2.1 shows a typical SAD entry.

```
add 172.16.2.54 172.16.3.51 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.54 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
```

Listing 2.1: Security Association database entry.

2.1.3 Security Policy Database

Security Associations are used to enforce security policy for traffic crossing the IPsec boundary. Thus, it is necessary to exist a database specifying what security services are to be offered to IP datagrams and in what fashion. The Security Policy Database (SPD) was elaborated to hold that information. This information comprehends the protocol to be used, ESP or AH, the mode to be used, Transport or Tunnel, to which source and destination hosts or networks are to be applied, and other data. A typical SPD entry is shown in Listing 2.2.

```
spdadd 192.168.1.0/24 192.168.1.0/24
   any -P in ipsec esp/tunnel/172.16.2.54-172.16.3.51/require;
spdadd 192.168.1.0/24 192.168.1.0/24
   any -P out ipsec esp/tunnel/172.16.3.51-172.16.2.54/require;
```

Listing 2.2: Security Association database entry.

2.1.4 IPsec Modes

IPsec uses two distinct modes for establishing secure communication between end hosts: transport and tunnel mode. These dictate which specific parts of the IP datagram are to be protected and consequently how IPsec protocols must be arranged to achieve this goal. IPsec modes therefore establish the basis of a security association (SA), an abstraction which contains all information relative to the negotiated security scheme.

Transport Mode

Transport mode is intended for secure host-to-host connections, allowing payload encryption and packet authentication. Despite being visible to all intermediate nodes, immutable fields in the IP header may not be modified since their integrity is verified by calculating the corresponding hash value.

Immutable fields include the version, header length, packet length, next protocol, identification, source IP and destination IP parameters. Transport mode, when used with authentication, is therefore severely limited on the global Internet, since the proliferation of middle-boxes, in particular those implementing NAT (network address translation) corrupt the calculated hash value upon modifying the IP header.

Figure 2.1: IPsec in Transport Mode

Tunnel Mode

Tunnel mode ensures the protection of the complete IP packet, through the use of encapsulation. This permits the use of differentiated policies for the original IP packet, which will constitute the payload, and the new IP header. This is particularly useful in establishing encrypted tunnels, such as those used in virtual private networks (VPN), where packets are encrypted prior to encapsulation. As with transport mode, the use of authentication raises NAT traversal issues.

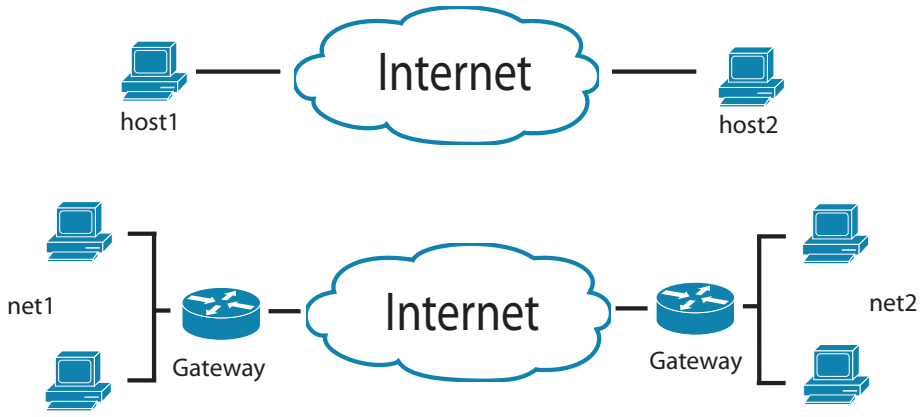


Figure 2.2: IPsec in Tunnel Mode

In tunnel mode however, authentication of the outer header may be a less pressing concern since the protection of the original packet is assured. As such, tunnel mode is particularly well suited for network-to-network and host-to-network connections.

2.1.5 AH - Authentication Header

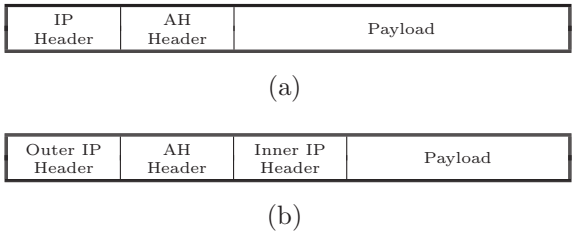
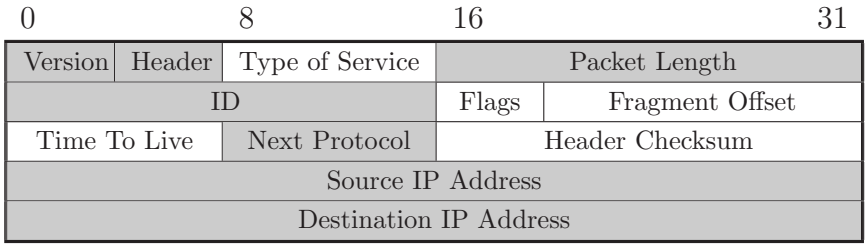


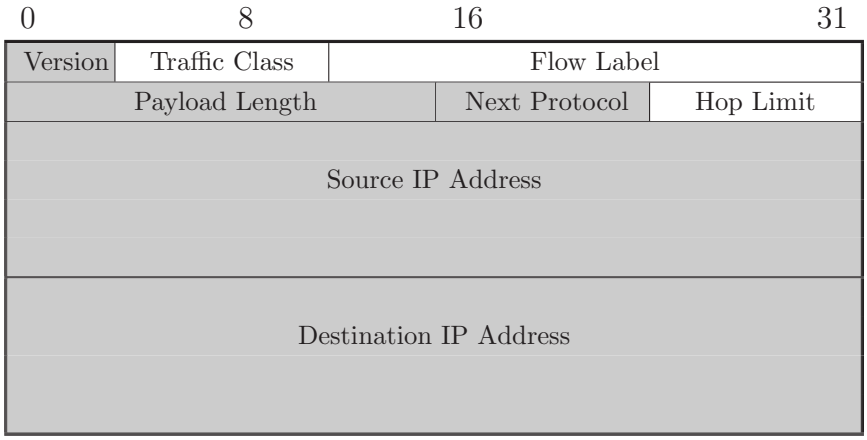
Figure 2.3: AH in (a) transport mode and (b) tunnel mode.

The authentication header (AH) provides replay protection and data integrity and origin authentication without encryption. The AH authentication comprehend the complete IP payload as well as the immutable fields of the IP Header and IPv6 extensions headers, both in transport and tunnel modes. Although in tunnel mode the IP payload is composed of the original IP packet, the authentication header does not function differently: AH authenticates the original IP packet and the immutable fields of the outer IP header and outer extension headers, if any. The authentication header is inserted between the IP header and the appropriate transport protocol or encapsulated payload, as shown in figure 2.3. Figure 2.4 shows the IP parameters authenticated by AH for both IPv4 and IPv6 headers.

As stated before, AH can be used alone or in conjunction with ESP. ESP may be used to provide the same anti-replay and similar integrity services, and it also provides a confidentiality (encryption) service. The primary difference between the integrity provided by ESP and AH is the extent of the coverage. Specifically, ESP does not protect any IP header fields unless those fields are encapsulated by ESP in Tunnel mode.



(a)



(b)

Figure 2.4: IPv4 (a) and IPv6 (b) headers, with highlighted fields protected by authentication.

AH Format

The AH header format is illustrated in Figure 2.5. This format is the same for both IP versions.

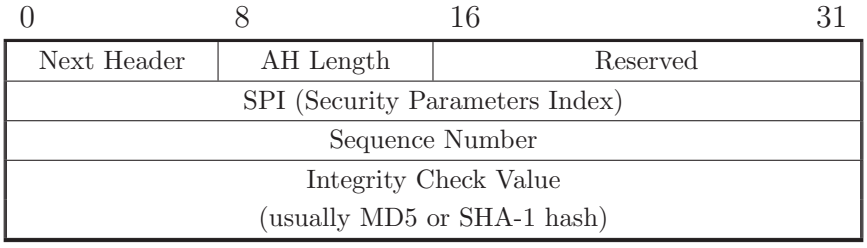


Figure 2.5: AH header.

A description of each field is made as follows:

- **Next Header** 8-bit field that identifies the protocol which immediately follows AH. The value of each protocol is established on the IP Protocol Numbers defined by IANA.
- **Payload Length** 8-bit field indicating the length of the AH. The value comes in 32-bit words minus 2. In IPv6 packets, the total length must be aligned on 8-byte units.
- **Reserved** 16-bit value reserved for future use that must be set to zero.
- **Security Parameters Index (SPI)** 32-bit value specified by the sender in order to allow the receiver identify the chosen security scheme. The values in range of 1 through 255 are reserved by IANA for future use.
- **Sequence Number** 32-bit field containing a monotonically increasing number for each packet sent. This value provides replay protection. The sequence number is initialized at zero in both sender and receiver counters and must not be allowed to cycle in the same SA. The sender and the receiver must establish a new SA prior the overflow of the field. A 64-bit version can be used by both parties (Extended Sequence Number). In order to reduce packet overhead, only the 32 less significant bits of the sequence number are set in this field thus maintaining the original length.

– **Integrity Check Value (ICV)** This is a variable-length field containing computed authentication signature. The field must be a multiple of 32-bit and may contain pad bits in order to the AH be aligned to 64-bit for IPv6 packets.

Calculating the Integrity Check Value

ICV is obtained by applying an integrity algorithm upon the packet. These algorithms include Message Authentication Codes (MACs) based on symmetric encryption algorithms like AES[U.S. FIPS PUB 197] or one way hash functions like MD5, SHA-1, SHA-256 and others.

2.1.6 Encapsulating Security Payload

Encapsulating Security Payload (ESP) was designed to provide data confidentiality, data origin authentication, data integrity assurance mechanisms and anti-replay protection. Associated security schemes are negotiated on establishing an IPsec SA, and may be provided over both transport and tunnel mode, as shown in 2.1.6. The ESP header is inserted between the IP header and the IP payload, as with AH, and add two additional areas to the payload: the ESP trailer, which is encrypted with the payload, and the ESP authentication field. In schemes were both ESP and AH are used, both are interchangeable. Such a scheme may make sense since ESP, unlike AH, provides no IP header authentication. Using encryption-only for confidentiality is possible but not recommended because it will only offer protection to passive attacks.

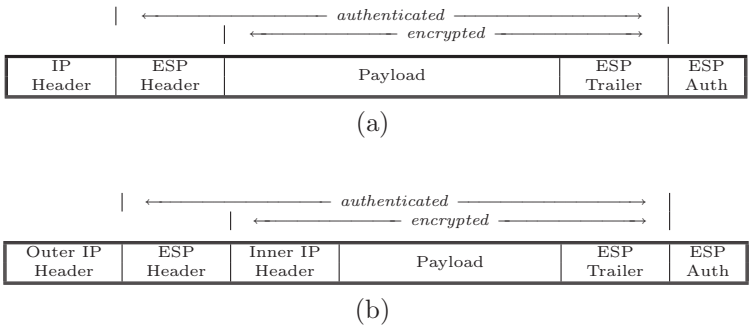


Figure 2.6: ESP in (a) transport mode and (b) tunnel mode.

Figure 2.6b, which represents the use of ESP in tunnel mode, illustrates traffic-flow confidentiality, a key feature provided by IPsec, in which

packet encryption and subsequent encapsulation provides complete privacy. Encryption is provided using symmetric-key algorithms such as DES [9], 3DES [10] and AES [11]. ESP allows data authentication based on a Keyed-HMAC (hash message authentication code) included in the ESP header. The HMAC is calculated using authentication algorithms supported by IPsec, such as MD5 [12] and SHA-1 [13].

ESP Format

The ESP header format is illustrated in figure 2.7. This format is the same for both IP versions.

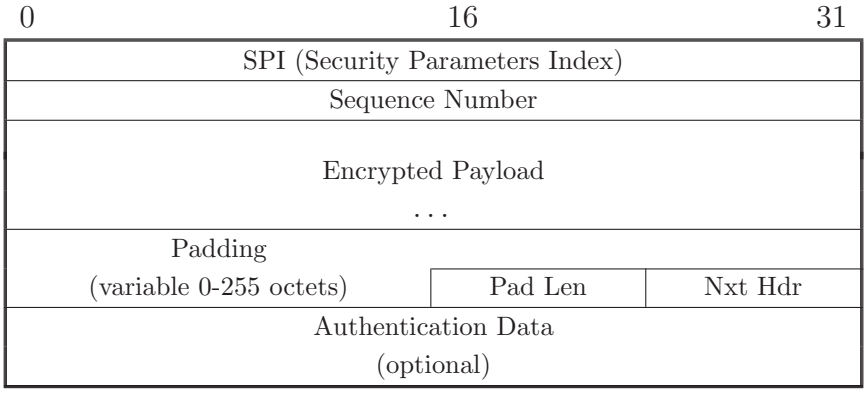


Figure 2.7: ESP Header and Trailer.

A description of each field is made as follows:

– **Security Parameters Index (SPI)** 32-bit value specified by the sender in order to allow the receiver identify the chosen security scheme. The values in range of 1 through 255 are reserved by IANA for future use.

– **Sequence Number** 32-bit field containing a monotonically increasing number for each packet sent. This value provides replay protection. The sequence number is initialized at zero in both sender and receiver counters and must not be allowed to cycle in the same SA. The sender and the receiver must establish a new SA prior the overflow of the field. A 64-bit version can be used by both parties (Extended Sequence Number). In order to reduce packet overhead,

only the 32 less significant bits of the sequence number are set in this field thus maintaining the original length.

- **Encrypted Payload** is a variable-length field containing data (from the original IP packet) described by the Next Header field. The Payload Data field is mandatory and is an integral number of bytes in length. Note that the beginning of the next layer protocol header must be aligned relative to the beginning of the ESP header as follows. For IPv4, this alignment is a multiple of 4 bytes. For IPv6, the alignment is a multiple of 8 bytes.

- **Padding** This field fills to purposes: If an encryption algorithm is employed that requires the plaintext to be a multiple of some number of bytes the Padding field is used to fill the plaintext (consisting of the Payload Data, Padding, Pad Length, and Next Header fields) to the size required by the algorithm; Padding also may be required, irrespective of encryption algorithm requirements, to ensure that the resulting ciphertext terminates on a 4-byte boundary. Specifically, the Pad Length and Next Header fields must be right aligned within a 4-byte word, as illustrated in the ESP packet format figures above, to ensure that the ICV field (if present) is aligned on a 4-byte boundary.

- **Pad Length** The Pad Length field indicates the number of pad bytes immediately preceding it in the Padding field. The range of valid values is 0 to 255, where a value of zero indicates that no Padding bytes are present. The Pad Length field is mandatory.

- **Next Header** 8-bit field that identifies the protocol which immediately follows AH. The value of each protocol is established on the IP Protocol Numbers defined by IANA.

- **Integrity Check Value (ICV)** This is a variable-length field containing computed authentication signature. The field must be a multiple of 32-bit and may contain pad bits in order to the AH be aligned to 64-bit for IPv6 packets.

2.1.7 Mandatory and recommended cryptographic algorithms

In order to guarantee interoperability between distinct implementations of the IPSec, some mandatory-to-support algorithms were specified [RFC4305]. Optional but recommended algorithms were also specified.

