Performance Evaluation for Secure Internet Group Management Protocol and Group Security Association Management Protocol

Lin Chen

A Thesis

in

The Department

of

Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science at

Concordia University

Montréal, Québec, Canada

September 2017

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Lin Chen**
Entitled:	**Performance Evaluation for Secure Internet Group Management Protocol and Group Security Association Management Protocol**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Marta Kersten-Oertel

______________________________ Examiner
Dr. Hovhannes A. Harutyunyan

______________________________ Examiner
Dr. Jaroslav Opatrny

______________________________ Supervisor
Dr. J. William Atwood

Approved by ______________________________
Volker Haarslev, Graduate Program (Research) Director
Department of Computer Science & Software Engineering

____________ 2017 ______________________________
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Performance Evaluation for Secure Internet Group Management Protocol and Group Security Association Management Protocol

Lin Chen

Multicast distribution employs the model of many-to-many so that it is a more efficient way of data delivery compared to traditional one-to-one unicast distribution, which can benefit many applications such as media streaming. However, the lack of security features in its nature makes multicast technology much less popular in an open environment such as the Internet. Internet Service Providers (ISPs) take advantage of IP multicast technology's high efficiency of data delivery to provide Internet Protocol Television (IPTV) to their users. But without the full control on their networks, ISPs can not collect revenue for the services they provide. Secure Internet Group Management Protocol (SIGMP), an extension of Internet Group Management Protocol (IGMP), and Group Security Association Management Protocol (GSAM) have been proposed to enforce receiver access control at the network level of IP multicast. In this thesis, we analyze operational detail and issues of both SIGMP and GSAM. An examination of performance of both protocols is also conducted.

Keywords: IP Multicast security, Receiver access control, Secure IGMP, Group security association.

Acknowledgments

First of all, I would like to thank my supervisor for the great amount of knowledge and guidance he gave me. Secondly, I want to thank my parents. I would not be able to achieve what I have achieved without their unconditional support. Finally, I am also grateful for every help I have got.

Contents

List of Figures

List of Tables

Acronyms

AH Authentication Header. 14, 15, 17, 18, 31, 37

API Application Programming Interface. 12

CSMA/CD Carrier Sense Multiple Access with Collision Detection. 46

ESP Encapsulating Security Payload. 14, 15, 17, 18, 22, 23, 31

G-IKEv2 Group Key Management using IKEv2. 16

GDOI Group Domain of Interpretation. 16

GM Group Member. 23, 24, 25, 40, 41, 44, 45, 46, 49, 50, 52, 53, 54

GPL GNU General Public License. 19

GSA Group Security Association. 3, 16, 22, 23, 24, 38, 40, 44, 48

GSAD Group Security Association Database. 35

GSAM Group Security Association Management Protocol. iii, x, 3, 22, 23, 24, 25, 27, 28, 30, 31, 35, 36, 37, 38, 40, 43, 44, 45, 48, 49, 50, 53

GSPD Group Security Policy Database. 16, 17, 18

ICMP Internet Control Message Protocol. 4, 31

IETF Internet Engineering Task Force. 4, 6, 16, 46

IGMP Internet Group Management Protocol. iii, 2, 3, 4, 6, 8, 12, 13, 22, 23, 28, 30, 40, 44, 45, 46, 48, 53

IGMPv1 Internet Group Management Protocol Version 1. 4, 5, 6, 7, 8, 9, 10, 11, 23

IGMPv2 Internet Group Management Protocol Version 2. 4, 6, 7, 8, 9, 10, 11, 22, 23

IGMPv3 Internet Group Management Protocol Version 3. 3, 4, 8, 9, 10, 11, 12, 22, 23, 31, 32, 33, 34, 40, 43, 46, 48, 49, 52, 53

IKE Internet Key Exchange. 16, 17

IKEv2 Internet Key Exchange Version 2. 16, 23, 24, 25, 37

IPsec Internet Protocol Security. vii, 3, 14, 15, 16, 17, 18, 22, 23, 28, 31, 35, 36, 37, 46, 48

IPTV Internet Protocol Television. iii, 2, 12, 30

ISP Internet Service Provider. iii, 2

MLD Multicast Listener Discovery. 3, 22, 28

MTU Maximum Transmission Unit. 10

NQ Non-Querier. 23, 24, 27, 40, 41, 44, 45, 46, 49, 50, 52, 53, 54

PAD Peer Authorization Database. 16, 17, 35

Q Querier. 22, 23, 24, 27, 40, 45, 46, 48, 54

QoE Quality of Experience. 30

SA Security Association. 15, 16, 17, 18, 25, 27, 38

SAD Security Association Database. 16, 17, 35

SIGMP Secure Internet Group Management Protocol. iii, vii, 3, 22, 23, 24, 28, 30, 31, 32, 33, 40, 43, 44, 45, 46, 48, 50, 53

SPD	Security Policy Database. 16, 17, 18, 35
SPI	Security Parameters Index. 15, 17, 18, 24, 25, 35, 38, 40, 41, 44, 45, 46, 48, 54
STB	Set-top Box. 2
TCP	Transmission Control Protocol. 20
UDP	User Datagram Protocol. 19, 20, 36
UML	Unified Modeling Language. 31, 32

Chapter 1

Introduction

IP multicast uses a one-to-many or many-to-many model for communication. A multicast packet is only sent by the source once and it would be replicated on the nodes of the network and reach multiple receivers. On the contrary, the other method of delivery: IP unicast employs the one-to-one model. Although IP unicast is less efficient compared to IP multicast, it is more popularly used on the Internet because of easier access control. Since a unicast packet will only reach one receiver and it is specified by the sender, end-to-end encryption can be achieved. While a multicast packet is seen as being sent away from the source and replicated on the way, the information of the receivers is hidden from the sender so that the sender has no control on who will receive the packet.

For example, IPTV uses IP multicast to deliver television content to subscribers, in contrast to traditional video streaming services that use IP unicast such as Youtube and Netflix. Although, IPTV services have been used across the world and generate a large amount of revenue (Zion, 2016), due to the fact that IP multicast has no built-in security feature, it is difficult to provide an IPTV service by someone who does not have the full control of the network. In the case of IPTV service provided by ISP, the multicast packets of the video stream are sent by servers controlled by them, replicated on multicast router controlled by them and arrive to the Set-top Box (STB) provided before the content of the video stream is shown on the TV. Because the ISPs control every waypoint of their IPTV delivery, they are able to account for the amount of content watched by the users and collect revenue from them.

Before the very first frame of video of IPTV is shown, several processes of communication must be done over the network between the subscriber's STB and the sources of video stream. Joining a multicast group using IGMP is the first process.

SIGMP and GSAM (Li & Atwood, 2016) are two protocols under the architecture (Atwood, 2007) for multicasting to provide security features for IGMP and Multicast Listener Discovery (MLD) by using Internet Protocol Security (IPsec) to protect their messages. SIGMP is an extension to Internet Group Management Protocol Version 3 (IGMPv3) and MLD. GSAM is a key management protocol for Group Security Associations (GSAs) used to protect SIGMP messages. Unlike IP unicast places access control on that only the receivers with proper credential can understand the content of the packet, SIGMP and GSAM take an alternative approach of controlling the interaction between hosts and routers so that multicast packets will only reach the ones with proper credential.

However, the security features provided by SIGMP and GSAM introduces additional latency. In this thesis, we present the operational details of the SIGMP and GSAM protocol bundle as well as its performance evaluation on ns-3 network simulator. The rest of the thesis is organized as follows. For Chapters 2, 3, 4, we introduce some background information of IGMP, IPsec and ns-3. Then we demonstrate the operation details of SIGMP and GSAM in Chapter 5. The problem statement and the objective are presented in Chapter 6. In Chapter 7, we illustrate our implementation of GSAM, SIGMP and IPsec on ns-3 simulator. Furthermore, we compare the performance between the protocol bundle and IGMPv3 in Chapter 8. Finally, in Chapter 9, we come to a conclusion and offer future work.