ESP encryption algorithms

Mandatory

– NULL Algorithm

– TripleDES-CBC

Recommended

– AES-CBC with 128-bit keys

– AES-CTR

ESP authentication algorithms

Mandatory

– HMAC-SHA1-96

– NULL Algorithm

Recommended

– AES-XCBC-MAC-96

– HMAC-MD5-96

Note that NULL algorithm in practice does not actually encrypts or authenticates data. One of the uses of the Null algorithm is for debug purposes.

AH authentication algorithms

Mandatory

– HMAC-SHA1-96

Recommended

– AES-XCBC-MAC-96

– HMAC-MD5-96

2.2 XCP - eXplicit Control Protocol

TCP is the standard end-to-end transport protocol of the Internet. The congestion control mechanisms introduced by the former are fundamental to maintain an high performance and stable network operation. As the Internet evolves and the high bandwidth and latency networks rise like Gigabit links and lossy wireless links, the current TCP congestion control algorithms begun to reach its limits, since TCP reacts adversely to increases in bandwidth or delay. Also, the growth of real-time distributions, introduces other transport protocols that contribute to the network congestion.

Since TCP has no explicit feedback from the network, TCP is limited to estimate congestion from packet loss. TCP responds conservatively to packet losses because of the AIMD (additive increase, multiplicative decrease) paradigm of the control algorithms. This scenario is not suited for lossy wireless technologies like UMTS, WLAN, DVB and Bluetooth, since the decrease policy is aggressive while the increase policy is conservative. A new approach is necessary to maintain the stability and optimization of the Internet.

2.2.1 Protocol Overview

The introduction of explicit congestion control protocols like the eXplicit Control Protocol represents a major advance in Internet congestion control. XCP can deliver the highest possible application performance in extremely high speed and high latency network links that TCP does not operate well. Doing so, XCP achieves maximum link utilization without bandwidth wastes due packet loss.

XCP foresee three types of participants: the sender hosts, the receiver hosts, and the intermediate nodes in which packet queuing occurs (typically routers). The senders specify their desired throughput in the outgoing packets and routers along the path adjust the value to reflect both fairness and available bandwidth. At the end of the path, the receiver will send back the value of the resulting throughput and the sender will adjust its congestion windows accordingly.

Figure 2.8 illustrates a possible XCP connection. The sender requests a increase of η_1 to the current congestion window, by putting the value in the XCP header. Router R_1 reads and analyses the value from the header and forwards the packet without changing it since it has enough capacity to cope with the requested increase in the flow's throughput by the sender.

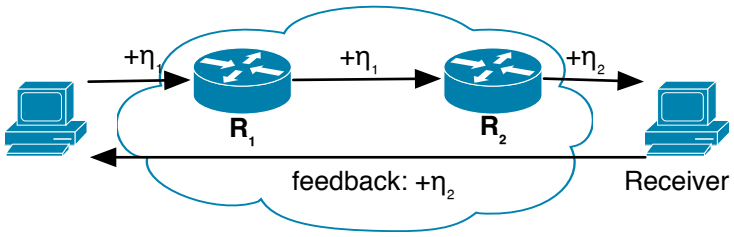


Figure 2.8: XCP flow path with corresponding feedback.

On the other hand, Router R_2 has no enough bandwith to answer to the requested value and replaces η_1 by a lower value, η_2, the throughput that R_2 can afford to spare to the flow. Finally, the receiver copies the final value, η_2 and sends it back to the origin as feedback. The original sender reads the resulting value and adjusts his congestion window accordingly.

2.2.2 Congestion Header

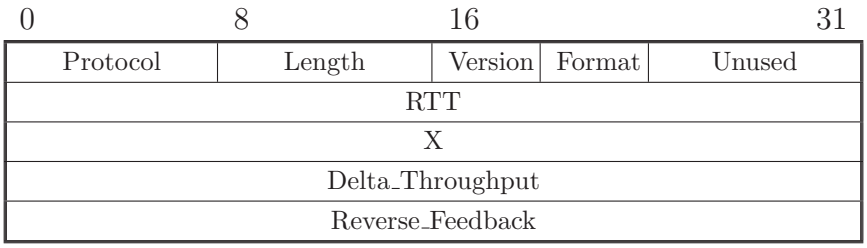


Figure 2.9: XCP congestion header, version 2.

Each field is explained as follows:

- **Protocol** indicates the next-level protocol used in the data portion of the packet. The values for various protocols are specified by IANA.
- **Length** indicates the length of the congestion header, measured in bytes. Length in this version of XCP will always be 20 bytes, or 0x14.
- **Version** indicates the version of XCP that is in use. The current format version is 0x02.

- **Format** indicates header format. Currently only the Standard Format and Minimal Format are defined.
- **unused** This field is unused and must be set to zero in the current version of XCP.
- **RTT** indicates the round-trip time measured by the sender, in fixed point format with 28 bits after the binary point, in seconds. This field is not used in Minimal Format.
- **X** indicates the inter-packet time of the flow as calculated by the sender, in fixed point format with 28 bits after the binary point, in seconds. This is the same format as used by the RTT field. Also not used in the Minimal Format.
- **Delta_Throughput** indicates the desired or allocated change in throughput. It is set by the sender to indicate the amount by which the sender would like to adjust its throughput, and it may be subsequently reduced by routers along the path. Also not used in Minimal Format.
- **Reverse_Feedback** indicates the value of Delta_Throughput received by the data receiver. The receiver copies the field Delta_Throughput into the Reverse_Feedback field of the next outgoing packet in the same flow.

A more recent XCP draft specifies a rearrangement to the Congestion Header fields in order to optimize performance on FCPGA implementation. However, only the version 2 is avaiable at ISI's XCP website.

2.2.3 End-System Functions

The sender must signal desired changes in throughput by calculating the appropriate *Delta_Throughput* value. This value reflects the per-packet distribution of the throughput change and is required in order to maintain XCP routers oblivious to per-flow congestion states. *Delta_Throughput* is obtained by calculating the difference between the desired and estimated throughput, which represents the appropriate change, and dividing the result by the number of packets in one round-trip time. The latter may be estimated by dividing the current throughput by the maximum segment size, thus obtaining an estimate for the current throughput in packets, and multiplying the resulting value by the round-trip time. The resulting equation is shown below (2.1):

$$Delta\_Throughput = \frac{t_p - t_a}{t_a \cdot \left(\frac{RTT}{MSS}\right)} \tag{2.1}$$

where t_p is the desired throughput, t_a is the current throughput, RTT is the current round-trip time estimate and MSS is the maximum segment size of outgoing packets. Additionally, the sender must set their current estimate of both RTT and X, the inter-packet time. X may be estimated by dividing the round-trip time by the number of outstanding packets or, alternatively, by obtaining the ratio between packet size and the current throughput.

The receiver simply copies the received *Delta_Throughput* value into the *Reverse_Feedback* field of outgoing packets, thus bringing closure to the system's feedback loop.

Upon receiving the feedback value, the sender adjusts it's output rate accordingly. Under TCP, this adjustment is achieved by modulating the congestion window, *cwnd*, as described by equation (2.2):

$$cwnd = max\left(cwnd + Reverse\_Feedback \cdot RTT, MSS\right) \tag{2.2}$$

A minimum value of MSS is required to avoid the "Silly Window Syndrome" [14].

2.2.4 Router Functions

Despite being an integral part of XCP end hosts perform few functions: the sender requests a change in throughput, while receivers merely return the resulting throughput as feedback. The core of XCP lies in the routers, where the de-coupling of utilization control from fairness control allows the system to converge to optimal efficiency.

Packet Arrival

On arrival at a router, the data contained in the packet's congestion header is used to update parameters used for further calculations. The router must update the total amount of incoming data, *input_traffic*, by incrementing the current value with the incoming packet's size in bytes. Additionally, the router must maintain an accurate estimate of the average RTT, d, across all flows, without maintaining per-flow state. This is achieved using (2.3):

$$d = \frac{\sum\left(X \cdot RTT\right)}{\sum X} \tag{2.3}$$

Consequently, the router must maintain both the summed total of incoming values of the inter-packet time X and the product of X with the corresponding RTT.

Utilization control

Utilization is controlled by adjusting the aggressiveness according to the spare bandwidth and the feedback delay. To this end, the router must periodically calculate the fair capacity at regular control intervals. In the current draft specification of XCP, this interval is defined as being the average RTT, d, across all flows. The bandwidth F available for distribution amongst the flows is given by:

$$F = \alpha \cdot (C - input\_bw) - \beta \cdot \frac{q}{d} \tag{2.4}$$

where C is the capacity of the link, *input_bw* is the input bandwidth since the last control interval and q is the persistent queue. α and β are constants which guarantee system stability. This calculation ensures efficiency by making maximal use of the available link capacity whilst simultaneously draining the current queue.

Fairness control

Fairness is ensured by reclaiming and reallocating bandwidth from flows with rates above their fair share. This requires the redistribution of previously allocated bandwidth by performing bandwidth shuffling, thereby certifying that new flows are attributed bandwidth even when the system is stable (F=0). The shuffling function used in the current implementation of XCP is presented in (2.5).

$$S_T = max\,(0, \gamma \cdot input\_bw - |F|) \tag{2.5}$$

where S_T represents the total bandwidth to be shuffled in the current control interval and $\gamma = 0.1$.

The fairness algorithm differentiates the total pool of capacity to be distributed between positive residue feedback, R_p, and the negative residue feedback, R_n.

$$R_p = S_T + max\,(F, 0) \tag{2.6}$$

$$R_n = S_T + max\,(-F, 0) \tag{2.7}$$

The final calculation carried out during the control interval timeout prepares the residue for usage on a per-packet basis. Since positive feedback is applied equally per-flow, the positive feedback scale factor, C_p, takes the positive residue feedback and divides it by the total sum of inter-packet time over the last control interval. Negative feedback, however, is proportional to capacity, therefore having a greater effect on flows occupying the most bandwidth. The negative feedback scale factor results from the division of the negative residue feedback by the total input traffic.

$$C_p = \frac{R_p}{\sum X} \tag{2.8}$$

$$C_n = \frac{R_n}{input\_traffic} \tag{2.9}$$

Before returning, the control interval timeout must schedule a new control interval in d seconds, using a newly calculated average RTT. Statistics collected during the control interval must also be reset.

Packet Departure

On packet departure, the router must compare the packet's *Delta_Throughput* value with the locally available capacity. To calculate this capacity, the positive and negative feedback associated to a packet must be known. The positive feedback, F_p, is calculated by multiplying the current estimate of the positive feedback scale factor C_p by the packet's declared inter-packet time X, thus allowing for fair per-flow distribution.

$$F_p = C_n \cdot X \tag{2.10}$$

Similarly, the negative feedback, F_n, is calculated by multiplying the current estimate of the negative feedback scale factor C_n by the packet's size, resulting in fairness by levelling the capacity attributed to each flow.

$$F_n = C_p \cdot Pkt\_size \tag{2.11}$$

The total feedback F_t which may be conceded to a packet may therefore be calculated as the difference between the packet's respective positive and negative feedback values:

$$F_t = F_p - F_n \tag{2.12}$$

Should F_t be lower than the packet's *Delta_Throughput*, *Delta_Throughput* is replaced with F_t before packet departure, otherwise the outgoing packet's congestion header remains unchanged.

2.3 TCP/IP and IPsec implementation in FreeBSD

The TCP/IP stack implementation in FreeBSD is heavily based on the reference 4.4 BSD TCP/IP protocol stack, the Net/3 [15]. FreeBSD is used by many major internet sites such Yahoo! and Verio [16] and is known to have one of the most robust and stable TCP/IP stack in existence [17] and also for its capability to run on modest equipment whilst able of high performance and serve thousands of connections.

The networking source code is organized in the standard UNIX fashion. The code referring to the IPv4 implementation is mostly concentrated in the folder */usr/src/sys/netinet/* while the IPv6 implementation lies on the folder */usr/src/sys/netinet*6/. Also, the source code for XCP is located in *sys/netinet/* and is common to both IP versions. Finally, the IPsec code lies on the folder *sys/netinet*6, since its support is mandatory for IPv6. The remaining networking code is scattered over other folders like *sys/kern*, *sys/sys* and *sys/net*.

Table 2.2 shows a list of the most important source files referring to IP, XCP, and IPsec with a brief descriptions of them.

2.3.1 Memory Buffers

One of the most important concepts of the FreeBSD stack are the Memory Buffers or MBUFs. These buffers are an essential piece in the entire TCP/IP implementation. The interoperability between the several protocol layers requires an easy manipulation of data made by the functions that compose those layers. Prepending, appending and removing data are constant needs when either a packet is being encapsulated from the upper layers to the lower layers or headers are removed as the packet crosses up the protocol stack. The MBUFs were made to support those tasks in a very efficient way. The most important features of MBUFs are as follows:

- **Fixed size** The size of the MBUF is always equal. The value is defined in the kernel source and the default size is 256 bytes. Part of the size comprises the MBUF header and the remaining the user data. The actual size of each section depends on the type of the MBUF. A more detailed description of the MBUF header and the various types will be made ahead.

- **Allows zero length records** MBUF size can be null which may be required by some protocols like UDP.

File	Description
netinet/xcp_var.h	XCP structures, macros and definitions
netinet/xcp.c	XCP input and output functions
netinet/tcp_var.h	TCP structures, macros and definitions
netinet/tcp_input.c	TCP input functions
netinet/tcp_output.c	TCP output functions
netinet6/tcp6_var.h	IPv6 specific TCP structures, macros and definitions
netinet/ip.h	IP structures, macros and definitions
netinet/ip6.h	IPv6 specific definitions
netinet6/ip_input.c	IPv4 input functions
netinet6/ip_output.c	IPv4 output functions
netinet6/ip6_input.c	IPv6 input functions
netinet6/ip6_output.c	IPv6 output functions
netinet6/ipsec.h	IPsec structures, macros and definitions
netinet6/ipsec6.h	IPv6 specific IPsec definitions
netinet6/ipsec.c	General IPsec functions
netinet6/esp_input.c	ESP input functions
netinet6/esp_output.c	ESP output functions
netinet6/ah_input.c	AH input functions
netinet6/ah_output.c	AH output functions
netinet6/ah_core.c	AH checksum functions

Table 2.2: Most important TCP/IP and IPsec source files

– **Pointer to the beginning of the user data** The user data can start at any place of the buffer. This facility allows an easy prepending of data. The upper layers can leave enough room in front of the buffer and the lower can simply copy their data and update the pointer.

– **Two linked lists** Since a packet may have more than 256 bytes in size, several MBUF can be allocated to store the packet. The header of the MBUF has a pointer to the next MBUF making this way a linked list of MBUFs. The other linked list is a pointer to the head MBUF chain of the next packet.

– **External MBUFs** A MBUF may have a buffer cluster attached to it. The size of the cluster is usually bigger than the size of an MBUF and the default value is 2048 bytes. This is convenient for large packets since otherwise it would require an elevated allocation of small-sized MBUFs.

MBUF types

Figure 2.10 shows the four different types of MBUFs. The m_flags field is set accordantly to the type of the MBUF.