Chapter 2

Internet Group Management Protocol

IGMPv1 (Deering, 1988), IGMPv2 (Fenner, 1997) and IGMPv3 (Kouvelas, Cain, Fenner, Deering, & Thyagarajan, 2002) are currently the three versions of IGMP standardized by the Internet Engineering Task Force (IETF). Like Internet Control Message Protocol (ICMP), IGMP is an integral part of Internet Protocol. It is required to be implemented by all hosts wishing to receive IP multicasts. Although the three versions work differently, they are backward compatible. In this chapter, we introduce all three versions of standard IGMP from the perspective of hosts, which means components that do not concern the operation of hosts may not be covered in detail.

2.1 IGMPv1

IGMPv1 is the first IGMP protocol standardized by IETF, introduced along with many other base notions of *Host Extensions of IP Multicasting* (Deering, 1988). There are only two types of messages in IGMPv1: *Host Membership Queries* and *Host Membership Report*. *Query* messages are sent by multicast routers to the local network and addressed to all-hosts group (address 224.0.0.1). Hosts respond to a *Query message* by replying a report to IP address of the specific host group on the network interface from which the query was received.

A single network interface of IGMPv1 host may transit among three possible states, with respect to any single IP host group: *Non-Member State*, *Delaying Member State* and *Idle Member State* respectively:

(1) Non-Member State: When the host does not belong to the group on the interface. This is initial state for all memberships on all network interfaces; it requires no storage in the host.

(2) Delaying Member State: When the host belongs to group on the interface and has a report delay timer running for that membership.

(3) Idle Member State: When the host belongs to the group on the interface and does not have a report delay timer running for that membership.

The following Figure 2.1 shows the relation of messages generation and transition of states:

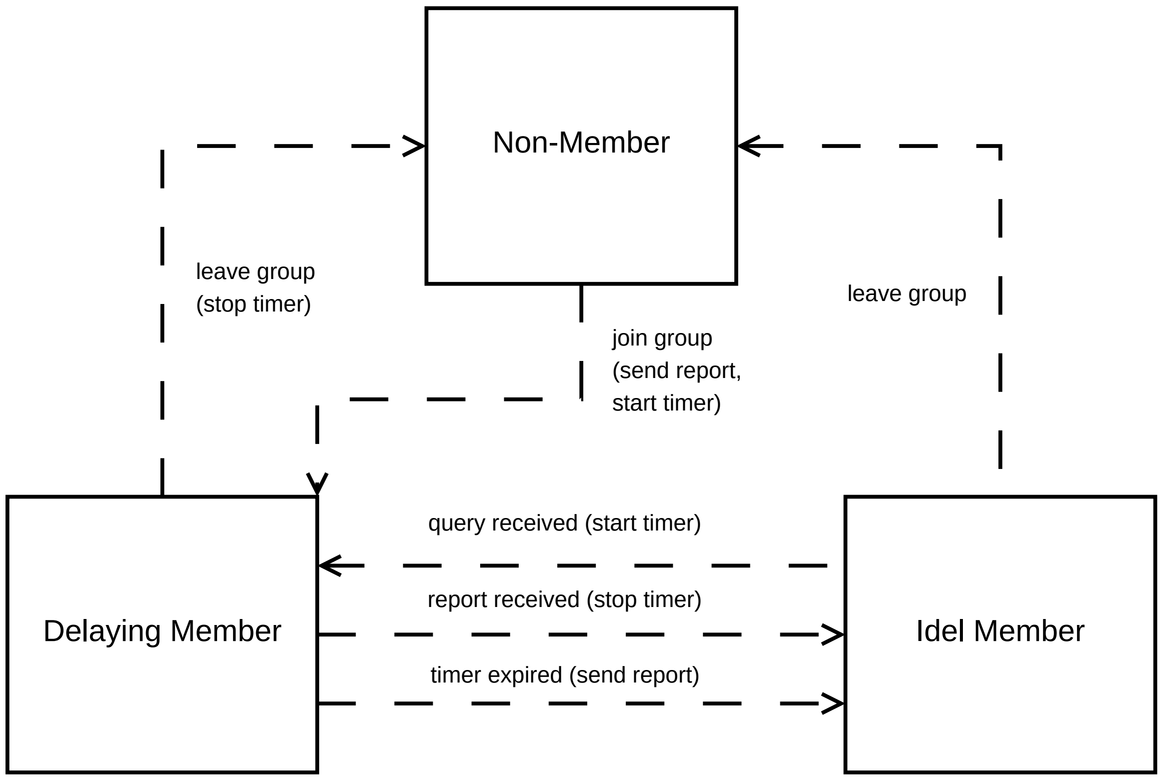


Figure 2.1: State Transistions in IGMPv1

Query messages are sent periodically by multicast routers. *Report* messages are generated when changes of interface state occur on hosts. Both *Query*and *Report* messages of concern to hosts have *time-to-live* of 1 and share the exact format shown below in Table 2.1:

The *Version* field must be 1 for both IGMPv1 *Query* and *Report* messages. The *Type* field indicates whether a message is a *Query* message or a *Report* message. It is set to 1 for *Query*

Version (4-bit)	Type (4-bit)	Unused (8-bit)	Checksum (16-bit)
Group Address (32-bit)			

Table 2.1: The Format of IGMPv1 Messages

messages and set to 2 for *Report* messages. The *Group Address* field holds the host group address of the group being reported and it is only valid in *Report* message. In *Query* messages, it should be set to 0 when sent and ignored when received.

In order to avoid an implosion of concurrent *Report* messages and to reduce the total number of *Report* messages transmitted hosts may not send reports immediately when receiving a *Query* and suppress its own action of sending *Report* message when it receives a report to the host group it is interested in from other hosts on the interface. To achieve the goal of avoiding report implosion, a delay timer per group is used by hosts. When a host receives a *Query* message, the timer for each of its group memberships is set to a random value between 0 and D seconds. (D is the maximum report delay, a parameter of the IGMPv1 that can be configured). When a timer expired, a *Report* message is generated for the corresponding host group. *Report* messages from other hosts causes hosts to stop their own timers for the reported host group upon reception and not to generate their reports for that group, with the result that only one report will be generated simultaneously on the network in the normal case.

2.2 IGMPv2

IGMPv2 is the second IGMP protocol standardized by IETF, with some changes and additional features. Firstly, a new type of message, *Leave Group* message, is added to IGMPv2, allowing hosts explicitly to leave a host group by sending the message rather than just stop responding to *Membership Query* message in IGMPv1. The group a host wishes to leave is specified by the group address in the *Leave Group* message. *Leave Group* messages are sent to IP address 224.0.0.2. Secondly, there are two kinds of *Query* messages in IGMPv2: *General Query* and *Group-Specific Query*. *Group-Specific Query* messages are used to inquire about membership to one specific host group on the attached network while *General Query* messages are for inquiring about membership in any host group. *Group-Specific Query* messages are sent to the IP address of the host group instead of 224.0.0.2 in the case of *General Query*.

Furthermore, the role of a multicast router is divided into *Querier* and *Non-Querier*. A *Querier* is a multicast router that handles *Report* messages and sends *Query* messages. A*Non-Querier* router acts as a backup for a *Querier* router. It handles *Report* messages but does not send *Query* messages. There can only be one *Querier* existing in a local network. An *Querier* election is performed when there is more than one multicast router residing in the local network.

Hosts may belong to the same three kinds of states in IGMPv2 as they do in IGMPv1. However, due to the new types of message, there are two new events that trigger state transition of a network interface on a host in addition to five in IGMPv1. The following Figure 2.2 shows messages generation and the transition of states of hosts in IGMPv2:

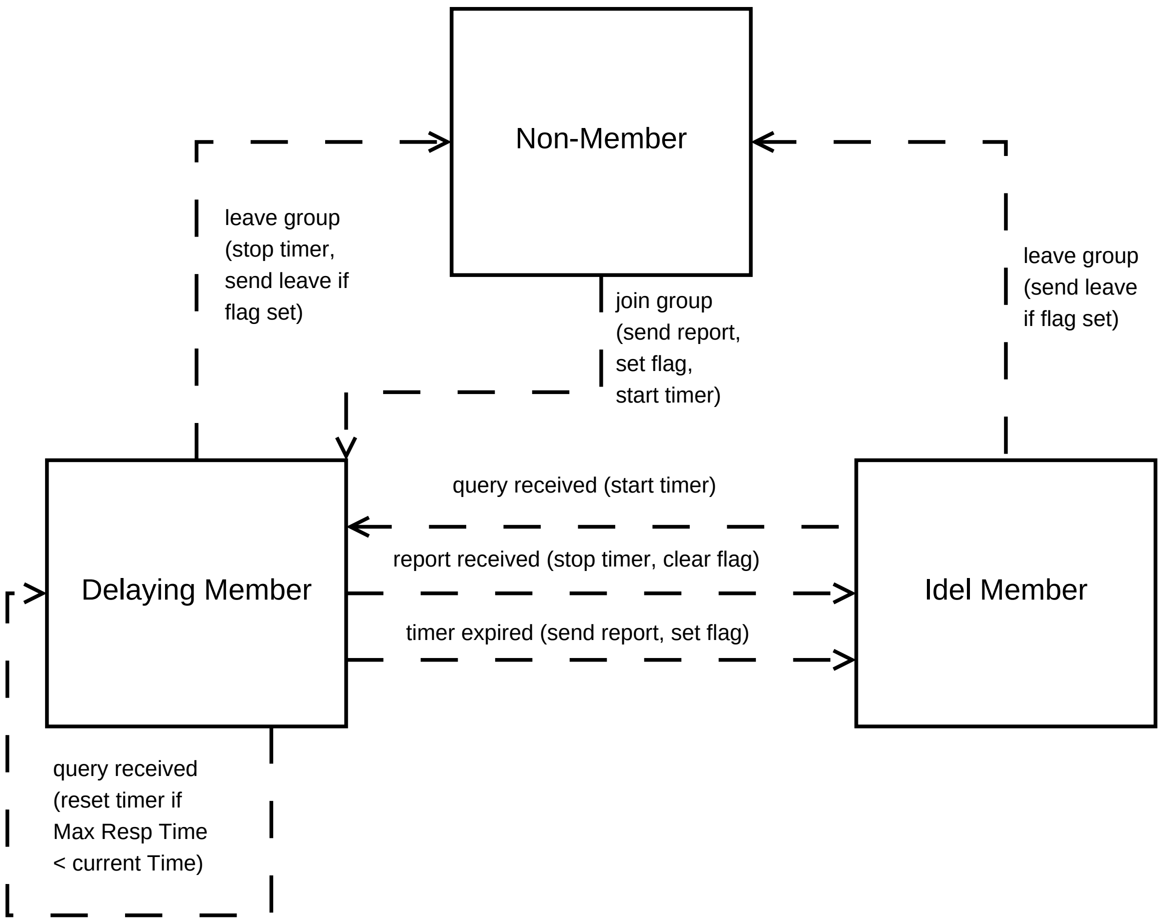

Non-Member
leave group (stop timer, send leave if flag set)
join group (send report, set flag, start timer)
leave group (send leave if flag set)
query received (start timer)
Delaying Member
report received (stop timer, clear flag)
timer expired (send report, set flag)
Idel Member
query received (reset timer if Max Resp Time < current Time)

Figure 2.2: State Transistions in IGMPv2

Moreover, the format of IGMPv2 messages is also changed stighly, as shown below in Table 2.2

Type (8-bit)	Max Resp Time (8-bit)	Checksum (16-bit)
Group Address (32-bit)		

Table 2.2: The Format of IGMPv2 Messages

Firstly, IGMPv2 messages are very similar to IGMPv1 messages. Both are 8 octets in normal cases. In the case of *General Query*, they are almost identical except that *Query* message in IGMPv2 has an additional *Max Resp Time* field defined in the unused part of IGMPv1 message. The field specifies the maximum allowed time before sending a responding report in units of $1/10$ second. This waiting time is a part of techniques used for avoiding report implosion.

Secondly, the first $8 - bit$ field of IGMPv2 packet is differently defined from IGMPv1. The first $4 - bit$ *Version* field and the following $4 - bit$ *Type* field of IGMPv1 is merged into the first single $8 - bit$ *Type* field in IGMPv2 packets. In IGMPv2, there are four types of message: *Membership Query* (*Type* $0x11$), *Version 2 Membership Report* (*Type* $0x16$), *Leave Group* (*Type* $0x17$). As we can see, IGMPv1 *Membership Query* message and IGMPv2 *Membership Query* message are identical. Though, IGMPv1 *Membership Report* (*Type* $0x12$ with respect to IGMPv2 packet format) will not be issued by IGMPv2, it can be recognized by IGMPv2 as backwards-compatibility.

Finally, new defined *Leave Group* messages are sent to 224.0.0.2 and *Group-Specific Query* messages are sent to the multicast address of the host group.

2.3 IGMPv3

IGMPv3 is the latest standardized version of the IGMP. It is very different from but also compatible with the previous two versions of IGMP. Features and differences will be introduced below.

The main added feature of IGMPv3 is the support of source filtering, giving a system the ability to report interest of a host group in receiving packets only from or all but not from specific source addresses, which is required to support Source-Specific Multicast. Source-Specific Multicast allows a receiver to specify the sources of IP packets, in contrast to the model of Any-source multicast, where any host can transmit to an host group (Bhattacharyya, 2003).

IGMPv3 databases maintain three kinds of records: *Socket State* 2.3, *Interface State* 2.4 and *Reception State* 2.5. Entries of *Socket State* and *Interface State* are used to record group memberships on IGMPv3 system on hosts. Entries of *Reception State* are used by IGMPv3 system on multicast routers for recording group memberships reported by hosts attached to the local networks.

An entry of *Socket State* is maintained for each socket and it conceptually consists of a set of records of the following form:

Interface	Multicast-address	Filter-mode	Source-list

Table 2.3: The Format of Socket State

An entry of *Interface State* is maintained for each interface and it conceptually consists of a set of the following form:

Multicast-address	Filter-mode	Source-list

Table 2.4: The Format of Interface State

An entry of *Reception State* is maintained for each group per attached network and it conceptually consists of a set of records of the following form:

Multicast-address	Group Timer	Filter-mode	Source Records (source address, source timer)

Table 2.5: The Format of Reception State

IGMPv3 is backward-compatible to IGMPv1 and IGMPv2, through downgrading the version of the protocol when IGMPv3 multicast routers receive messages of lower version protocol. The backward-compatibility requires IGMPv3 multicast routers to have implementations of IGMPv1 and IGMPv2.

Though IGMPv3 has only one main added feature and is backward-compatible to the previous versions of the protocol, it is almost completely different under the hood. Firstly, *Query* messages and *Report* messages have different formats, which are completely different from what they are in IGMPv1 and IGMPv2. Secondly, systems on hosts maintain reception states per socket in addition to Interface State. Finally, multiple timers and counters are used in IGMPv3 for source-filtering, compared with is only one timer in IGMPv1 and IGMPv2.

The following Table 2.6 shows the format of IGMPv3's *Query* messages:

Type = 0x11 (8-bit)	Max Resp Code (8-bit)	Checksum (16-bit)
Group Address (32-bit)		
Resv|S|QRV (8-bit)	QQIC (8-bit)	Number of Sources (N) (16-bit)
Source Address [1] (32-bit) Source Address [2] (32-bit) ● ● ● Source Address [N] (32-bit)		

Table 2.6: The Format of IGMPv3's Query Messages

IGMPv3 adds a third kind of *Query* message: *Group-and-Source-Specific Query*, in additional to *General Query* and *Group-Specific Query*. A *Group-and-Source-Specific Query* message is also sent to the IP address of the host group, as it is for *Group-Specific Query* messages. An IGMPv3 *Query* message may have length larger than eight octets. The length of a message depends on how many *Source Addresses* are presented. The first 8 octets of IGMPv3 *Query* messages are similar to those in IGMPv2. The *Type* field for IGMPv3 *Query* is fixed to $0x11$ as it is in IGMPv2.

The following Table 2.7 shows the format of IGMPv3's *Report* messages:

Type = 0x22 (8-bit)	Reserved (8-bit)	Checksum (16-bit)
Reserved (16-bit)		Number of Group Records (M) (16-bit)
Group Record [1] (variable length)		
Group Record [2] (variable length)		
● ● ●		
Group Record [N] (variable length)		

Table 2.7: The Format of IGMPv3's Report Messages

In IGMPv1 and IGMPv2, a host can only express its interest in membership for one group per report. In IGMPv3, however, the change of *Report*'s format allows a host to express its interest in multiple groups in one report. The destination IP address of *Report* messages is also changed to

244.0.0.22 instead of the IP address of host groups in previous two versions. The size of *Report* message is also limited by MTU of the network. If the size of a single *Report* message exceeds the limit because of too many *Group Records*, the host needs to split the *Report* message to be sent.