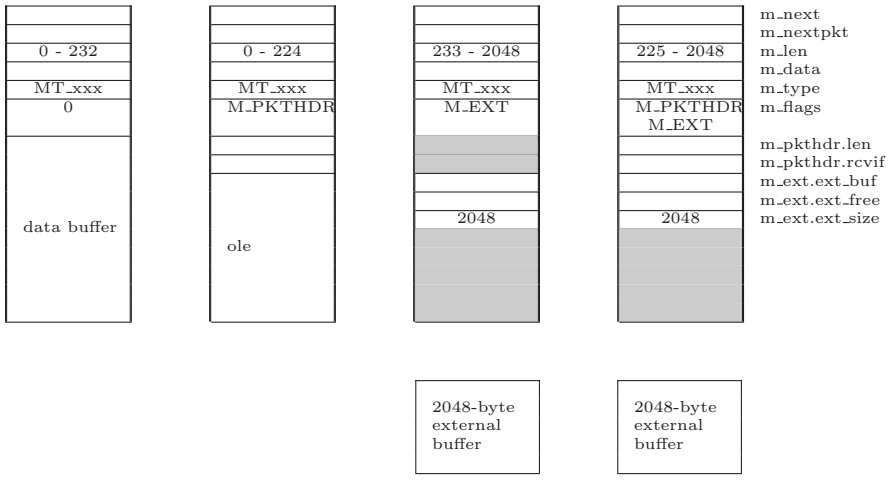


Figure 2.10: The four types of MBUFs.

The shadow fields are note used. Each field is explained as follows:

- *m_next* Pointer to the next MBUF of the packet chain. Its value is null if the packet fits on a single MBUF.
- *m_nextpkt* Pointer to the head MBUF of the next packet.
- *m_len* Size in bytes of the data in the MBUF.
- *m_data* Pointer to the beginning of the data on the MBUF.
- *m_type* Type of the data contained in the buffer.
- m_flags Flags referring to the type of MBUF. A regular MBUF like the first MBUF in the figure 2.10 has this value cleared. A head MBUF which corresponds to the start of a packet, will have the M_PKTHDR flag set. The third and fourth MBUFs have the flag M_EXT set, indicating that a cluster is attached to it being the latter a head MBUF.

MBUF macros and functions

There are many macros and functions to deal with MBUFs. These macros and functions constitutes an extent API which allows to perform several tasks like allocating an MBUF, aligning the data within a MBUF or pull up data. During the integration of XCP with IPsec this API was used extensively.

MGETHDR	MGETHDR(struct mbuf *m, int *nowait*, int *type*) Allocate an MBUF and initialize it as a packet header. The flag M_PKTHDR is set and the pointer *m_data* points to the beginning of the user data buffer.
MGET	MGET(struct mbuf *m, int *nowait*, int *type*) Allocates a regular MBUF. No flag is set and the the data pointer is set.
MH_ALIGN	MH_ALIGN(struct mbuf *m, int *len*) Set the *m_data* pointer of an mbuf containing a packet header to prvide room for an object of size *len* bytes at the end of the MBUF's data buffer.
M_PREPEND	M_PREPEND(struct mbuf *m, int *len*, int *nowait* Prepend *len* bytes of data in front of the data in the MBUF pointed to by m. If there is no enough room, a new MBUF is allocated.
mtod	*type* mtod(struct mbuf *m, *type*) Type cast the pointer to the data area of the MBUF pointed by m to *type*

Table 2.3: MBUF macros

Tables 2.3 and 2.4 show the most used macros and functions, respectively, during the integration.

m_pullup	struct mbuf *m_pullup(struct mbuf *m, int len) Rearrange the existing data in the MBUF chain point to by m so that the first len bytes of data are stored contiguously in the first MBUF in the chain.
m_copydata	void m_copydata(struct mbuf *m, int $offset$, int len, caddr_t cp) Copy len bytes of data from the MBUF chain pointed to by m into the buffer pointed to by cp
m_free	struct mbuf *m_free(struct mbuf *m) Free the single MBUF pointed to by m.

Table 2.4: MBUF functions

Chapter 3

Integrating XCP with IPsec (IPv4)

XCP integration with IPsec is a necessary step in the protocol's standarization.

The integration of XCP with IPsec brings advantages to each other. By taking into account network flows based on the IPsec security policies, explicit congestion control protocols gain a better perspective on how feedback must be distributed amongst participating end-hosts. Conversely, securing the congestion header avoids vulnerabilities in the feedback system estabilished by protocols like XCP, such as man-in-the-middle attacks manipulating data used by routers to adjust system performance, or corrupting feedback used to adjust an end-host's send rate.

The current implementation of XCP does not yet support IPsec connections for XCP flows. IPsec transforms are defined to be applied immediately after the IP Header. This either does not allow routers along the path to modify the congestion header due to packet authentication in AH mode, or hides the congestion header from the router altogether if encryption is used in ESP mode. Figure 3.1 shows the current scenario.

To solve this issue, IPsec and TCP/IP implementation must be modified in order to be XPC aware and properly insert the congestion header before both the authentication header and encryption header.

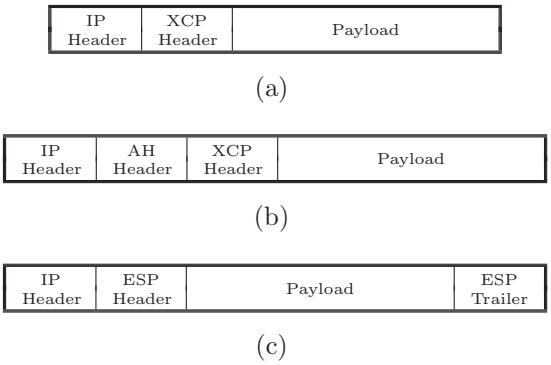


Figure 3.1: A (a) XCP packet and the same packet after (b) authentication, using AH, and (c) encryption, using ESP.

3.1 Current scenario

In this section is made a description of the current packet travel from the transport layer to network layer using IPsec and XCP in IPv4 connections. Also in this section, are identified the restrictions of using IPsec and XCP in IPv4 connections and is shown the implemented solution.

3.1.1 IPsec and XCP packet flow

Figure 3.2 shows the current sequence of function calls for both outgoing and incoming packets.

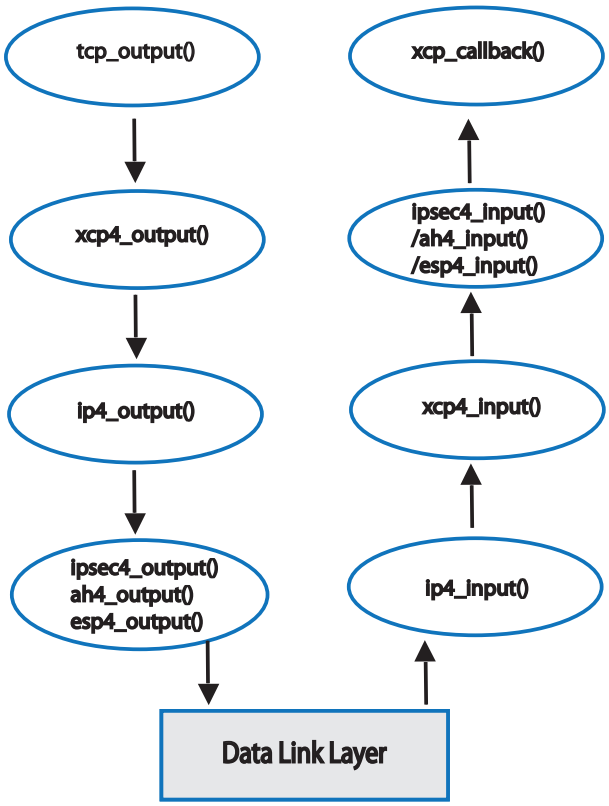


Figure 3.2: Function calls for outgoing and incoming IPv4 packets

Outgoing packets

1. When a user request of a TCP/XCP socket is made, the *tcp_output*(). Among other processings, this function is responsible for data segmentation and it size based on the path MTU. Note that for regular TCP sockets, the functions skips the call of XCP output and goes directly to IP output.

2. The *xcp*4_*output* function is responsible for creating and prepend the XCP header and initialize its values, including the remote feedback, if avaiable.

3. After being called by the XCP output, the *ip*4_*output*() function fills the IPv4 Header and checks for any security policy based on the destination address. If a policy should be enforced, the function calls the IPsec routines.

4. *ipsec*4_*output* checks the policies and calls the appropriate security protocol, ESP and/or AH. The output function of each protocol will perform transformations upon the packet. At the end, the packet returns to the ip output which will deliver it to the data link layer.

Incoming packets

1. The data link layer calls the IP input function. This function will do integrity checks to the incoming packet and afterwards will read from the Next Protocol field the IPv4 header and call the appropriate input function. Note that all input functions will return the IP function. The former must provide the next protocol value in order to the latter call the next input function.

2. The IPsec input functions will process the packet by verifying its authenticity and decrypt the payload.

3. The XCP function will remove and store the XCP header for later processing.

4. Finally the TCP input function will assemble the data and call again XCP to process the packet and determinate the proper throughput.

3.1.2 Current limitations and proposed solution

Because the IPsec output functions are called after the XCP processing, the XCP header will be included in the payload of the IP packet, which is

either authenticated or encrypted, not allowing the routers to change the header. To overcome this limitation the following solution is proposed:

- At the IPsec output functions level, treat the XCP header as an IPv4 Header option. Doing so, the XCP header will appear immediately after the IPv4 header and it would not be subject to encryption.
- Add support for the authentication of the new pseudo IPv4 header option. Since AH stills authenticates parts of the IPv4 header, the extra fields of XCP should be also authenticated.
- For tunnel mode, modifications to the TCP routine is necessary because of the duplication of the XCP header. As recommended by the XCP draft, the XCP header should be present in both inner and outer packets.

3.2 IPsec routines

3.2.1 Separating the congestion header from the IPsec payload

IPsec performs transformations after separating the outgoing packet in two: the IP header and the respective payload. Although the IP header remains unchanged, immutable fields may be authenticated and therefore network nodes along the path may not alter some header parameters. Since the packet payload may be completely authenticated or encrypted, the congestion header must be extracted from the payload before the chosen IPsec transform is applied.

Modifying this behaviour requires changes to *ipsec4_splithdr()*, the function belonging to *ipsec.c* responsible for separating the IPv4 header from the payload. This function is used before applying IPsec transforms and, when called:

1. Receives a pointer, *ptr*, to a complete IP packet through the function's argument and copies the pointer's value to an auxiliary pointer, *aux*.
2. Increments *aux* by the IP header length.
3. Checks for IP options and increments *aux* by the appropriate length.
4. If the packet header, located between *ptr* and *aux*, is not stored in the same *mbuf*, the packet is realigned to allow header allocation to an individual *mbuf*. This is a performance optimisation.
5. Return *aux*, which now points to payload.

The Listing 3.1 shows the added code to the function.

```
3221     nxt_proto = &ip->ip_p;
3222
3223 inspect_next_header:
3224     switch(*nxt_proto){
3225 #ifdef XCP
3226         case IPPROTO_XCP:
3227             xh = (struct xcphdr_x *) (mtod(m, caddr_t) + hlen);
3228             hlen += xh->cmn.len;
3229             nxt_proto = &xh->cmn.proto;
3230             goto inspect_next_header;
3231 #endif
3232 //      default: /*noop*/
3233     }
```

Listing 3.1: *netinet6/ipsec.c* detection and split of XCP header

3221 - 3233

Reads the current Next Protocol field from IP header. If next protocol is XCP, casts the packet, pointed to by *aux*, as a XCP header in order to extract the appropriate header length. This value is used to increment *aux*, which now points to the XCP, rather than IP, payload. An overview of this process and the resulting ESP packet is shown in figure 3.3.

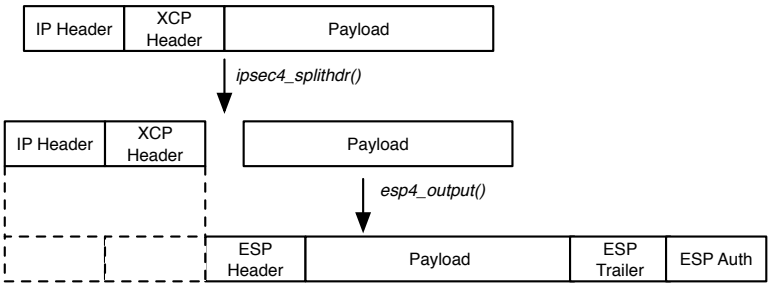


Figure 3.3: The IP and XCP headers are split from the payload, which is processed by ESP, and then concatenated with the encrypted payload to obtain the outgoing packet.

While this alone solves protocol layering, further modifications must be made to guarantee that IPsec transforms function properly with XCP.

3.2.2 Inserting XCP before ESP

Since ESP only deals with the payload, few modifications must be made for XCP integration. As with all IPsec transforms, ESP assumes it directly follows IP. This has direct implications: ESP modifies the IP header's next protocol field to point to itself without inspecting for end-to-network headers, resulting in a mismatch between the logical disposition of headers, as defined by the headers themselves, and the physical disposition of headers.

This is easily understood by analysing the reception of a packet with both XCP and ESP (figure 3.4). Once IP processing is completed, the networking stack invokes the function responsible for interpreting the next header, which IP declares to be ESP. Upon casting the XCP header as ESP, the function responsible for parsing the ESP header, *esp_input*(), quickly discards the packet as corrupt. Correcting this behaviour in IPv4 requires modifying *esp4_output*() in *esp_output.c* so that *esp_output*(), the function which executes the bulk of the transform process common to both IPv4

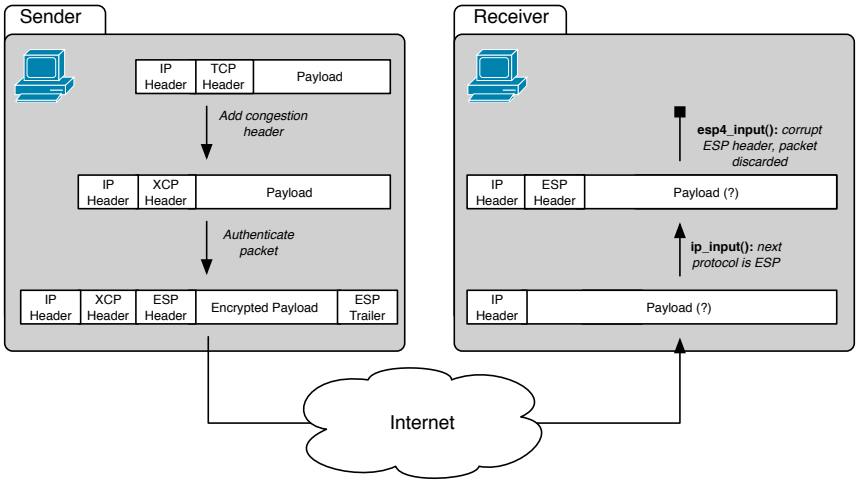


Figure 3.4: Congestion header being interpreted as ESP header.

and IPv6, changes the protocol field of the header immediately preceding the ESP. Listing 3.2 show the added code.

```
701     nxt_proto = &ip->ip_p;
702
703     //Loop through packet headers. If headers should not become part of payload,
704     //increment payload pointer with header length.
705     while(1){
706        switch(*nxt_proto){
707
708 #ifdef XCP      //for now, only XCP header is accounted for
709           case IPPROTO_XCP:
710              xh = (struct xcphdr_x *) (mtod(m,caddr_t) + (int)(ip->ip_hl<<2 ));
711              nxt_proto = &xh->cmn.proto;
712              break;
713 #endif
714
715           default:
716              /* XXX assumes that m->m_next points to payload */
717              return esp_output(m, nxt_proto, m->m_next, isr, AF_INET);
718        }
719     }
```

Listing 3.2: *netinet6/esp_output.c* skiping XCP header

701 - 712

If the current Next Protocol after IP header is XCP, change the pointer of the Next Protocol from the IP header to the XCP header. Doing so,

allows the ESP to write its protocol value to the XCP header instead of the IP header.