In IGMPv3 *Report* messages, only the first 4 octets resemble to *Report* messages in IGMPv1 and IGMPv2. Other than that, the *Number of Group Records* field and a vector of *Group Records* are presented. The *Number of Records* field must agree with the number of *Group Records* in the message.

The following Table 2.8 shows the format of IGMPv3's *Group Record* substructure in *Report* messages:

Record Type (8-bit)	Aux Data Len (8-bit)	Number of Sources (N) (16-bit)
Multicast Address (32-bit)		
Source Address [1] (32-bit) Source Address [2] (32-bit) ● ● ● Source Address [N] (32-bit)		
Auxiliary Data (variable length)		

Table 2.8: The Format of IGMPv3's Group Recrods

The fields in *Group Records* are specified in the following:

(1) *Record Type*: there are three categories of *Group Record*: *Current-State Record*, *Filter-Mode-Change Record*, *Source-List-Change Record* respectively. Each category has two types of *Group Record* so that there are six kinds of *Group Record* in total. Different types of *Group Record* report not only the membership of a host group but also the source addresses the group members wish to listen to due to source-filtering feature added in IGMPv3. All six kinds will be introduced in detail later along with changes and maintenance of reception states of host groups in IGMPv3 multicast routers.

(2) Aux Data Len: the length, in bytes, of *Auxiliary Data* presented after the vector of *Source Addresses*. This field may contain 0.

(3) Number of Sources: it indicates how many *Source Addresses* are presented in the *Group Record*.

(4) Multicast Address: it contains the IP multicast address of the host group to which the *Group Record* pertains.

(5) Source Address: the vector of *Source Addresses* lists IP unicast addresses of the sources of the multicast data stream.

(6) Auxiliary Data: this field contains additional information related to the *Group Record*, for future use. It is not defined in the IGMPv3 document and must be set to 0 when sent and ignored when received.

As there is a size limit of one single *Report* message, when a single *Group Record* contains too many source addresses that will not fit into the *Report* message, those source addresses need to be split into different *Group Records*.

Query messages are sent by a special multicast router called the *Querier*. *General Query* messages are periodically sent to request membership information from hosts in an attached network. In addition, *Group-Specific Query* and *Group-and-Source-Specific Query* messages may be sent when the router receives *Filter-Mode-Change* or *Source-List-Change* records based on how the router state is changed.

Report messages, containing group records, are sent by hosts. When an invocation of *IPMulticastListen* API by applications changes the state of an *Interface*, a *Source-List-Change Record* or *Filter-Mode-Change Record* is generated. When a host receives a *Query* messages, a *Current-State Record* is generated. *Group Records* are carried in *Report* messages and may be retransmitted in later reports to cover the possibility of reports being missed by multicast routers.

2.4 Security and Performance

Every IGMP message is directly attached to a IPv4 header and delivered to neighboring local network in plaintext. Therefore, anyone attached to the local network can receive every IGMP message. Furthermore, the *Querier* will process all the *Report* messages it receives meaning that anyone on the local network can join any group.

The process of joining a member group in IGMP is essential to IPTV process and it contributes a certain amount of latency to *Channel Zapping Time* of IPTV. In commercial IPTV, the delay of channel switching consists of three components (Manzato & da Fonseca, 2013) :

(1) Network Delay (including IGMP delay): usually shorter than 100-200ms

(2) Synchronization Delay: 500-2000ms

(3) Buffering Delay: 1-2s

As we can see, the latency caused by IGMP only occupies a small portion of *Channel Zapping Time*.

Chapter 3

IPsec

IPsec (Seo & Kent, 2005) is a suite of protocols used to protect communications over IP network, providing confidentiality, data integrity, access control, and data source authentication to IP datagrams. An IPsec system usually consists of following components:

(1) Security Protocols (Protocols which protect the IP traffic)

(2) Key Management Protocols

(3) Databases

An IPsec system acts as a boundary that filters all IP packets. When passing the boundary, an IP packet can be discarded, bypassed or be reprocessed according the rules stored in IPsec databases.

3.1 Security Protocols

Authentication Header (AH) (Kent, 2005a) and Encapsulating Security Payload (ESP) (Kent, 2005b) are the two protocols. Both AH and ESP provide their security services by either wrapping the payload of the IP packet or the entire IP packet, depending on which mode, *Transport Mode* or *Tunnel Mode*, is used. In *Transport Mode*, only the payload of the IP packet is modified and protected. In *Tunnel Mode*, the whole IP packet becomes the payload of the protect IPsec packet and a new IP header is constructed for the packet. The methods of packet transformation of AH and ESP are introduced below:

The following tables 3.1, 3.2, 3.3 show how an IPv4 packet is modified before and after applying AH in *Transport Mode* as well as in *Tunnel Mode*:

original IP header	TCP	Data

Table 3.1: Before Applying AH

original IP header	AH	TCP	Data
<-mutable field processing->	<-immutable fields->		
<- authenticated except for mutable fields ->			

Table 3.2: After Applying AH in Transport Mode

new IP header	AH	original IP header	TCP	Data
<-mutable field processing->	<- immutable fields ->			
<- authenticated except for mutable fields in new IP header ->				

Table 3.3: After Applying AH in Tunnel Mode

The following tables 3.4, 3.5, 3.6 show how an IPv4 packet is modified before and after applying ESP in *Transport Mode* as well as in *Tunnel Mode*:

original IP header	TCP	Data

Table 3.4: Before Applying ESP

original IP header	ESP header	TCP	Data	ESP trailer	ESP ICV
		<- encryption ->			
	<- integrity ->				

Table 3.5: After Applying ESP in Transport Mode

new IP header	ESP	original IP header	TCP	Data	ESP trailer	ESP ICV
		<- encryption ->				
	<- integrity ->					

Table 3.6: After Applying ESP in Tunnel Mode

The following tables 3.7, 3.8 show the data structures of AH and ESP mentioned above:

Next Header (8-bit)	Payload Len (8-bit)	RESERVED (16-bit)
Security Parameters Index (SPI) (32-bit)		
Sequence Number Field (32-bit)		
Integrity Check Value-ICV (variable length)		

Table 3.7: The Format of AH

Security Parameters Index (SPI) (32-bit)		
Sequence Number Field (32-bit)		
Payload Data (variable length)		
	Padding (0-255 bytes)	
	Pad Length (8-bit)	Next Header (8-bit)
Integrity Check Value-ICV (variable length)		

Table 3.8: Top-Level Format of an ESP Packet

For the IPsec system to process an incoming AH or ESP packet, the field of Security Parameters Index (SPI) is used along with other fields in the IP header of a packet to identify the Security Association (SA) in the IPsec system.

3.2 Key Management Protocols

Internet Key Exchange (IKE), Internet Key Exchange Version 2 (IKEv2), Group Domain of Interpretation (GDOI) and Group Key Management using IKEv2 (G-IKEv2) are a series of protocols that are used create to and manage SA in an IPsec system. IKE and IKEv2 are used to create manage unicast SA while GDOI and G-IKEv2 are used to create and manage GSA for protecting the multicast traffic.

Take IKEv2 as an example, IKEv2 is used between two entities to negotiate and create one or more pairs of unidirectional IPsec SA. The process of creating the first SA pair consists of two phases. In phase one, IKEv2 creates a bidirectional IKE SA and establishes a secure and authenticated channel for IKEv2 communication in phase two. In the second phase, the two entities negotiate what encryption method and cryptographic keys to be used, what traffic to protect. The result of such negotiation is a pair of unidirectional IPsec SAs created on both entities. In addition, more child SA pairs can be negotiated afterward.

3.3 Databases

There are three kinds of major conceptional databases in an IPsec system: The Security Policy Database (SPD) for unicast or Group Security Policy Database (GSPD) for multicast, the Security Association Database (SAD) and the Peer Authorization Database (PAD). These databases and their interfaces allow system administrators to control the processing details of all inbound and outbound IP traffic. IETF defines the general model for these databases, however, their implementations are largely a local matter.

The SPD is an ordered database. It specifies the policies that determine the disposition of all IP traffic bound or outbound from a host or security gateway. Each IPsec system must have at least one SPD. The format of SPD entries depends on implementation. Each SPD entry specifies packet disposition as *BYPASS*, *DISCARD* or *PROTECT* and it is keyed by one or more selectors such as packet's local address, remote address, next layer protocol number, etc. GSPD entries employ a different structure of IP address selectors due to the fact the symmetric relationship associated with local and remote address values is not enough. In an IPsec enabled system, SPD (or GSPD for multicast) must be consulted during processing of all IP traffic.

SAD contains parameters that are associated with each established (keyed) SA. Each SA entry defines the parameters to provide security service of either AH or ESP. For outbound processing, each SAD entry is pointed by the SPD cache. For inbound processing, the SPI alone or in conjunction with other fields in IP header is used to look up an SA.

PAD provides a link between an SA management protocol (such as IKE) and the SPD. It embodies several critical functions:

(1) Identifies the peers or groups of peers that authorized to communicate with this IPsec entity

(2) Specifies the protocol and method used to authenticate each peer

(3) Provides the authentication data for each peer

(4) Constrains the types and value of IDs that can be asserted

(5) Constrains the types and values of IDs that can be asserted by a peer with regard to child SA

creation, to ensure that the peer does not assert identities for lookup in the SPD that it is not authorized to represent, when child SAs are created

(6) Peer gateway location info

For an outbound packet, SPD (or GSPD for multicast) is consulted for looking up a matched policy entry. If no matched policy entry found or the matched match specifies the disposition of the packet is *DISCARD*, the packet would not pass through the IPsec boundary and would be dropped. If a matched policy is found and it states the disposition of the packet is *BYPASS*, the packet would pass through the IPsec boundary without any additional processing. If a matched policy is found and it states the disposition of the packet is *PROTECT*, a look up of SA entry would be performed using the link to SPD contained in the policy entry. If a matched SA entry is found, transformation of the packet would be performed and a AH or ESP header would be added to the packet. Otherwise, the packet would either be dropped or a SA negotiation process would start depending on the implementation and configuration of the IPsec system. For an inbound packet, the same processing of consulting with SPD is performed. In the case where a matched policy entry is found and the policy entry specifies the disposition of the packet is *PROTECT*, a look up of SA entry is performed using the SPI value provided in the IPsec header alone or in conjunction with other information in the IP header. If a matched SA is found and the packet is genuine, the IPsec header would be removed and the reverse transformation would be performed. Otherwise, the packet would be dropped.

Chapter 4

ns-3

We choose ns-3 as our simulation platform. On its introduction webpage: "ns-3 is a discrete-event network simulator, targeted primarily for research and educational use. ns-3 is free software, licensed under the GNU GPLv2 license, and is publicly available for research, development, and use. The goal of the ns-3 project is to develop a preferred, open simulation environment for networking research: it should be aligned with the simulation needs of modern networking research and should encourage community contribution, peer review, and validation of the software. The ns-3 project is committed to building a solid simulation core that is well documented, easy to use and debug, and that caters to the needs of the entire simulation workflow, from simulation configuration to trace collection and analysis. Furthermore, the ns-3 software infrastructure encourages the development of simulation models that are sufficiently realistic to allow ns-3 to be used as a real-time network emulator, interconnected with the real world and that allows many existing real-world protocol implementations to be reused within ns-3." (Nsnam, 2015)

4.1 Programming Model

ns-3 employs the model of Object-oriented programming. Similar entities, processes and functions are categorized into an ns-3 module. A module of ns-3 contains definition and declaration of one or more C++ classes that compose a protocol in the network stack, abstraction of a physical entity or a software component in a network system or a utility for simulation. For example, the

UDP modules contains classes of UDP header, UDP protocol as well as data structures of UDP sockets. There are three most frequently used classes of ns-3 infrastructure in this project: *ns3::Ptr*, *ns3::Object* and *ns3::Header*. Firstly, *ns3:Ptr* is ns-3's built-in smart point class, providing functions of auto resource recycling and bounds checking. Secondly, objects of classes that inherit *ns3::Object* class can be referred to by *ns3::Ptr* pointers. Thirdly, objects of classes that inherit *ns3::Header* class can be easily aggregated to objects of *ns3::Packet* class using *ns3::Packet::AddHeader()* method.

4.2 Abstraction of a Host

As shown in Figure 4.1, in ns-3, a network host is modeled as an object of *ns3::Node* class. Installing a network stack on a network host means aggregating various objects of classes representing different components of the network stack into an object of *ns3::Node*.

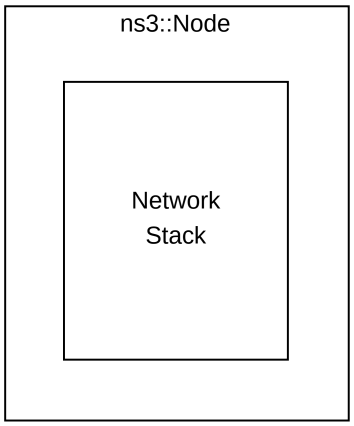


Figure 4.1: An Object of ns3::Node

4.3 Abstraction of IP Network Stack

As we can see from Figure 4.2, ns-3 has a well layered model of IPv4 communication. Each compartment holds the name of a ns-3 built-in class that represents a component of network system of an entity. Firstly, class *ns3::Socket* represents Socket of operating system. Class *ns3::Ipv4L4Protocol* presents forth layer's protocols in the network stack. For example, the built-in classes for TCP, UDP and ICMPv4 are all subclasses of *ns3::Ipv4L4Protocol*. Class *ns3::Ipv4L3Protocol* and class *ns3::Ipv4Interface* represent Internet Protocol itself and the descriptor of interface in operating system respectively. Furthermore, class *ns3::NetDevice* represents physical devices. Finally, network packets or data-link frames are represented by class *ns3::Packet*. The physical link that packets transmit through are modeled as class *ns3::Channel*. *ns3::Socket*, *ns3::Ipv4L4Protocol* and *ns3::NetDevice* are abstract classes. Classes derived from these three classes can be seamlessly integrated into the communication model.

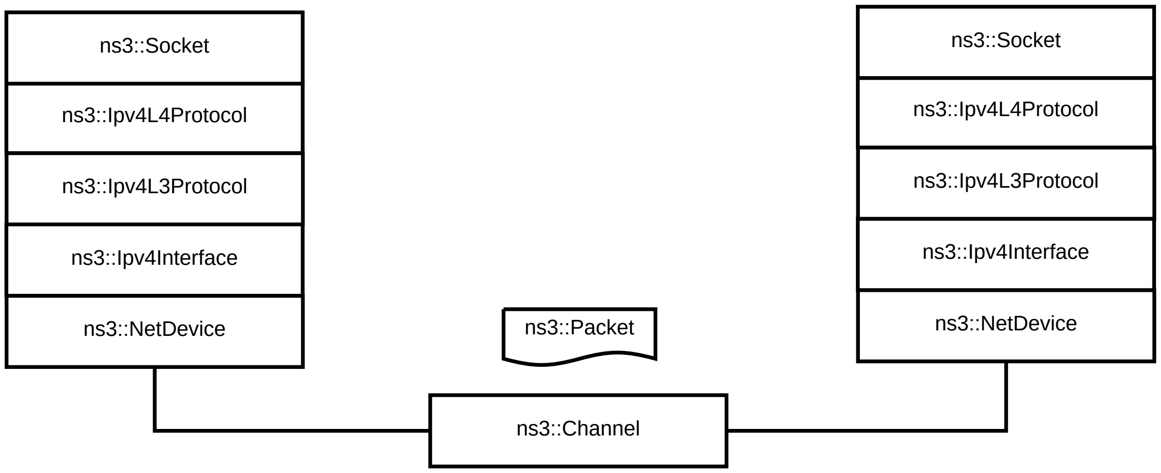


Figure 4.2: ns-3 IP Stack

Chapter 5

SIGMP and GSAM

SIGMP and GSAM is an approach (Li & Atwood, 2016), within the Secure Architecture (Atwood, 2007), to provide security service for IGMP/MLD messages so that revenue collection is possible.