715 - 718
Calls the esp_output() with the correct pointer to the Next Protocol.

3.2.3 Inserting XCP before AH

As with ESP, outgoing packets processed by AH must be corrected in order to be correctly interpreted by the receiver, which may be achieved by altering *ah4_output*() to switch the protocol field from XCP to AH if a congestion header is present. Since AH authenticates the IP header however, *ah4_output*() is also responsible for calling the checksum calculation of both the IP header and the congestion header, since IPsec treats XCP as an IP option. This raises issues not encountered in ESP: we must guarantee that the outgoing congestion header, as processed by AH, is identical to the congestion header received by the end-host for which a security association has been established. As can be seen in the Listing 3.3 a similar loop is made to proper correct the point where the Next Protocol stands.

```
210     nxt_proto = &ip->ip_p;
211
212 inspect_next_header:
213     switch(*nxt_proto){
214 #ifdef XCP
215         case IPPROTO_XCP:
216             xh = (struct xcphdr_x *) (mtod(m, caddr_t) + hlen);
217             hlen += xh->cmn.len;
218             nxt_proto = &xh->cmn.proto;
219             goto inspect_next_header;
220 #endif
221 //      default: //noop;
```

Listing 3.3: *netinet6/ah_output.c* skiping XCP header

210 - 221
If the current Next Protocol after IP header is XCP, change the pointer of the Next Protocol from the IP header to the XCP header. Doing so, allows the AH to write its protocol value to the XCP header instead of the IP header.

Authenticating XCP

Treating XCP as an IP option, positioning the congestion header before IPsec transforms, has undesirable implications on header authentication. While XCP precedes IPsec in processing outgoing packets, incoming packets are processed by the order in which protocol headers are presented. The resulting mismatch in processing order is particularly worrying when dealing with authentication, since headers between IP and the transport protocol typically delete themselves after processing is completed.

This is illustrated in figure 3.5. The function responsible for processing the authentication header, *ah_input*, receives the packet with no XCP header, even though the header had been used to compute the hash value for the packet at the sender. Since the locally calculated hash value does not match the hash value contained in the packet, *ah_input* discards the packet as it may pose a security risk.

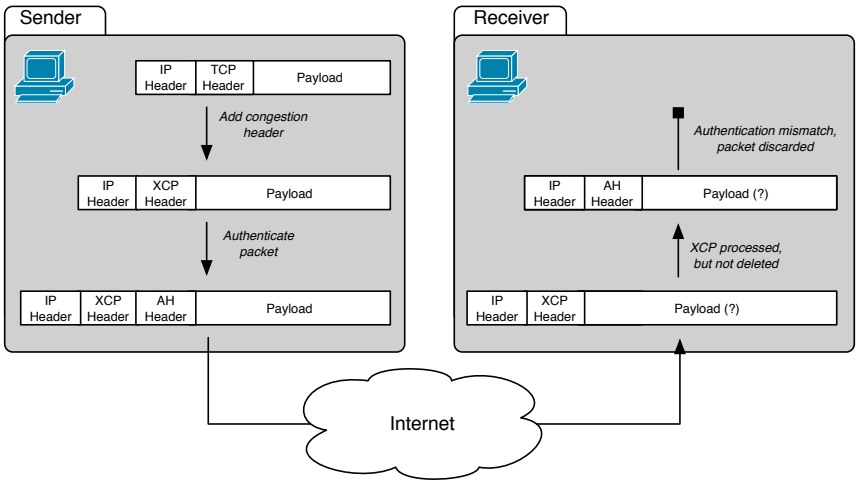


Figure 3.5: Current input processing of XCP/AH packets. The authenticated XCP header deletes itself before AH processing, therefore corrupting the packet.

Since the removed XCP is stored in a Tag, the AH checksum function can probe for its existence and recall the values in order to correctly compute the hash value. Also, is necessary to add support of authentication to the XCP header immutable fields.

First is necessary to correct some fields of the IP header that were modified by the removal of the XCP header. Listing 3.4 show the implemented

code.

```
1232#ifdef XCP
1233        /* check if XCP has already deleted from incoming mbuf
1234         * if so, correct next protocol field and packet length
1235         */
1236        tag = m_tag_find(m, PACKET_TAG_XCP, NULL);
1237        if(tag){
1238           xh = (struct xcphdr_x *)(tag+1);
1239           iphdr.ip_p = IPPROTO_XCP;
1240           iphdr.ip_len += htons(xh->cmn.len);
1241        }
1242
1243#endif
```

Listing 3.4: *netinet6/ah_core.c* correcting IP header

1232 - 1243

If a XCP Tag exists, change IP header Next Protocol field to the XCP value and increase the payload length in the size of the XCP header. This allows the checksum function to compute the correct value since that were the values at sender.

The code that adds support for XCP authentication, as well as capability to probe for the XCP Tag and callback the values if necessary, follows:

```
1352#ifdef XCP
1353    case IPPROTO_XCP:
1354        {
1355           struct xcphdr_x xh_immutable;
1356
1357           // if no tag exists, packet is outgoing
1358           if(!tag){
1359              xh = (struct xcphdr_x *) (mtod(m, caddr_t) + advancewidth);
1360              advancewidth = xh->cmn.len; //header is in mbuf
1361           } else {
1362              advancewidth = 0; //header not present.
1363           }
1364
1365           bcopy(xh, &xh_immutable, xh->cmn.len);
1366           //clear data
1367           xh_immutable.delta_tput=0;
1368           hdrtype = xh->cmn.proto;
1369           (algo->update)(&algos, (u_int8_t *)&xh_immutable, xh->cmn.len);
1370           break;
1371        }
1372#endif
```

Listing 3.5: *netinet6/ah_core.c* Tag detection and XCP authentication support

1352 - 1363

Defines the advance in bytes for the next loop cycle. If no tag exists, the packet is outgoing, and the header is present then the advance value is the XCP length.

1365 - 1370

Copies the XCP header to a temporary variable and sets the mutable field, *Delta_Throughput*, to zero and then calls the hash algorithm function.

3.3 Tunnel Mode

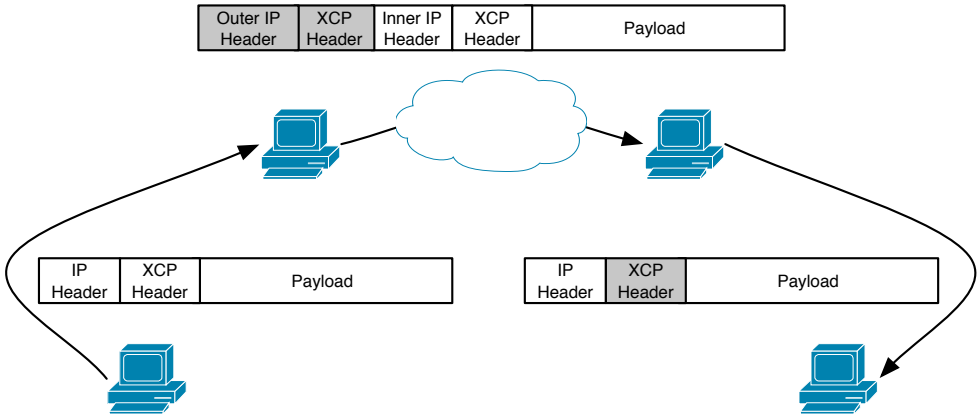


Figure 3.6: Tunneling XCP packets.

Modifications so far have focused on obtaining compability between XCP and IPsec transforms. While this covers explicit congestion control over transport mode, tunnel mode requires further changes in order to support XCP. Specifically, when a flow enters an IPsec tunnel the congestion header must be replicated to the outer IP header as is recommended by the XCP draft. Upon tunnel exit, the congestion header must be copied from the outer header into the inner header, ensuring that any changes to the *Delta_Throughput* value are taken into consideration.

3.3.1 IPsec routines

The first step taken was to ensure correct encapsulation of the XCP packet. The function responsible for IPv4 encapsulation, *ipsec4_encapsulate*() was altered in a similar fashion to *ipsec_splithdr*(), masquerading XCP as an IP option to ensure it remains attached to the IP header.

Replicating the outer XCP header to replace the inner header is more delicate since it requires probing the inner packet until the XCP header is obtained. The *Delta_Throughput* value of the outer XCP header is then copied to the inner header, as no other field is altered by routers. This additionally ensures that no inappropriate header modifications are made in an IP tunnel. Listing 3.6 shows the additional code.

```
377#ifdef XCP
378        if(tag){
379           xh = (struct xcphdr_x *) (mtod(m,caddr_t) + (int)(ip->ip_hl<<2 ));
380                      xh->delta_tput = xt->delta_tput;
381        }
382#endif
```

Listing 3.6: *netinet6/esp_input.c* Tag detection and XCP authentication support

If a XCP Tag exists, the *Delta_Throughput* value stored in the Tag is copied to the same field of the inner XCP header.

3.3.2 TCP Output

As described before, XCP operation in Tunnel mode foresees the presence of two headers, one in the encapsualted IP packet and other in the outer packet. This extra header represents an additional overhead that TCP Output should take into account in order to correctely calculate the maximum segmente size of each packet. Failing to do so will result in packet fragmentation at the Network Layer.

To avoid fragmentation, XCP Output must be modified to correctely adjust the segment size. Listing 3.7 shows the added code for TCP Output.

```
749    {
750       ip = mtod(m, struct ip *);
751       th = (struct tcphdr *)(ip + 1);
752       m->m_pkthdr.len = m->m_len = sizeof(struct tcpiphdr);
753       tcpip_fillheaders(inp, ip, th);
754
755       if (inp == NULL)
756          sp = ipsec4_getpolicybyaddr(m, dir, IP_FORWARDING, &error);
757       else
758          sp = ipsec4_getpolicybypcb(m, dir, inp, &error);
759
760       if (sp == NULL)
761          return 0;    /* XXX should be panic ? */
762    }
763
764    for (isr = sp->req; isr != NULL; isr = isr->next) {
765       if (isr->saidx.mode == IPSEC_MODE_TUNNEL) {
766          switch (((struct sockaddr *)&isr->saidx.dst)->sa_family) {
767          case AF_INET:
768            ipoptlen += (xcp_debug_header) ? XCPHDR_X_DBG_LEN:XCPHDR_X_LEN;
769             break;
770#ifdef INET6
771          case AF_INET6:
772            ipoptlen += (xcp_debug_header) ? XCPHDR_X_DBG_LEN:XCPHDR_X_LEN;
773             break;
774#endif
775          default:
```

```
776                 ipseclog((LOG_ERR, "ipsec_hdrsiz: "
777                     "unknown AF %d in IPsec tunnel SA\n",
778                     ((struct sockaddr *)&isr->saidx.dst)->sa_family));
779                 break;
780             }
781         }
782     }
```

Listing 3.7: *netinet6/tcp_output.c* maximum segment size

749 - 761

A temporary MBUF is allocated to hold a skeletal TCP/IP header. This header is used to get the IPsec policy that must be applied to the destination address.

764 - 782

Searches the policy structure for entries indicating the use of Tunnel mode. If a entry is found, the size of a XCP header is added to *ipoptlen* which holds the size of the extra data besides the TCP and the IP headers.

Chapter 4

Integrating XCP with IPsec (IPv6)

This chapter presents the implementation of IPsec with XCP for IPv6 networks. Since IPsec support is mandatory in IPv6 implementations, the awareness of end-to-network protocols becomes an even more important matter in order to provide explicit congestion control for the future networks without compromising security. The outline of the chapter follows the previous one. First is presented the current scenario for IPv6 flows using XCP and IPsec. Afterwards, the current issues are pointed and a proposed solution is shown. Finally in the sections 4.2 through 4.5 the implementation of the proposed solution is described.

4.1 Current scenario

In this section is made a description of the current packet travel from the transport layer to network layer using IPsec and XCP in IPv6 connections. Also in this section, are identified the restrictions of using IPsec and XCP and is shown the proposed solution.

4.1.1 IPsec and XCP packet flow

Figure 4.1 shows the current sequence of function calls for both outgoing and incoming packets.

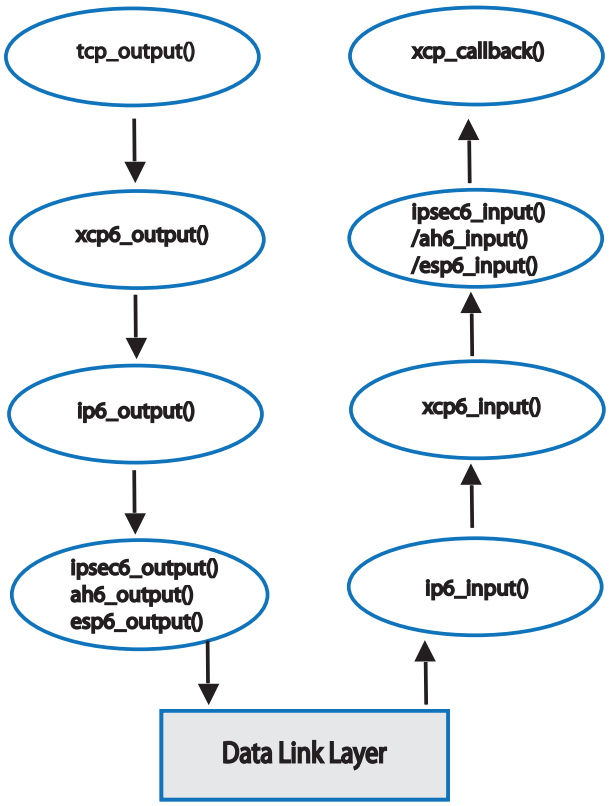


Figure 4.1: Function calls for outgoing and incoming IPv4 packets

Outgoing packets

1. When a user request of a TCP/XCP socket is made, the *tcp_output*(). Among other processings, this function is responsible for data segmentation and it size based on the path MTU. Note that for regular TCP sockets, the functions skips the call of XCP output and goes directly to IP output.

2. Like the IPv4 version, the *xcp*6*_output*() function is responsible for creating and prepend the XCP header and initialize its values, including the remote feedback, if avaiable.

3. After being called by the XCP output, the *ip*6*_output*() function fills the IPv6 Header and checks for any security policy based on the destination address. If a policy should be enforced, the function calls the IPsec routines. Also, this function creates the extension headers and prepend them to the IP packet.

4. *ipsec*6*_output* checks the policies and calls the appropriate security protocol, ESP and/or AH. The output function of each protocol will perform transformations upon the packet. At the end, the packet returns to the ip output which will deliver it to the data link layer.

Incoming packets

1. The data link layer calls the IP input function. This function will do integrity checks to the incoming packet and afterwards will read from the Next Protocol field the IPv4 header and call the appropriate input function. Note that all input functions will return the IP function. The former must provide the next protocol value in order to the latter call the next input function.