The generation of the delivery tree of a multicast stream depends on information of multicast group memberships in the IGMP/MLD database. The SIGMP and GSAM protocol bundle uses IPsec to provide per group access control over IGMP/MLD messages without modifying any existing protocol. A user with a proper credential for a given secure group can negotiate a GSA pair with Q . Then a SIGMP *Report* message for that group protected by such GSA from the user can safely pass through the IPsec boundary on Q , so that the user can successfully join the group. Although SIGMP and GSAM can be applied to both IGMP and MLD , the rest of this chapter focuses on IGMP.

5.1 SIGMP

SIGMP is an extension based on IPsec for IGMPv2, IGMPv3 and MLD , offering the following additional services:

(1) Compatibility: As an extension of IGMP, SIGMP employs identical packet format of regular IGMP message so that it can communicate to entitles that do not have SIGMP support. SIGMP also does not affect the operation of IGMP databases.

(2) Confidentiality: SIGMP messages are protected by an ESP header.

(3) Authentication: Corresponding GSAs used by ESP is correlated with an application-level credential by GSAM so that protected SIGMP messages are authenticated.

Identical to IGMP, there are three kinds of roles in SIGMP, Q , NQ and GM . Q and NQ are multicast routers. As an IGMP extension, SIGMP enabled hosts should provide functionalities of IGMPv1, IGMPv2 and IGMPv3. Every SIGMP message is an enveloped ESP message of IPsec and protected by a pair of GSAs. This means performance overhead will be introduced because the process of GSAM must be done before sending the SIGMP message. Therefore, an SIGMP enabled host should not include membership of secure groups in any *Report* message for responding to a *General Query message*. The differences between IGMPv3 Message and SIGMP Message are shown in following Table 5.1

		IGMPv3	SIGMP
GQ	Sender	Q	Identical with IGMPv3
	Receiver	All EUs and NQs	
	Destination Address	244.0.0.1	
GSA	Sender	Q	Identical with IGMPv3
	Receiver	EUs that have joined the group and NQs	
	Destination Address	IP Address of the Group	
GSSQ	Sender	Q	Identical with IGMPv3
	Receiver	EUs that have joined the group and NQs	
	Destination Address	IP Address of the Group	
Report	Sender	EU	EU
	Receiver	Q and NQs	Q and NQs
	Destination Address	224.0.0.22	IP Address of the Group

Table 5.1: IGMPv3 Message vs SIGMP Message

5.2 GSAM

GSAM is a key management protocol to establish the parameters for the GSA s used by SIGMP participants. The operation of GSAM is divided into two phases of message exchanges. Like IKEv2 (Kivinen, Hoffman, Kaufman, Nir, & Eronen, 2014), a message exchange in GSAM consists of a pair of messages: a request and a response. Considering a message can be lost during the transmission, retransmission of a request message might take place. One must reply with a response message upon receiving a request message. However, a response message should never be transmitted twice.

The first phase consists of two message exchanges:

(1) Init Exchange: The purpose of this exchange is negotiation of SPI values and cryptographic keys. A single *Init SA* of GSAM is established on each endpoint by this exchange.

(2) Auth Exchange: Using the secure channel protected by the *Init SAs*, two endpoints further negotiate other parameters and exchange id information. Additional to *Init SA*, an *Auth SA* of GSAM is established on each endpoint by this exchange.

By the end of the first phase, an secure and authenticated channel for communication is established between two participants, which can either be a Q and a GM or a Q and an NQ . The channel protected by the *Auth SA* is used for communication in phase two.

The second phase consists of the following exchanges and messages:

(1) GSA Distribution Exchange: The Q distributes a pair of GSAs it generates to the GM that is joining the group and all NQs through a *GSA Push* message. Whoever receives it should respond a GSA with *Ack* message or *SPI Rejection* message if there is a SPI conflict.

(2) SPI Resolve Exchange: If there is a SPI conflict, Q will send a *SPI Request* message to the GM and all NQs to gather their currently occupied SPI values. Whoever receives it should respond with a *SPI Report* message containing their used SPI values.

(3) GSA Repush Message: After the *SPI Resolve Exchange*, the Q sends a *GSA Repush* message containing a pair of GSAs with revised SPI values.

The purpose of phase two message exchanges is distribution of GSAs. Since the SPI values of GSAs are solely determined by the Q, they might conflict with the ones being used on other hosts. A pair of GSAs consists of a *GSA_Q* and *GSA_R*. *GSA_Qs* is used to protect *Query* messages of SIGMP and *GSA_Rs* is used for *Report* messages. Therefore, a *GSA_Q* may be rejected by a GM or an NQ while a *GSA_R* can only be rejected by an NQ.

In this thesis, the format of GSAM messages are very similar to IKEv2's shown in Table 5.2.

GSAM reuses the following data structures of IKEv2 messages. Important fields will be explained below:

IKEv2 Header (28-byte)
Generic Payload Header (4-byte)
Payload Substructure (variable length)

Table 5.2: The Format of IKEv2 Messages

(1) The header of IKEv2 messages:

IKE SA Initiator's SPI (64-bit)				
IKE SA Responder's SPI (64-bit)				
Next Payload (8-bit)	MjVer (4-bit)	MnVer (4-bit)	Exchange Type (8-bit)	Flags (8-bit)
Message ID (32-bit)				
Length (32-bit)				

Table 5.3: The Format of IKEv2 Header

Fields *IKE SA Initiator's SPI* and *IKE SA Responder's SPI* specify the pair of SPI values to be used to indentify a unique IKEv2/GSAM SA. The *Initiator's SPI* is chosen by the intiator of a certain phase of the process, for example, the GM during the first phase of GSAM negotiation. Then, the *Responder's SPI* is chosen by the responder. Because the messages of IKEv2 and GSAM employ a chain-like structure of payloads, the field *Next Payload* is used to indicate the type of payload that immediately follows the current data structure. In the case of message header, it indicates the type of first payload of the message. *MjVer* is major Version and *MnVer* is short for Minor Version. These two fields indicate the version of the protocol. *Exchange Type* indicates the type of exchange being used. IKEv2 defines four values for this field, shown in Table 5.4. In the case of GSAM, messages' *Exchange Type* are set to: *IKE_SA_INIT* in *Init Exchange* during phase one, *IKE_AUTH* in *Auth Exchange* during phase one, *CREATE_CHILD_SA* for *GSA Push* and *INFORMATIONAL* during the rest of the period of the process. The field Flags specifies whether the message is from the initiator or is a respose, as well as indicates that the transmitter is capable of speaking a higher major version of the protocol. *Message ID* is the message identifier used to control retransmission of lost packets and matching of requests and responses. *Length* indicates the length of the total message including the header.

(2) The data structure *Generic Payload Header*:

Exchange Type	Value
IKE_SA_INIT	34
IKE_AUTH	35
CREATE_CHILD_SA	36
INFORMATIONAL	37

Table 5.4: The Defined Values of Exchange Type

Next Payload (8-bit)	Critical (1-bit)	RESERVED (7-bit)	Payload Length (16-bit)

Table 5.5: The Format of IKEv2 Generic Payload Header

The field *Next Payload* here has the indentical meaning as it is in the header of the message. The *Critical* bit is set by the sender to let the recipient know whether it wants the payload skipped if unrecognized.

(3) The data structure of the *Security Association Payload Substructure*. The *Proposals* also use a chain-like structure as the payloads of the message do:

<Proposals >(variable length)

Table 5.6: The Format of IKEv2 Security Association Payload Substructure

(4) The data structure of the *Proposal Substructure* in *Security Association Payload Substructure*:

Last Substruc (8-bit)	RESERVED (8-bit)	Payload Length (16-bit)	
Proposal Num (8-bit)	Protocol ID (8-bit)	SPI Size (8-bit)	Num Transforms (8-bit)
SPI (variable length)			
<Transforms >(variable length)			

Table 5.7: The Format of IKEv2 Proposal Substructure

The *Last Substruc* field is set to 0 if this was the last *Proposal Substructure*, and a value of 2 if there are more *Proposal Substructures*. The *Protocol ID* has three possible values, listed in Table 5.8.

(5) The data structure of the *Identification Payload Substructure*:

(6) The data structure of the *Authentication Payload Substructure*:

Protocol	Protocol ID
IKE	1
AH	2
ESP	3

Table 5.8: The Defined Values of Protocol ID

ID Type (8-bit)	RESERVED (24-bit)
Identification Data (variable length)	

Table 5.9: The Format of IKEv2 Identification Payload Substructure

Auth Method (8-bit)	RESERVED (24-bit)
Authentication Data (variable length)	

Table 5.10: The Format of IKEv2 Authentication Payload Substructure

(7) The data structure of the *Nonce Payload Substructure*:

Nonce Data (variable length)

Table 5.11: The Format of IKEv2 Nonce Payload Substructure

(8) The data structure of the *Traffic Selector Payload Substructure*:

Number of TSs (8-bit)	RESERVED (24-bit)
<Traffic Selectors >(variable length)	

Table 5.12: The Format of IKEv2 Traffic Selectors Payload Substructure

(9) The data structure of the *Traffic Selector Substructure* in *Traffic Selector Payload Substructure*:

TS Type (8-bit)	IP Protocol ID (8-bit)	Selector Length (16-bit)
Start Port (16-bit)		End Port (16-bit)
Starting Address (variable length)		
Ending Address (variable length)		

Table 5.13: The Format of IKEv2 Traffic Selectors Substructure

Moreover, the following new data structures are defined for GSAM messages:

(1) The data structure of the *Group Security Association Payload Substructure*:

GSA Push Id (32-bit)
<Source Traffic Selector >(variable length)
<Destination Traffic Selector >(variable length)
<Proposals >(variable length)

Table 5.14: The Format of GSAM Group Security Association Payload Substructure

GSA Push Id is used to indentify a specific phase two exchange because all phase two exchanges between the Q and an NQ are protected by the same pair of GSAM SA.

(2) The data structure of the *Group Proposal Substructure* in *Group Security Association Payload Substructure*:

0 (last) or 2 (not last) (8-bit)	GSA Type (8-bit)	Payload Length (16-bit)	
Proposal Num (8-bit)	Protocol ID (8-bit)	SPI Size (8-bit)	Num Transforms (8-bit)
SPI (variable length)			
<Transforms >(variable length)			

Table 5.15: The Format of GSAM Group Proposal Substructure

(3) The data structure of the *Group Notify Payload Substructure*:

Protocol ID (8-bit)	SPI Size (8-bit)	Notify Msg Type (8-bit)	Number of SPIs (8-bit)
GSA Push Id (32-bit)			
<Source Traffic Selector >(variable length)			
<Destination Traffic Selector >(variable length)			
<SPIs >(variable length)			

Table 5.16: The Format of GSAM Group Notify Payload Substructure

5.3 Summary

In summary, the SIGMP and GSAM protocol bundle is an approach that provides security features for IGMP/MLD. In addition to the provided security features, the protocol bundle also has the following performance advantages:

(1) No modification to existing protocols: By using IPsec, the additional security features are transparent to IGMP/MLD.

(2) Integration: Although an extra key management protocol is introduced, no addition physical entity is required for the operation of the protocol bundle because the roles in key management are mapped to roles in IGMP/MLD.

Chapter 6

Problem Statement

The SIGMP and GSAM protocol bundle works within the secure IP multicast architecture to enforce receiver access control, and more importantly to allow the generation of revenue. The security of SIGMP and GSAM protocol bundle has been validated (Li & Atwood, 2016). However, this security feature adds extra latency to the process of joining a group. Latency is one of the factors of Quality of Experience (QoE). QoE affects a customer's decision about purchasing a service. Therefore, the performance of SIGMP and GSAM protocol bundle needs to be evaluated.

IGMP is one of the primary underlying protocols of IPTV service. IPTV is a very good commercial use case to evaluate the performance of the protocol bundle because it is a paid service and a user is sensitive to the time of channel changing. Changing channels in IPTV requires IGMP to join multicast groups, which contributes a certain amount of latency to the total time of channel changing.

The goals of this work are to analyze and evaluate the performance through simulation on ns-3 and to propose improvement based on analysis and results of simulation.

Chapter 7

Implementation of SIGMP and GSAM on ns-3

Although ns-3 provides a solid pack of infrastructure modules for simulation, it lacks implementation of IGMPv3 and IPsec. Therefore, modules of IGMPv3 and IPsec need to be developed prior to implementation of SIGMP and GSAM. Although encryption and authentication is used to protect SIGMP and GSAM messages, it is not vital in simulation. In this implementation, all messages are transmitted in plain text. Moreover, we use a simplified AH instead of the required ESP for simplicity. In this chapter, we will introduce the model of implementation of the protocol bundle on ns-3 with Unified Modeling Language (UML). The implementation consists of the following five modules:

(1) IGMPv3 and SIGMP protocols module 7.2

(2) IGMPv3 data structures and databases module 7.3

(3) IPsec and GSAM databases module 7.4

(4) GSAM protocol module 7.5

(5) GSAM and simple AH data structures module 7.6

ns-3's website provides a tutorial for using the simulator and a complementary wiki for related contents including some How-to articles and links for development. However, the resource for developers is limited. In this thesis, we use the exisiting ns-3 modules , like ICMP, as examples for developing our own modules. Moreover, to examine the correctness of our custom modules, we use debugging tools to check intermediate outputs and log files for final outputs.

7.1 UML

UML (Unified Modeling Language, 2017) is a general-purpose, developmental, modeling language in the field of software engineering, which is intended to provide a standard way to visualize the design of a system.

In object oriented programming, UML class diagram (Class diagram, 2017) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

In the diagram, classes are represented with boxes that contain three compartments:

(1) The top compartment contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.

(2) The middle compartment contains the attributes of the class. They are left-aligned and the first letter is lowercase.

(3) The bottom compartment contains the operations the class can execute. They are also left-aligned and the first letter is lowercase.

Lines and arrows represent relationship between classes, Figure 7.1 shows the representations:

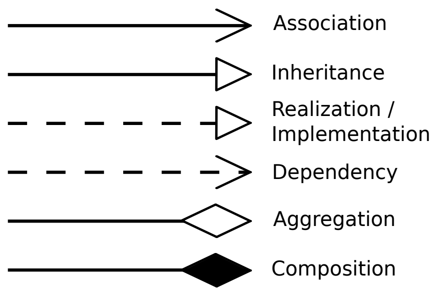


Figure 7.1: Notation of Lines and Arrows in UML

In the rest of this chapter, for readability, we only show the top compartment of UML figures of classes of the implementation. Full versions of those figures are included in Appendix A.

7.2 IGMPv3 and SIGMP Protocols Module

The module of IGMPv3 and SIGMP protocols shown in Figure 7.2 only has one class: *ns3::Igmpv3L4Protocol*. It defines the operation of sending and receiving IGMPv3 as well as SIGMP messages. Since SIGMP only provides functions of sending secure report to the group address and not responding to general query, several extra class member methods are sufficient. The class *ns3::Igmpv3L4Protocol* inherits the *ns3::IpL4Protocol* class so that objects of this class can be integrated into IP network stack on hosts in simulation. Each IP network stack has exactly one object of class *ns3::Igmpv3L4Protocol*.

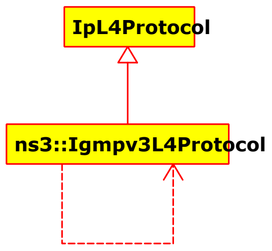


Figure 7.2: Implementation of IGMPv3 and SIGMP Protocols

7.3 IGMPv3 Data Structures and Databases Module

This module can be divided into two parts. The first part consists of classes of IGMPv3 data structures. The second part of the module includes the classes of IGMPv3 databases. There are four classes, shown in Figure 7.3, in the first part of the module, which represent the corresponding data structures: *Query* (Table 2.6), *Report* (section 2.7) and *Group Record* (Table 2.8) as well as the header of messages in IGMPv3. All of them inherit *ns3::Header* class of ns-3 infrastructure so that objects of these classes can be aggregated into objects of *ns3::Packet* during the simulation.