2. The IPsec input functions will process the packet by verifying its authenticity and decrypt the payload.

3. The XCP function will remove and store the XCP header for later processing.

4. Finally the TCP input function will assemble the data and call again XCP to process the packet and determinate the proper throughput.

4.1.2 Current limitations and proposed solution

The IPv6 stack faces the same problems of the IPv4 stack: the IPsec output function are called after the XCP processing and thus the latter will be considered payload data. On the other hand, IPv6 offers other possibilities to address this issue other than the aproach made for IPv4.

Since the IPv6 implementation forsees the mandatory support of IPsec, the stack is designed from base to offer the security servicies. Also, IPv6 has Extension Headers to the main header. This differ from the original IP header options since now each extension header is independent from the others and has its own output and input routines. The IPsc implementation is done in a fashion that allows the extension header to be skipped from encryptation or have selective authentication. Based on this elements, the following solution is proposed:

- Define a new IPv6 extension header. This new extension header would correspond to the actual XCP header.
- At the IP Output level, separete the XCP header from the payload and handle it as a regular extension header. Doing so, will allow the XCP header to skip encryption without having to modify the IPsec ESP routines.
- Add support for the authentication of the new extension header by defining the immutable fields in the AH routines.
- For tunnel mode, modifications to the TCP routine is necessary because of the duplication of the XCP header. As recommended by the XCP draft, the XCP header should be present in both inner and outer packets.

4.2 IP Output routines

4.2.1 Defining a new IPv6 Extension Header

The first step is to define a new IPv6 extension header. The structure *ip6_exthdr* defined in the file *ip6_output.c* contains a pointer for each extension header and is added a new pointer to hold the XCP header as shown in the Listing 4.1.

```
120 struct ip6_exthdrs {
121     struct mbuf *ip6e_ip6;
122     struct mbuf *ip6e_hbh;
123     struct mbuf *ip6e_dest1;
124     struct mbuf *ip6e_rthdr;
125     struct mbuf *ip6e_dest2;
126 #ifdef XCP
127     struct mbuf *ip6e_xch;          /* XCP Header */
128 #endif
129 };
```

Listing 4.1: *netinet6/ip6_output.c* additional IPv6 extension header

4.2.2 Detecting and separating the XCP Header

Afterwards, the XCP header must be removed from the payload and put in an additional MBUF, like an ordinary IPv6 extension header. The *ip6_output*() function is responsable to assemble the IP packet and send it to the network layer. Listings 4.2 and 4.2 shows the added code to cope with the XCP header.

```
231 #ifdef XCP
232     if (ip6->ip6_nxt == IPPROTO_XCP) {
233         struct xcphdr_x *xh;
234         struct mbuf *mx;   /* The new mbuf to hold XCP header,
235                               like an ordinary IPv6 extension header */
236
237         if (m->m_len < sizeof(struct ip6_hdr) + sizeof(struct xcphdr_x)) {
238             if ((m = m_pullup(m, sizeof(struct ip6_hdr) +
239                     sizeof(struct xcphdr_x))) == NULL) {
240                 error = ENOBUFS;
241                 goto freehdrs;
242             }
243         }
244
245         ip6 = mtod(m, struct ip6_hdr *);
246         xh = (struct xcphdr_x *) (mtod(m, struct ip6_hdr *) + 1);
247         hlen = xh->cmn.len;
248         /* just in case */
249         if (m->m_len < sizeof(struct ip6_hdr) + hlen) {
250             error = ENOBUFS;
```

```
251             goto freehdrs;
252         }
```

Listing 4.2: *netinet6/ip6_output.c* detection of XCP header

The description of the code is made as follows:

232 - 243

If the next protocol field in the IP header is XCP then a sanity check is made on the packet to ensure that its size has at least the size of an IPv6 header amd a XCP header. An error is returned otherwise.

245 - 252

After the above check, the length of the XCP header is read from the appropriate field, *cmn.len*, and stored. Another check is made to ensure that the size of the header matches the read value.

```
253         /* the new mbuf */
254         MGET(mx, M_DONTWAIT, MT_DATA);
255 
256         if (!mx)
257             return (ENOBUFS);
258 
259         if (hlen > MLEN) {
260             MCLGET(m, M_DONTWAIT);
261             if ((m->m_flags & M_EXT) == 0) {
262                 m_free(m);
263                 return (ENOBUFS);
264             }
265         }
266         /* move XCP Header to separate mbuf */
267         bcopy((caddr_t) xh, mtod(mx, caddr_t), hlen);
268         mx->m_len += hlen;
269         ip6->ip6_nxt = xh->cmn.proto;
270         ip6->ip6_plen -= hlen;
271 
272         /* remove header from original mbuf and pullup data */
273         bcopy((caddr_t) xh + hlen, (caddr_t) xh, m->m_len -
274             sizeof(struct ip6_hdr) - hlen);
275         m->m_len -= hlen;
276         m->m_pkthdr.len -= hlen;
277         exthdrs.ip6e_xch = mx;
278         hlen = 0;
279     }
280 #endif /* XCP */
```

Listing 4.3: *netinet6/ip6_output.c* separation of XCP header

253 - 265

Before the XCP header is separated, a new MBUF must be allocated to

hold the data. This new MBUF will be treated like an extension header further ahead.

267 - 279
The XCP header is copied to the new MBUF and the next protocol field in the IP header is updated. Also, the payload length of the packet is reduced in the size of the XCP header. The final packet length will be calculated later.

An overview of this process and the resulting packet is shown in figure 4.2.

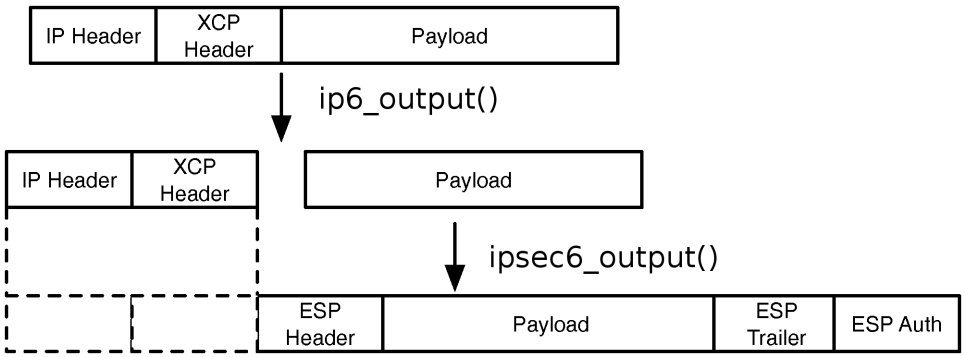


Figure 4.2: The XCP header is split from the payload and stored as an IPv6 extension header.

Just before the function calls the IPsec routines, *ipsec6_output_trans*() and *ipsec6_output_tunnel*() , the final packet chain is assembled by prepending the extension headers, including XCP, and the packet length is calculated.

4.3 IPsec routines

The IPsec routines are called during the IP packet assemble in *ip6_output*() function. The latter checks if the destinations falls in any SA policies for the outgoing flows and calls the *ipsec6_output_trans*(). This function does the following when is called:

1. Receives a pointer to the head MBUF, a pointer to the MBUF containing the extension header prior the IP payload and a pointer to the next protocol value stored in this last extension header.

2. Checks if all the pointers are valid and if the SA data is present.

3. If the IPsec mode is Tunnel, exits the function and returns to the IP output in order to the tunnel routine to be called.

4. Checks the IPsec transformation to be applyed to the packet. If it is ESP, calls the *esp6_output*() function, else if it is AH, calls the *ah6_output*() function.

5. After the packet is processed by the appropriate protocol, the final length is set and the function returns to IP output.

Since the ESP output function only encrypts the data starting from the last extension header, no change is needed on the output function. The XCP header will be excluded from the process.

4.3.1 AH functions

Concerning AH, the extension headers are used in the overall packet authentication by the hash functions, so further modifications to this routines must be made to cope with XCP.

In order to support XCP header authentication correctly, *ah6_calcchksum*() the function responsible for calculating the packet checksum, must be modified. Rather than computing the complete congestion header with no differentiation between parameters, AH copies the XCP header with no differentiation between parameters, AH copies the XCP header and sets the *Delta_Throughput* mutable field to zero. Since the modified function is contained in *ah_core.c*, both outgoing and incoming processing will behave in the same manner therefore ensuring that the hash values will match. Hence *Delta_Throughput* is always set to zero, the parameter can be safely altered by the routers along the path.

The *ah6_calccksum*() first cycles through the IP headers, calculating the hash over the appropriate fields of each header. After all IP headers are processed, the function proceds to the payload. Note that in Tunnel mode, the inner IP header is considered regular data in the payload, so all fields are authenticated. In order to the hash function detect the XCP header, new code must be added to the initial cycle. The Listing 4.4 shows the new code added to the AH hash function.

```
1510 #ifdef XCP
1511     case IPPROTO_XCP:
1512         {
1513             struct xcphdr_x xhcopy;
1514             struct m_tag *xtag;
1515 
1516             xtag = m_tag_find(m, MTAG_XCP);
1517 
1518             if (!xtag) {
1519             if (newoff - off != sizeof(struct xcphdr_x)) {
1520                 error = EINVAL;
1521                 goto fail;
1522             }
1523 
1524             m_copydata(m, off, newoff - off, (caddr_t)&xhcopy);
1525             xhcopy.delta_tput=0;  /* The only mutable field */
1526             (algo->update)(&algos, (u_int8_t *)&xhcopy,
1527                         sizeof(struct xcphdr_x));
1528             }
1529         }
1530         break;
1531 #endif /* XCP */
```

Listing 4.4: *netinet6/ah_core.c* protecting XCP header immutable fields

1516 - 1522

Search for an XCP Tag. If the Tag does not exists, the packet is outgoing. Afterwards, a sanity check is made by comparing the XCP length field with the actual size.

1524 - 1528

Copies the XCP header to a temporary variable. Then the only mutable field, *Delta_Throughput*, is zeroed and the proper hash algorithm is called to process the header stored on the variable.

Incoming packets

Treating XCP as an extension header brings undesirable implications on header authentication. While XCP precedes IPsec in processing outgoing

packets, incoming packets are processed by the order in which protocol headers are presented. The resulting mismatch in processing order is particularly worrying when dealing with authentication, since the XCP header is deleted and stored on a Tag after being processed by the XCP input function.

This is illustrated in Figure 4.3. The function responsible for processing the authentication header, *ah6_input*(), receives the packet with no XCP header, even though the header had been used to compute the hash value for the packet at the sender. Since the locally calculated hash value does not match the hash value contained in the packet, *ah6_input*() discards the packet as it may pose a security risk.

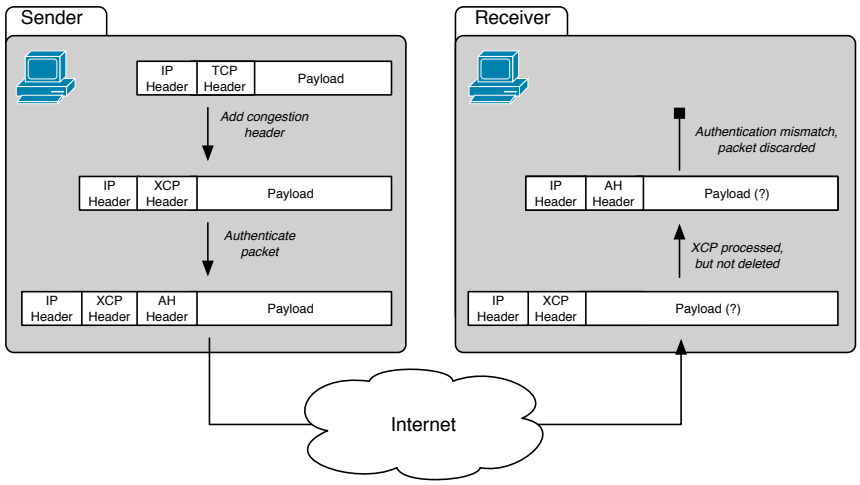


Figure 4.3: Current input processing of XCP/AH packets. The authenticated XCP header is deleted before AH processing, therefore corrupting the packet.

To overcome this issue, the checksum function must verify the existence of an XCP Tag and, if present, calculate the authentication hash using the values stored in the Tag. This can be achieved by adding additional code at the point when the outer IP header is being processed. Doing so, ensures that the search for the tag is always made since, at least, one IP header must exist. Listing 4.5 shows the new code.

```
1477 #ifdef XCP
1478         {
1479         struct m_tag *xtag;
1480
1481         xtag = m_tag_find(m, MTAG_XCP);
```

```
1482
1483            if (xtag) {
1484                struct xcphdr_x xhcopy;
1485
1486                m_copydata(m, off + sizeof(xhcopy), sizeof(xhcopy),
1487                        (caddr_t)&xhcopy);
1488                xhcopy.delta_tput=0;
1489                (algo->update)(&algos, (u_int8_t)&xhcopy,
1490                        sizeof(struct xcphdr_x));
1491                /* XCP length must be taken into account */
1492                ip6copy.ip6_len += htons(xh->cmn.len);
1493            }
1494            }
1495#endif /* XCP */
```

Listing 4.5: *netinet6/ah_core.c* searching for XCP Tag and checking the header

1483 - 1493

If an XCP Tag exists then the packet is incoming. The values stored on the Tag are copied to a temporary variable, the mutable field, *Delta_Throughput* is set to zero and the proper hash algorith is called to process the header. Note that although the XCP header is the last of the IPv6 extension headers, if any, to be hashed at the origin, here would be the first. This does not pose any problem since the hash is a checksum and it does not matter the order in which the data blocks are processed.

The proposed solution is shown in figure 4.4.

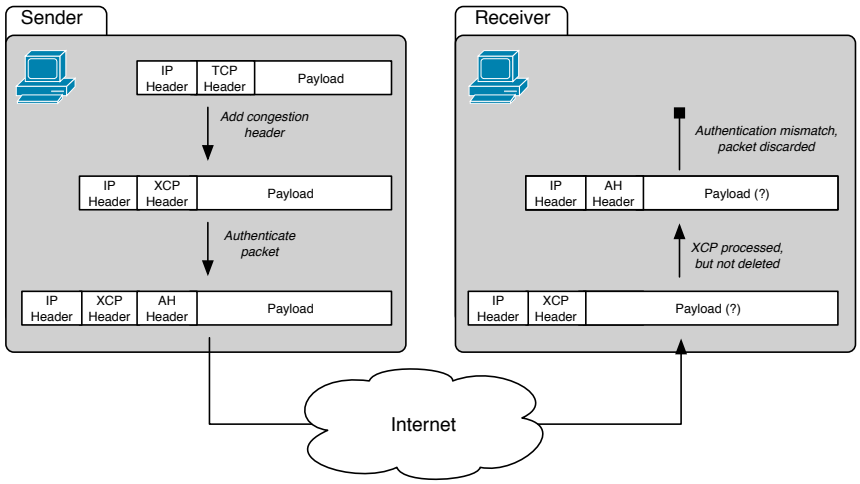


Figure 4.4: Correct input processing of XCP/AH packets.

4.4 IP Input routines

The *ip6_input*() function, located in the file with the same name, is responsible for processing the IP packet. Unlike the output function, the extension headers are not processed in separeted MBUFs. Instead, the function reads from next protocol value of the headers and calls the appropriate handler. Each handler function is responsible to return the next protocol value and the offset corresponding to the length of itself so that the ip6 input can call the subsequent handler. For Transport mode in both ESP and AH, no additional code is needed to add to the input function. The aditional Tunnel mode code is covered on the next section.

The input function of each protocol, relies on the *ip6_nexthdr*() function to determinate the next protocol and the offset of the extension headers so it is necessary to add the support for the new XCP header. The added code is presented on the Listing 4.6.

```
1546 #ifdef XCP
1547     case IPPROTO_XCP:
1548 #endif
1549         if (m->m_pkthdr.len < off + sizeof(ip6e))
1550             return -1;
1551         m_copydata(m, off, sizeof(ip6e), (caddr_t)&ip6e);
1552         if (nxtp)
1553             *nxtp = ip6e.ip6e_nxt;
1554 #ifdef XCP
1555         /* XXX XCP Header Length comes in
1556             bytes */
1557         if (proto == IPPROTO_XCP)
1558             off += ip6e.ip6e_len;
1559         else
1560 #endif
```

Listing 4.6: *netinet6/ip6_input.c* support for the XCP header to the *ip6_nexthdr* function

1546 - 1560

After a sanity check, the extension header is copied to a temporary variable and then the next protocol is copied to be returned to the IP input. Also, a selective behaviour is done when copying the offset value of the XCP header because the latter comes in bytes instead of multiple of 4 bytes like the other extension headers.

4.5 Tunnel mode

4.5.1 IP Output

Since the XCP header must be copied to the outer IP packet before the call of the encapsulation routines, aditional changes were made to *ip6_output* function. Listings 4.7 and 4.8 shows the code.

```
647 #ifdef XCP
648         if (exthdrs.ip6e_xch) {
649         MGET(mx, M_DONTWAIT, MT_DATA);
650
651         if (!mx)
652             return (ENOBUFS);
653
654         if (exthdrs.ip6e_xch->m_len > MLEN) {
655             MCLGET(m, M_DONTWAIT);
656             if ((m->m_flags & M_EXT) == 0) {
657                 m_free(m);
658                 return (ENOBUFS);
659             }
660         }
661         }
662         /* copy XCP Header to separate mbuf */
663         bcopy(mtod(exthdrs.ip6e_xch, caddr_t), mtod(mx, caddr_t),
664               exthdrs.ip6e_xch->m_len);
665         mx->m_len += exthdrs.ip6e_xch->m_len;
666 #endif /* XCP */
```

Listing 4.7: *netinet6/ip6_output.c* copy the XCP header to a temporary MBUF

647 - 660

If XCP header is present, a new MBUF is allocated to hold it.

663 - 665

The current XCP header located in its own MBUF is copied to the new one. Note that after encryption, the inner will be no longer accessible.

```
703 #ifdef XCP
704         ip6 = mtod(m, struct ip6_hdr *);
705         xh = mtod(mx, struct xcphdr_x*);
706         xh->cmn.proto = ip6->ip6_nxt;
707         ip6->ip6_nxt = IPPROTO_XCP;
708         ip6->ip6_plen += exthdrs.ip6e_xch->m_len;
709         m->m_pkthdr.len += exthdrs.ip6e_xch->m_len;
710         mx->m_next = m->m_next;
711         m->m_next = mx;
712 #endif /* XCP */
```

Listing 4.8: *netinet6/ip6_output.c* add the XCP header MBUF to the packet chain

703 - 712

After the packet encapsulation, the XCP header is added to the packet chain, after the IPv6 header. The next protocols fields of both IPv6 and XCP headers are correct to reflect the disposition. Also, the Packet payload length is increased in the size of the XCP header.

4.5.2 TCP Output

As in IPv4, the XCP operation in Tunnel mode mandates the presence of two headers as well. The approach is the same as in the IPv4, change TCP Output routine in order to detect the use of XCP in Tunnel mode and allocate extra space for the inner header. Part of the code to add support to XCP Tunnel mode is shared with IPv4. Lines from 764 to 782 of the Listing 3.7 are common to both protocols. Lines from 749 to 761 are replaced by the code in the Listing 4.9.

```
712     {
713     struct inpcb *inp;
714     struct mbuf *mips;
715     struct ip *ip;
716     struct secpolicy *sp = NULL;
717     struct ipsecrequest *isr;
718 #ifdef INET6
719     struct ip6_hdr *ip6;
720 #endif
721     struct tcphdr *th;
722
723     if ((tp == NULL) || ((inp = tp->t_inpcb) == NULL))
724         return (0);
725    MGETHDR(mips, M_DONTWAIT, MT_DATA);
726     if (!mips)
727         return (0);
728
729 #ifdef INET6
730     if ((inp->inp_vflag & INP_IPV6) != 0) {
```

```
731         ip6 = mtod(mips, struct ip6_hdr *);
732         th = (struct tcphdr *)(ip6 + 1);
733         mips->m_pkthdr.len = mips->m_len =
734             sizeof(struct ip6_hdr) + sizeof(struct tcphdr);
735         tcpip_fillheaders(inp, ip6, th);
736 
737         if (inp == NULL)
738             sp = ipsec6_getpolicybyaddr(mips, IPSEC_DIR_OUTBOUND,
739                     IP_FORWARDING, &error);
740         else
741             sp = ipsec6_getpolicybypcb(mips, IPSEC_DIR_OUTBOUND,
742                     inp, &error);
743 
744         if (sp == NULL)
745             return 0;
746 
747     } else
```

Listing 4.9: *netinet6/tcp_output.c* maximum segment size

712 - 747

A temporary MBUF is allocated to hold a skeletal TCP/IP header. This header is used to get the IPsec policy that must be applied to the destination address.

Chapter 5

Results

This chapter presents the results obtained in the use of the integration of XCP with IPsec. Sections 5.2 through 5.5 show connectivity tests using the several combinations of IPsec security services and IP versions. The final section presents a performance test for both data rate and packet processing time.

5.1 Testbed Setup

The testbed was composed by three end-hosts in different subnets connected by one intermediate router. All nodes were equipped with a 100Mbit/s capable network interface card, a Pentium-3 733MHz CPU and 512 MB of RAM. The objective of all subsequent test scenarios was to establish a successful connection between a client and the sink for both IPv4 and IPv6. Results were obtained using *wireshark* [18], a protocol dissector, running on R_1, with both the *libcrypt* library for packet decryption and validation and a patch for XCP header detection which it was developed.

All traffic was generated using XCP test tools, namely *xstream* and *xserver*, which provided respectively both client and server applications for communicating over XCP sockets. Prior to all tests, IPsec policies (SAD and SPD) on both end-hosts were adjusted to reflect the appropriate scenario settings.

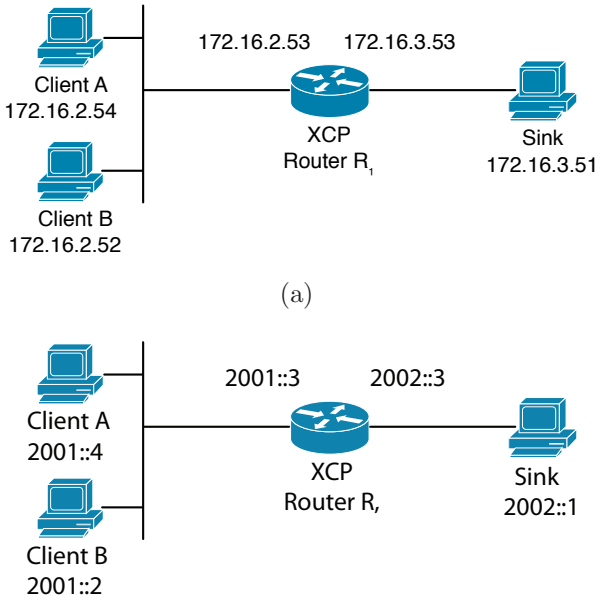


Figure 5.1: XCP/IPsec testbed setup for both IPv4 (a) and IPv6 (b).

5.2 XCP with AH

To assess XCP authentication using AH, client A and the sink were set up with an authentication policy in transport mode as described by listings 5.1 and 5.2 for IPv4 test and listings 5.3 and 5.4 for the IPv6 test. The authentication algorithm used was HMAC-SHA1.

```
add 172.16.2.54 172.16.3.51 ah 9991 -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.54 ah 9992 -A hmac-sha1 "12345678901234567890";

spdadd 172.16.3.51 172.16.2.54 any -P in ipsec ah/transport/ /require;
spdadd 172.16.2.54 172.16.3.51 any -P out ipsec ah/transport/ /require;
```

Listing 5.1: Client security policy with AH for IPv4.

```
add 172.16.2.54 172.16.3.51 ah 9991 -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.54 ah 9992 -A hmac-sha1 "12345678901234567890";
```

```
spdadd 172.16.2.54 172.16.3.51 any -P in ipsec ah/transport/ /require;
spdadd 172.16.3.51 172.16.2.54 any -P out ipsec ah/transport/ /require;
```

Listing 5.2: Server security policy with AH for IPv4.

```
add 2001::4 2002::1 ah 9991 -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::4 ah 9992 -A hmac-sha1 "12345678901234567890";

spdadd 2002::1 2001::4 any -P in ipsec ah/transport/ /require;
spdadd 2001::4 2002::1 any -P out ipsec ah/transport/ /require;
```

Listing 5.3: Client security policy with AH for IPv6.

```
add 2001::4 2002::1 ah 9991 -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::4 ah 9992 -A hmac-sha1 "12345678901234567890";

spdadd 2001::4 2002::1 any -P in ipsec ah/transport/ /require;
spdadd 2002::1 2001::4 any -P out ipsec ah/transport/ /require;
```

Listing 5.4: Server security policy with AH for IPv6.

All policies were enforced using the *setkey* command, after which a connection was established, first for IPv4 and then for IPv6, between the client application, *xstream*, running on client A, and *xserver*, which ran continuously on the testbed sink. An extract of results, as captured by *wireshark* on the router, are listed in Tables 5.1 and 5.2 for IPv4 and IPv6 respectively.

Listing 5.5: Overview of resulting XCP/AH IPv4 packet.

```
Internet Protocol, Src: 172.16.2.54, Dst: 172.16.3.51
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 128
    Identification: 0x1557 (5463)
    Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 63
    Protocol: eXplicit Congestion Protocol (0x72)
    Header checksum: 0xc82b [correct]
    Source: 172.16.2.54 (172.16.2.54)
    Destination: 172.16.3.51 (172.16.3.51)
eXplicit Congestion Protocol
    Protocol: AH (0x33)
    Header Length: 40
```

```
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Authentication Header
    Next Header: TCP (0x06)
    Length: 24
    SPI: 0x0000270a
    Sequence: 5029
    IV: F8613EB4AA1B65986695E556
Transmission Control Protocol, Src Port: 64426, Dst Port: 2525
```

Listing 5.6: Overview of resulting XCP/AH IPv6 packet.

```
Internet Protocol, Src: 2001::4, Dst: 2002::1
    Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload Length: 128
    Next Header: eXplicit Congestion Protocol (0x72)
    Hop limit: 255
    Source: 2001::4 (2001::4)
    Destination: 2001::1 (2001::1)
eXplicit Congestion Protocol
    Protocol: AH (0x33)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Authentication Header
    Next Header: TCP (0x06)
    Length: 24
    SPI: 0x0000270a
    Sequence: 5029
    IV: BA34C2B4AA1B6598AA45E556
Transmission Control Protocol, Src Port: 64426, Dst Port: 2525
```

Time	Source	Destination	Protocol	Info
4.125327	172.16.2.54	172.16.3.51	TCP	64426 > 2525 [SYN] Seq=0 Len=0 MSS=1460 WS=1 TSV=2167312 TSER=0
4.126505	172.16.3.51	172.16.2.54	TCP	2525 > 64426 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1 TSV=1980160 TSER=2167312
4.127118	172.16.2.54	172.16.3.51	TCP	64426 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=0 TSV=2167314 TSER=1980160
4.128022	172.16.2.54	172.16.3.51	TCP	64426 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=1384 TSV=2167314 TSER=1980160
4.229352	172.16.3.51	172.16.2.54	TCP	2525 > 64426 [ACK] Seq=1 Ack=1385 Win=66608 Len=0 TSV=1980263 TSER=2167314
4.230355	172.16.2.54	172.16.3.51	TCP	64426 > 2525 [ACK] Seq=1385 Ack=1 Win=66608 Len=1384 TSV=2167416 TSER=1980263
4.331325	172.16.3.51	172.16.2.54	TCP	2525 > 64426 [ACK] Seq=1 Ack=2769 Win=66608 Len=0 TSV=1980365 TSER=2167416

Table 5.1: Establishment of TCP/IPv4 session with XCP and AH.

Time	Source	Destination	Protocol	Info
3.125427	2001::4	2002::1	TCP	34656 > 2525 [SYN] Seq=0 Len=0 MSS=1460 WS=1 TSV=2167312 TSER=0
3.126305	2002::1	2001::4	TCP	2525 > 34656 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1 TSV=1980160 TSER=2167312
3.127418	2001::4	2002::1	TCP	34656 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=0 TSV=2167314 TSER=1980160
3.128422	2001::4	2002::1	TCP	34656 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=1384 TSV=2167314 TSER=1980160
3.229152	2002::1	2001::4	TCP	2525 > 34656 [ACK] Seq=1 Ack=1385 Win=66608 Len=0 TSV=1980263 TSER=2167314
3.230255	2001::4	2002::1	TCP	34656 > 2525 [ACK] Seq=1385 Ack=1 Win=66608 Len=1384 TSV=2167416 TSER=1980263
3.331425	2002::1	2001::4	TCP	2525 > 34656 [ACK] Seq=1 Ack=2769 Win=66608 Len=0 TSV=1980365 TSER=2167416

Table 5.2: Establishment of TCP/IPv6 session with XCP and AH.

These tables show a successful establishment of a TCP/IPv4 and IPv6 connection using both XCP and AH. A detailed analysis of the first packet of both IP protocols is provided in Listings 5.5 and 5.6.

As expected, the XCP Header comes before the AH Header, and the highlighted *Protocol* and *next header* parameters are sequenced in the appropriate order. The TCP header shows the establishment of a connection to the remote port 2525, to which the server-side XCP socket is bound.

5.3 XCP with ESP

Assessing packet encryption using ESP, whilst maintaining explicit congestion control, was achieved using the same test procedure used for AH. Client B and the sink were both configured to use an encryption policy over transport mode, as described in listings 5.7 and 5.8 for the IPv4 test, and in Listings 5.9 and 5.10 for the IPv6 test. The encryption algorithm used was DES-CBC.

```
add 172.16.2.52 172.16.3.51 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.52 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";

spdadd 172.16.3.51 172.16.2.52 any -P in ipsec esp/transport/ /require;
spdadd 172.16.2.52 172.16.3.51 any -P out ipsec esp/transport/ /require;
```

Listing 5.7: Client security policy with ESP for IPv4.

```
add 172.16.2.52 172.16.3.51 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.52 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";

spdadd 172.16.2.52 172.16.3.51 any -P in ipsec esp/transport/ /require;
spdadd 172.16.3.51 172.16.2.52 any -P out ipsec esp/transport/ /require;
```

Listing 5.8: Server security policy with ESP for IPv4.

```
add 2001::2 2002::1 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::2 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";

spdadd 2002::1 2001::2 any -P in ipsec esp/transport/ /require;
spdadd 2001::2 2002::1 any -P out ipsec esp/transport/ /require;
```

Listing 5.9: Client security policy with ESP for IPv6.

```
add 2001::2 2002::1 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::2 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
```

```
spdadd 2001::2 2002::1 any -P in ipsec esp/transport/ /require;
spdadd 2002::1 2001::2 any -P out ipsec esp/transport/ /require;
```

Listing 5.10: Server security policy with ESP for IPv6.

As with AH, IPsec policies were enforced using the *setkey* command. The corresponding connection establishment using the standard XCP client and server applications on both end-hosts is represented in Table 5.3 for IPv4 test and in Table 5.4, as captured on the router using *wireshark*.

Time	Source	Destination	Protocol	Info
7.824979	172.16.2.52	172.16.3.51	TCP	50871 > 2525 [SYN] Seq=0 Len=0 MSS=1460 WS=1 TSV=55897432 TSER=0
7.825695	172.16.3.51	172.16.2.52	TCP	2525 > 50871 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1 TSV=1581962 TSER=55897432
7.826461	172.16.2.52	172.16.3.51	TCP	50871 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=0 TSV=55897434 TSER=1581962
7.827945	172.16.2.52	172.16.3.51	TCP	50871 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=1371 TSV=55897434 TSER=1581962
7.928693	172.16.3.51	172.16.2.52	TCP	2525 > 50871 [ACK] Seq=1 Ack=1372 Win=66608 Len=0 TSV=1582066 TSER=55897434
7.930192	172.16.2.52	172.16.3.51	TCP	50871 > 2525 [ACK] Seq=1372 Ack=1 Win=66608 Len=1371 TSV=55897536 TSER=1582066
8.030639	172.16.3.51	172.16.2.52	TCP	2525 > 50871 [ACK] Seq=1 Ack=2743 Win=66608 Len=0 TSV=1582168 TSER=55897536

Table 5.3: Establishment of TCP/IPv4 session with XCP and ESP.

Time	Source	Destination	Protocol	Info
7.424979	2001::2	2002::1	TCP	28743 > 2525 [SYN] Seq=0 Len=0 MSS=1460 WS=1 TSV=55897432 TSER=0
7.425695	2002::1	2001::2	TCP	2525 > 28743 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1 TSV=1581962 TSER=55897432
7.426461	2001::2	2002::1	TCP	28743 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=0 TSV=55897434 TSER=1581962
7.427945	2001::2	2002::1	TCP	28743 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=1371 TSV=55897434 TSER=1581962
7.528693	2002::1	2001::2	TCP	2525 > 28743 [ACK] Seq=1 Ack=1372 Win=66608 Len=0 TSV=1582066 TSER=55897434
7.530192	2001::2	2002::1	TCP	28743 > 2525 [ACK] Seq=1372 Ack=1 Win=66608 Len=1371 TSV=55897536 TSER=1582066
8.630639	2002::1	2001::2	TCP	2525 > 28743 [ACK] Seq=1 Ack=2743 Win=66608 Len=0 TSV=1582168 TSER=55897536

Table 5.4: Establishment of TCP/IPv6 session with XCP and ESP.

Once again, these results demonstrate a successful TCP connection with XCP in IPv4 and IPv6, this time using ESP for payload encryption. A

detailed perspective of the first packet of each IP protocol is provided in Listings 5.11 and 5.12, which both confirms correct header placement and rectified protocol fields.

Listing 5.11: Overview of resulting XCP/ESP IPv4 packet.

```
Internet Protocol, Src: 172.16.2.52, Dst: 172.16.3.51
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 136
    Identification: 0x5ecb (24267)
    Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 63
    Protocol: eXplicit Congestion Protocol (0x72)
    Header checksum: 0x7eb1 [correct]
    Source: 172.16.2.52 (172.16.2.52)
    Destination: 172.16.3.51 (172.16.3.51)
eXplicit Congestion Protocol
    Protocol: ESP (0x32)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Encapsulating Security Payload
    SPI: 0x00002708
    Sequence: 46179
    IV: D6766250C22F2193
    Pad
    Pad Length: 2
    Next header: TCP (0x06)
    Authentication Data [correct]
Transmission Control Protocol, Src Port: 50871, Dst Port: 2525
```

Listing 5.12: Overview of resulting XCP/ESP IPv6 packet.

```
Internet Protocol, Src: 2001::4, Dst: 2002::1
    Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload Length: 136
    Next Header: eXplicit Congestion Protocol (0x72)
    Hop limit: 255
    Source: 2001::4 (2001::4)
    Destination: 2002::1 (2002::1)
eXplicit Congestion Protocol
    Protocol: ESP (0x32)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
```

```
Encapsulating Security Payload
    SPI: 0x00002708
    Sequence: 46179
    IV: AD876850934C78D1
    Pad
    Pad Length: 2
    Next header: TCP (0x06)
    Authentication Data [correct]
Transmission Control Protocol, Src Port: 50871, Dst Port: 2525
```

5.4 XCP with AH and ESP

The final transport mode test involved establishing a TCP connection between end-hosts using XCP for congestion control, AH for packet authentication and ESP for packet encryption. To this end, client B and the sink were configured with both security policies, as detailed in Listings 5.13 and 5.14 for IPv4 and Listings 5.15 and 5.16 for IPv6, respectively. Encryption and authentication algorithms remained the same as those used in previous tests, namely DES-CBC and HMAC-SHA1. This time however, ESP authentication was not used, as it would be redundant whilst using AH.

```
add 172.16.2.52 172.16.3.51 esp 9991 -E des-cbc "12345678";
add 172.16.3.51 172.16.2.52 esp 9992 -E des-cbc "12345678";
add 172.16.2.52 172.16.3.51 ah 9993 -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.52 ah 9994 -A hmac-sha1 "12345678901234567890";

spdadd 172.16.3.51 172.16.2.52 any -P in ipsec esp/transport/ /require;
spdadd 172.16.3.51 172.16.2.52 any -P in ipsec ah/transport/ /require;
spdadd 172.16.2.52 172.16.3.51 any -P out ipsec esp/transport/ /require;
spdadd 172.16.2.52 172.16.3.51 any -P out ipsec ah/transport/ /require;
```

Listing 5.13: Client security policy with AH and ESP for IPv4.

```
add 172.16.2.52 172.16.3.51 esp 9991 -E des-cbc "12345678";
add 172.16.3.51 172.16.2.52 esp 9992 -E des-cbc "12345678";
add 172.16.2.52 172.16.3.51 ah 9993 -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.52 ah 9994 -A hmac-sha1 "12345678901234567890";

spdadd 172.16.2.52 172.16.3.51 any -P in ipsec esp/transport/ /require;
spdadd 172.16.2.52 172.16.3.51 any -P in ipsec ah/transport/ /require;
spdadd 172.16.3.51 172.16.2.52 any -P out ipsec esp/transport/ /require;
spdadd 172.16.3.51 172.16.2.52 any -P out ipsec ah/transport/ /require;
```

Listing 5.14: Server security policy with AH and ESP for IPv4.

```
add 2001::2 2002::1 esp 9991 -E des-cbc "12345678";
add 2002::1 2001::2 esp 9992 -E des-cbc "12345678";
add 2001::2 2002::1 ah 9993 -A hmac-sha1 "12345678901234567890";
add 2001::1 2001::2 ah 9994 -A hmac-sha1 "12345678901234567890";

spdadd 2002::1 2001::2 any -P in ipsec esp/transport/ /require;
spdadd 2002::1 2001::2 any -P in ipsec ah/transport/ /require;
spdadd 2001::2 2002::1 any -P out ipsec esp/transport/ /require;
spdadd 2001::2 2002::1 any -P out ipsec ah/transport/ /require;
```

Listing 5.15: Client security policy with AH and ESP for IPv6.

```
add 2001::2 2002::1 esp 9991 -E des-cbc "12345678";
add 2002::1 2001::2 esp 9992 -E des-cbc "12345678";
add 2001::2 2002::1 ah 9993 -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::2 ah 9994 -A hmac-sha1 "12345678901234567890";

spdadd 2001::2 2002::1 any -P in ipsec esp/transport/ /require;
spdadd 2001::2 2002::1 any -P in ipsec ah/transport/ /require;
spdadd 2002::1 2001::2 any -P out ipsec esp/transport/ /require;
spdadd 2002::1 2001::2 any -P out ipsec ah/transport/ /require;
```

Listing 5.16: Server security policy with AH and ESP for IPv6.

After setting these security policies and establishing the TCP connection, the log presented in tables 5.5 for IPv4 and 5.6 for IPv6 were obtained from running *wireshark* on the intermediate router.

Time	Source	Destination	Protocol	Info
14.297127	172.16.2.52	172.16.3.51	TCP	64641 > 2525 [SYN] Seq=0 Len=0 MSS=1460 WS=1 TSV=57740983 TSER=0
14.298755	172.16.3.51	172.16.2.52	TCP	2525 > 64641 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1 TSV=3425410 TSER=57740983
14.300131	172.16.2.52	172.16.3.51	TCP	64641 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=0 TSV=57740988 TSER=3425410
14.302268	172.16.2.52	172.16.3.51	TCP	64641 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=1347 TSV=57740989 TSER=3425410
14.404285	172.16.3.51	172.16.2.52	TCP	2525 > 64641 [ACK] Seq=1 Ack=1348 Win=66608 Len=0 TSV=3425516 TSER=57740989
14.406543	172.16.2.52	172.16.3.51	TCP	64641 > 2525 [ACK] Seq=1348 Ack=1 Win=66608 Len=1347 TSV=57741094 TSER=3425516
14.508218	172.16.3.51	172.16.2.52	TCP	2525 > 64641 [ACK] Seq=1 Ack=2695 Win=66608 Len=0 TSV=3425620 TSER=57741094

Table 5.5: Establishment of TCP/IPv4 session with XCP, AH and ESP.

These table show a successful establishment of a TCP connection using XCP, AH and ESP. A detailed view of the first packet for each IP protocol is shown in Listings 5.17 and 5.18.

Once again, XCP precedes AH and ESP, whilst maintaining protocol sequencing coherent with packet formation. The order by which AH and ESP header are layered is dictated by the corresponding order in the policies previously defined for end-hosts. In this case, the payload was first encrypted and then authenticated.

Time	Source	Destination	Protocol	Info
4.194127	2001::2	2002::1	TCP	43875 > 2525 [SYN] Seq=0 Len=0 MSS=1460 WS=1 TSV=57740983 TSER=0
4.193755	2002::1	2001::2	TCP	2525 > 43875 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1 TSV=3425410 TSER=57740983
4.213131	2001::2	2002::1	TCP	43875 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=0 TSV=57740988 TSER=3425410
4.216268	2001::2	2002::1	TCP	43875 > 2525 [ACK] Seq=1 Ack=1 Win=66608 Len=1347 TSV=57740989 TSER=3425410
4.314285	2002::1	2001::2	TCP	2525 > 43875 [ACK] Seq=1 Ack=1348 Win=66608 Len=0 TSV=3425516 TSER=57740989
4.315543	2001::2	2002::1	TCP	43875 > 2525 [ACK] Seq=1348 Ack=1 Win=66608 Len=1347 TSV=57741094 TSER=3425516
4.414218	2002::1	2001::2	TCP	2525 > 43875 [ACK] Seq=1 Ack=2695 Win=66608 Len=0 TSV=3425620 TSER=57741094

Table 5.6: Establishment of TCP/IPv6 session with XCP, AH and ESP.

Listing 5.17: Overview of resulting XCP/AH/ESP IPv4 packet.

```
Internet Protocol, Src: 172.16.2.52, Dst: 172.16.3.51
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 160
    Identification: 0xf7ea (63466)
    Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 63
    Protocol: eXplicit Congestion Protocol (0x72)
    Header checksum: 0xe579 [correct]
    Source: 172.16.2.52 (172.16.2.52)
    Destination: 172.16.3.51 (172.16.3.51)
eXplicit Congestion Protocol
    Protocol: AH (0x33)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Authentication Header
    Next Header: ESP (0x32)
    Length: 24
    SPI: 0x00002708
    Sequence: 19547
    IV: 2CD7D29A1EE3503FC9FF8134
Encapsulating Security Payload
    SPI: 0x00002708
    Sequence: 19547
    IV: 247BCE86926079F6
    Pad
    Pad Length: 2
    Next header: TCP (0x06)
    Authentication Data [correct]
Transmission Control Protocol, Src Port: 64641, Dst Port: 2525
```

Listing 5.18: Overview of resulting XCP/AH/ESP IPv6 packet.

```
Internet Protocol, Src: 2001::2, Dst: 2001::1
    Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload Length: 160
    Next Header: eXplicit Congestion Protocol (0x72)
    Hop limit: 255
    Source: 2001::2 (2001::2)
    Destination: 2001::1 (2001::1)
eXplicit Congestion Protocol
    Protocol: AH (0x33)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Authentication Header
    Next Header: ESP (0x32)
    Length: 24
    SPI: 0x00002708
    Sequence: 19547
    IV: 4AB9F4AB47DAECC9683A8FF4
Encapsulating Security Payload
    SPI: 0x00002708
    Sequence: 19547
    IV: 729384AD860953FF
    Pad
    Pad Length: 2
    Next header: TCP (0x06)
    Authentication Data [correct]
Transmission Control Protocol, Src Port: 64641, Dst Port: 2525
```

5.5 XCP in tunnel mode

The final test required validating adequate congestion control in tunnel mode. In such cases, end-hosts must be configured with a virtual interface which parses the inner packet. Client A and the sink were both configured in such a manner, with the resulting interface belonging to the 192.168.1.0/24 subnet for the IPv4 test and to the 2003::/64 subnet for the IPv6 test. The encryption policy and algorithm remained the same as in previous tests, but in tunnel mode. The use of AH in tunnel mode is unnecessary as ESP authenticates inner IP packet. Additionally, FreeBSD 6.0 currently drops packets with AH in tunnel mode, since authenticating the outer header does not guarantee the inner packet has not been tampered with. Configuration files for both client and sink are listed in 5.19 and 5.20 for IPv4 and listed in 5.21 and 5.22 for IPv6.

```
add 172.16.2.54 172.16.3.51 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.54 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";

spdadd 192.168.1.0/24 192.168.1.0/24
   any -P in ipsec esp/tunnel/172.16.3.51-172.16.2.54/require;
spdadd 192.168.1.0/24 192.168.1.0/24
   any -P out ipsec esp/tunnel/172.16.2.54-172.16.3.51/require;
```

Listing 5.19: Client security policy with tunnel mode encryption for IPv4.

```
add 172.16.2.54 172.16.3.51 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 172.16.3.51 172.16.2.54 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";

spdadd 192.168.1.0/24 192.168.1.0/24
   any -P in ipsec esp/tunnel/172.16.2.54-172.16.3.51/require;
spdadd 192.168.1.0/24 192.168.1.0/24
   any -P out ipsec esp/tunnel/172.16.3.51-172.16.2.54/require;
```

Listing 5.20: Server security policy with tunnel mode encryption for IPv4.

```
add 2001::4 2002::1 esp 9991 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::4 esp 9992 -E des-cbc "12345678"
                                     -A hmac-sha1 "12345678901234567890";

spdadd 2003::/64 2003::/64
```

```
   any -P in ipsec esp/tunnel/2002::1 2001::4/require;
spdadd 2003::/64 2003::/64
   any -P out ipsec esp/tunnel/2001::4 2002::1/require;
```

Listing 5.21: Client security policy with tunnel mode encryption for IPv6.

```
add 2001::4 2002::1 esp 9991 -E des-cbc "12345678"
                                    -A hmac-sha1 "12345678901234567890";
add 2002::1 2001::4 esp 9992 -E des-cbc "12345678"
                                    -A hmac-sha1 "12345678901234567890";

spdadd 2003::/64 2003::/64
   any -P in ipsec esp/tunnel/2001::4 2002::1/require;
spdadd  2003::/64 2003::/64
   any -P out ipsec esp/tunnel/2002::1 2001::4/require;
```

Listing 5.22: Server security policy with tunnel mode encryption for IPv6.

For the IPv4 test, Client A, configured with an IP address of 192.168.1.54, launched the *xstream* application to connect to the corresponding *xserver*, running on the testbed sink, configured with the IP address 192.168.1.51. An extract of the resulting log captured by *wireshark* is listed in table 5.7. For the IPv6 test, Client A was configured with an IP address of 2003::4 and the sink with 2003::1. An extract of the resulting log is listed in Table 5.8

Time	Source	Destination	Protocol	Info
33.709072	192.168.1.54	192.168.1.51	TCP	64863 > 2525 [SYN] Seq=0 Len=0 MSS=1240 WS=1 TSV=62442663 TSER=0
33.716051	192.168.1.51	192.168.1.54	TCP	2525 > 64863 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1240 WS=1 TSV=7914128 TSER=62442663
33.719154	192.168.1.54	192.168.1.51	TCP	64863 > 2525 [ACK] Seq=1 Ack=1 Win=66312 Len=0 TSV=62442673 TSER=7914128
33.738299	192.168.1.54	192.168.1.51	TCP	64863 > 2525 [ACK] Seq=1 Ack=1 Win=66312 Len=1131 TSV=62442676 TSER=7914128
33.756981	192.168.1.54	192.168.1.51	TCP	64863 > 2525 [ACK] Seq=1132 Ack=1 Win=66312 Len=1131 TSV=62442695 TSER=7914128
33.763062	192.168.1.51	192.168.1.54	TCP	2525 > 64863 [ACK] Seq=1 Ack=2263 Win=65180 Len=0 TSV=7914176 TSER=62442676
33.775456	192.168.1.54	192.168.1.51	TCP	64863 > 2525 [ACK] Seq=2263 Ack=1 Win=66312 Len=1131 TSV=62442714 TSER=7914128
33.801438	192.168.1.51	192.168.1.54	TCP	2525 > 64863 [ACK] Seq=1 Ack=4525 Win=65180 Len=0 TSV=7914214 TSER=62442714

Table 5.7: Establishment of TCP/IPv4 session with XCP in tunnel mode.

Time	Source	Destination	Protocol	Info
3.203452	2003::4	2003::1	TCP	65431 > 2525 [SYN] Seq=0 Len=0 MSS=1240 WS=1 TSV=62442663 TSER=0
3.212341	2003::1	2003::4	TCP	2525 > 65431 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1240 WS=1 TSV=7914128 TSER=62442663
3.214574	2003::4	2003::1	TCP	65431 > 2525 [ACK] Seq=1 Ack=1 Win=66312 Len=0 TSV=62442673 TSER=7914128
3.235469	2003::4	2003::1	TCP	65431 > 2525 [ACK] Seq=1 Ack=1 Win=66312 Len=1131 TSV=62442676 TSER=7914128
3.254356	2003::4	2003::1	TCP	65431 > 2525 [ACK] Seq=1132 Ack=1 Win=66312 Len=1131 TSV=62442695 TSER=7914128
3.264563	2003::1	2003::4	TCP	2525 > 65431 [ACK] Seq=1 Ack=2263 Win=65180 Len=0 TSV=7914176 TSER=62442676
3.275345	2003::4	2003::1	TCP	65431 > 2525 [ACK] Seq=2263 Ack=1 Win=66312 Len=1131 TSV=62442714 TSER=7914128
3.301345	2003::1	2003::4	TCP	2525 > 65431 [ACK] Seq=1 Ack=4525 Win=65180 Len=0 TSV=7914214 TSER=62442714

Table 5.8: IPv6: Establishment of TCP/IPv6 session with XCP in tunnel mode.

These tables show the successful establishment of a TCP connection with XCP and ESP in tunnel mode for both IPv4 and IPv6. Note that both Source and Destination IP addresses correspond to the inner IP Packet. A detailed view of the first packet for each IP protocol is shown in Listings 5.23 5.24.

Listing 5.23: Overview of resulting XCP IPv4 packet in tunnel mode.

```
Internet Protocol, Src: 172.16.2.54, Dst: 172.16.3.51
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 200
    Identification: 0x5b41 (23361)
    Flags: 0x00
    Fragment offset: 0
    Time to live: 63
    Protocol: eXplicit Congestion Protocol (0x72)
    Header checksum: 0xc1f9 [correct]
    Source: 172.16.2.54 (172.16.2.54)
    Destination: 172.16.3.51 (172.16.3.51)
eXplicit Congestion Protocol
    Protocol: ESP (0x32)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Encapsulating Security Payload
    SPI: 0x0000270a
    Sequence: 9
    IV: 17E4D0515CBF8923
    Pad
```

```
    Pad Length: 6
    Next header: IPIP (0x04)
    Authentication Data [correct]
Internet Protocol, Src: 192.168.1.54, Dst: 192.168.1.51
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 104
    Identification: 0x5b40 (23360)
    Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: eXplicit Congestion Protocol (0x72)
    Header checksum: 0x5b2a [correct]
    Source: 192.168.1.54 (192.168.1.54)
    Destination: 192.168.1.51 (192.168.1.51)
eXplicit Congestion Protocol
    Protocol: TCP (0x06)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Transmission Control Protocol, Src Port: 64863, Dst Port: 2525
```

Listing 5.24: Overview of resulting XCP IPv6 packet in tunnel mode.

```
Internet Protocol, Src: 2001::4, Dst: 2002::1
    Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload Length: 200
    Next Header: eXplicit Congestion Protocol (0x72)
    Hop limit: 255
    Source: 2001::4 (2001::4)
    Destination: 2002::1 (2002::1)
eXplicit Congestion Protocol
    Protocol: ESP (0x32)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Encapsulating Security Payload
    SPI: 0x0000270a
    Sequence: 9
    IV: 546ADBE878AD3453
    Pad
    Pad Length: 6
    Next header: IPIP (0x04)
    Authentication Data [correct]
Internet Protocol, Src: 2003::4, Dst: 2003::1
    Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload Length: 104
    Next Header: eXplicit Congestion Protocol (0x72)
    Hop limit: 255
    Source: 2003::4 (2003::4)
```

```
    Destination: 2003::1 (2003::1)
eXplicit Congestion Protocol
    Protocol: TCP (0x06)
    Header Length: 40
    Version: 2
    Format: Minimal (0x02)
    Reverse Feedback: 0
Transmission Control Protocol, Src Port: 64863, Dst Port: 2525
```

As expected, all headers appear in the correct order, with two XCP headers present in the complete packet structure. During transmission intermediate routers modify the outer header, since the inner XCP header is inaccessible. On arrival, the inner header is replaced with the values contained in the outer header, therefore guaranteeing that any bottleneck along the path is correctly determined by XCP.

5.6 Performance Test

The final test evaluates the impact of the implementation on the data rate and packet processing time. This test compares the performance between TCP and XCP with and without IPsec. The test was made between client A and client B and the obtained results are shown in the following sections.

5.6.1 Data Rate

To measure the data rate, client A was running *xstream* to generate traffic while client B was running *xserver*. The *xstream* application outputs an average data rate every 10 seconds and sends packets with size of 1500 byte. The test duration was 5 minutes for each combination of TCP and XCP in AH, ESP and Tunnel Mode. The test was conducted for both IPv4 and IPv6 scenarios as shown in Tables 5.9 and 5.10.

IPsec Proto	TCP (Mbit/s)	W/o IPsec	XCP (Mbit/s)	W/o IPsec	XCP/TCP
None	93.86	-	90.95	-	-3.11%
AH	51.09	-45.56%	49.49	-47.75%	-3.14%
ESP	28.06	-70.11	27.18	-71.04%	-3.13%
Tunnel	24.57	-73.82%	23.43	-75.04%	-4.66%

Table 5.9: Average data rate for the several IPsec scenarios in IPv4.

IPsec Proto	TCP (Mbit/s)	W/o IPsec	XCP (Mbit/s)	W/o IPsec	XCP/TCP
None	92.53	-	89.69	-	-3.08%
AH	48.04	-48.08%	46.55	-49.69%	-3.11%
ESP	26.55	-71.31	25.73	-72.19%	-3.08%
Tunnel	23.8	-74.28%	22.71	-75.46%	-4.58%

Table 5.10: Average data rate for the several IPsec scenarios in IPv6.

The third and fifth columns show the difference in percentage between the rate of TCP without IPsec and the rate of each scenario. The last column shows the difference between XCP and TCP for the same IPsec configuration. In the first entry of each table, the difference between TCP and XCP is arrond 3.1% which corresponds to the additional overhead of the XCP header. Also, between IPv4 and IPv6, a general decrease of

arround 2.0% is seen due to the larger IPv6 header. It can be seen that the rate falls to at least an half of TCP when IPsec is used. The reason of such decrease, which is much larger than the overhead of the AH/ESP header, is due to CPU limitation. Nevertheless, the difference between TCP and XCP is still arround 3.1% for AH and ESP, the same as without IPsec, showing that the added code does not degrade the data rate in both IPv4 and IPv6. For Tunnel mode the rate drops even further, in percentage comparing with the previous scenarios, due to the copy of the XCP header included in the inner IP packet, adding an extra overhead.

5.6.2 Packet Processing Time

The average packet processing time was derived by measuring the difference between the time a packet arrives at the sink and an ack is sent back. The values were obtained with *Wireshark* for the several scenarios in IPv4 and IPv6 as shown in tables 5.11 and 5.12.

IPsec Proto	TCP (ms)	W/o IPsec	XCP (ms)	W/o IPsec	XCP/TCP
None	0.149	-	0.165	-	10.74%
AH	0.259	73.83%	0.287	92.39%	10.68%
ESP	0.280	87.92	0.308	106.49%	9.88%
Tunnel	0.336	125.95%	0.380	155.03%	12.87%

Table 5.11: Average processing time for the several IPsec scenarios in IPv4.

IPsec Proto	TCP (ms)	W/o IPsec	XCP (ms)	W/o IPsec	XCP/TCP
None	0.152	-	0.167	-	3.08%
AH	0.279	83.99%	0.291	91.12%	3.87%
ESP	0.332	118.42	0.345	126.97%	3.92%
Tunnel	0.335	120.39%	0.347	128.29%	3.58%

Table 5.12: Average processing time for the several IPsec scenarios in IPv6.

The third and fifth columns show the difference in percentage between the rate of TCP without IPsec and the rate of each scenario. The last column shows the difference between XCP and TCP for the same IPsec configuration. As with the previous table, IPsec introduces a higher delay which can go up to 155%. Also, XCP has a higher processing time than the standard TCP but this delay is approximately equal for all all scenarios,

so it can be inferred that the implementation has no visible impact on the processing time. It is also noticeable that IPv6 has considerable less delay than IPv4 in all IPsec scenarios. This is due to the native support of IPsec in the IPv6 stack.

Chapter 6

Conclusion

In this final chapter conclusions are drawn on the proposed objectives and the developed work that was made.

FreeBSD stack A great deal of effort went into learning the inner workings of the networking stack. Engineered towards performance first and foremost, understanding the kernel presents a challenge within itself. The typical FreeBSD scattered and sparsely documented also made the learning more difficult.

Nevertheless, all the networking routines were pinpointed which allowed to derive the modifications to the kernel.

Explicit congestion control with network security The second task was to integrate explicit congestion protocols (e.g XCP) with IPsec, the security framework responsible for providing data integrity and privacy at the network layer, in IPv4 and IPv6. The proposed implementation corrects architectural fallacies in the networking stack which prevent routers from providing feedback for explicit congestion control protocols in the presence of packet encryption or authentication. This approach was validated by establishing secure end-to-end connections subject to congestion control by such protocols. Also, the performance test shown that the modifications has a negligible impact on the data rate and packet processing time.

These modifications are relevant in allowing explicit congestion control to be performed without compromising security and provide additional protection to immutable data fields associated to the congestion control header, an important benefit in a control system where end-host statistics influence core network response.

Whilst the implementation successfully extended IPsec to become aware of an explicit congestion control protocol, protocol transparency requires

that network layer security be decoupled from IP. With the emergence of end-to-network protocols such as XCP and RCP, the borderline between network and transport layer may no longer be drawn at IP. By incorrectly assuming that security must follow the IP header, IPsec constitutes an obstacle to any protocol wishing to operate both over IP and at the network layer. In IPv6, these issues may be partially circumvented by implementing XCP as an IPv6 Extension header as was made on the proposed approach.

Appendix A

BSD License

Bibliography

[1] I. S. Institute, "Transmission Control Protocol - Darpa Internet Program Protocol Specification." RFC 793, September 1981.

[2] V. Jacobson and M. Karels, "Congestion Avoidance and Control," tech. rep., ACM Computer Communication Review, August 1988.

[3] A. Falk, D. Katabi, and Y. Pryadkin, "Specification for the Explicit Control Protocol (XCP) - draft-falk-xcp-02." Internet-Draft, November 2006.

[4] N. Dukkipati and N. McKeown, "Processor Sharing Flows in the Internet," Tech. Rep. TR04-HPNG-061604, Stanford University, June 2004.

[5] R. Atkinson, "Security Architecture for the Internet Protocol." RFC 1825, August 1995.

[6] S. Kent and K. Seo, "Security Architecture for the Internet Protocol." RFC 4301, December 2005.

[7] S. Kent, "IP Authentication Headeri." RFC 4302, Decemmber 2005.

[8] S. Kent, "IP Encapsulating Security Payload (ESP)." RFC 4303, Decemmber 2005.

[9] C. Madson and N. Doraswamy, "The ESP DES-CBC Cipher Algorithm With Explicit IV." RFC 2405, November 1998.

[10] R. Pereira and R. Adams, "The ESP CBC-Mode Cipher Algorithms." RFC 2451, November 1998.

[11] R. Housley, "Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)." RFC 4309, December 2005.

[12] C. Madson and R. Glenn, "The Use of HMAC-MD5-96 within ESP and AH." RFC 2403, November 1998.

[13] C. Madson and R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH." RFC 2404, November 1998.

[14] D. Clark, "Window and Acknowledgement Strategy in TCP." RFC 813, July 1982.

[15] "About FreeBSD's Internetworking." http://www.freebsd.org/internet.html.

[16] "Nearly 2.5 Million Active Sites running FreeBSD." http://news.netcraft.com/archives/2004/06/07/sitesrunningfreebsd.html.

[17] Michael O'Brien, "SunExpert," August 1996.

[18] "Wireshark - Network Protocol Analyzer." http://www.wireshark.org/.