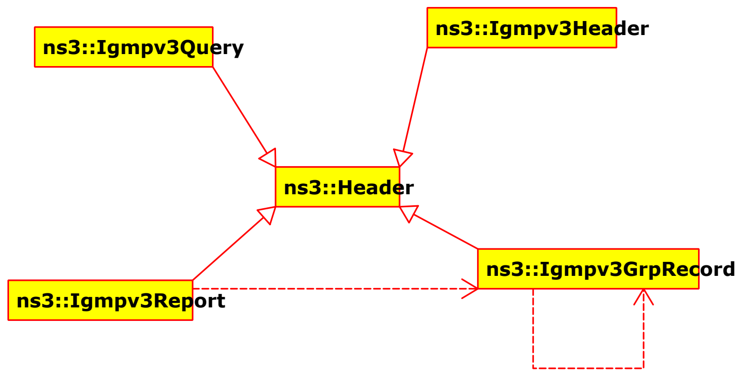


Figure 7.3: Implementation of IGMPv3 Data Structures

Figure 7.4 shows implementation of IGMPv3 databases in UML.

Firstly, each object of class *ns3::IGMPv3InterfaceStateManager* represents a per interface database of IGMPv3's *Interface State*. Each object of class *ns3::IGMPv3SocketStateManager* represents a per socket database of IGMPv3's *Socket State*. An object of class *ns3::Igmpv3Manager* on each IGMPv3 system provides the functionality of an access interface. Each objects of class *ns3::IGMPv3SocketState* represents the multicast reception state (Table 2.3) for a specific socket on a host. Each object of *ns3::IGMPv3InterfaceState* represents a per-interface multicast reception state (Table 2.4) on a host. Objects of classes *ns3::IGMPv3MaintenanceState* and *ns3::IGMPv3MaintenanceSrcRecord* represent the desired reception states (Table 2.5) for attached networks by a multicast router. Secondly, these states and actions caused by changes of these states are managed by *ns3::IGMPv3InterfaceStateManager*. These manager and state classes also contain various timers defined in IGMPv3. Finally, all aforementioned classes inherit *ns3::Object* class so that objects of them can be referenced by ns-3 internal smart pointers of *ns3::Ptr*.

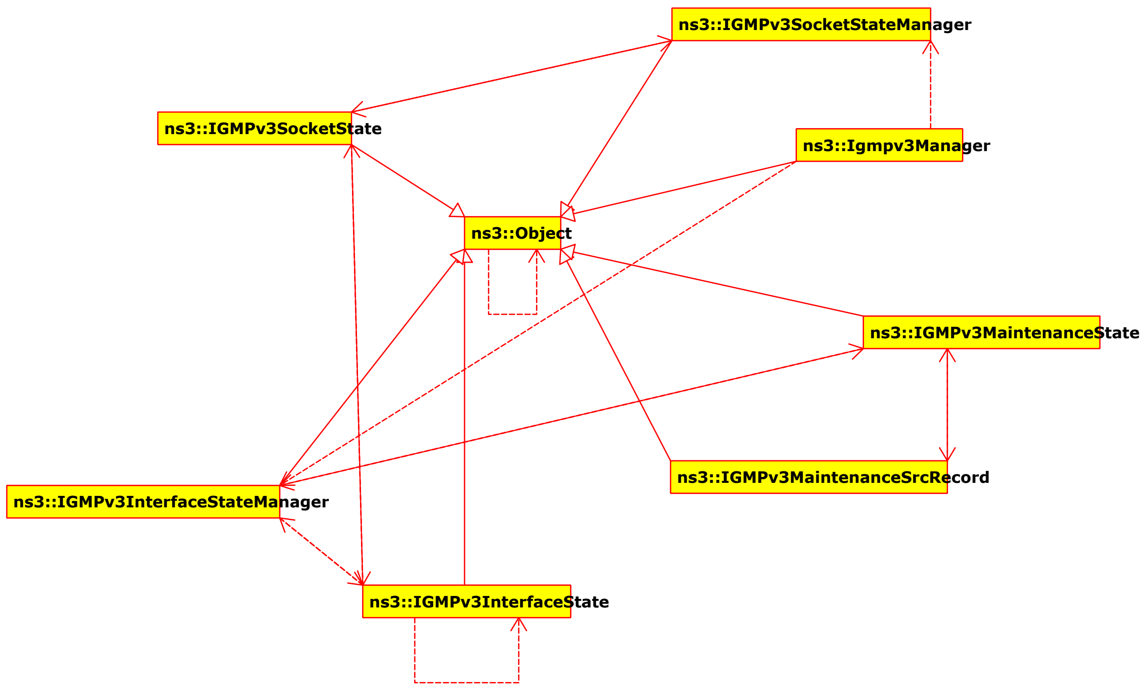

ns3::IGMPv3SocketStateManager
ns3::IGMPv3SocketState
ns3::Igmpv3Manager
ns3::Object
ns3::IGMPv3MaintenanceState
ns3::IGMPv3MaintenanceSrcRecord
ns3::IGMPv3InterfaceStateManager
ns3::IGMPv3InterfaceState

Figure 7.4: Implementation of IGMPv3 Databases

7.4 IPsec and GSAM Databases Module

Figure 7.5 shows the implementation of IPsec and GSAM databases.

For IPsec, we implement three databases, SPD, SAD and PAD respectively. For simplicity, our implementation of SAD also provides the functions of Group Security Association Database (GSAD). Each network host has one object of *ns3::IpSecDatabase* class, one object of *ns3::IpSecPolicyDatabase* class and multiple object of *ns3::IpSecSADatabase* class. The object of *ns3::IpSecPolicyDatabase* represents the SPD on the system. The ones of *ns3::IpSecSADatbase* represent the SADs. While the PAD is not modeled as an object of an independent class, its functions are integrated into the object of *ns3::IpSecDatabase*. Furthermore, the object of *ns3:IpSecDatabase* also provides access interfaces to the objects that represent the SPD and SADs. Objects of classes *ns3::IpSecPolicyEntry* and *ns3::IpSecSAEntry* represents the entries of policies and security associations. Their formats are shown in Table 7.1 and Table 7.2.

The functions of GSAM databases are provided by the object of *ns3:IpSecDatabase*. Objects of *ns3::GsamInitSession* class store information of init exchange of phase one. Objects of *ns3::GsamSession* store the information of auth exchange in phase one and message exchanges in phase two. Several objects of *ns3::GsamSession* can be derived from one object of *ns3::GsamInitSession* because the per group control takes place in auth exchange and message exchanges in the second phase and we want to reuse the channels established in init exchanges in phase one. Each object of *ns3::GsamInitSession* and *ns3::GsamSession* has one pair of objects of *ns3::GsamSa* storing the SPI pair generated in init and auth exchanges and other relevant information. Objects of *ns3::GsamSessionGroup* are used to provide access interface to sessions related to the same multicast group on hosts.

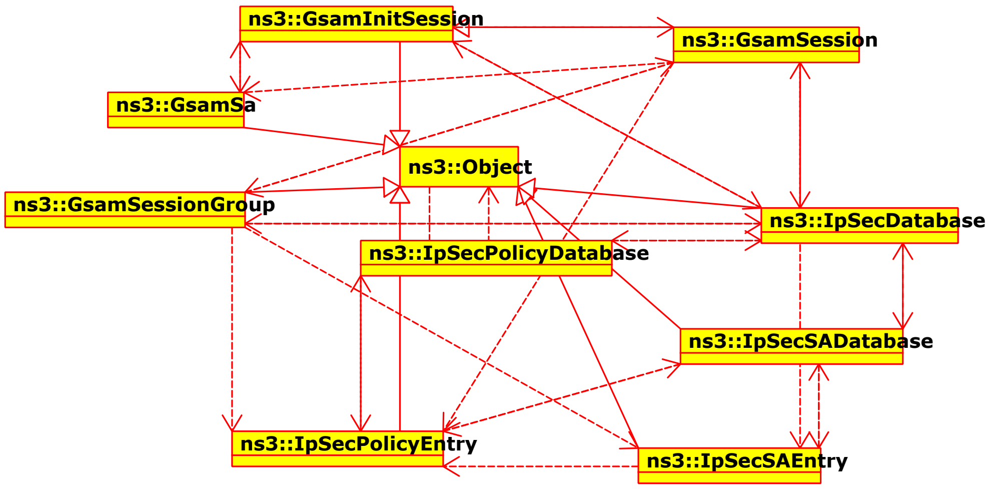


Figure 7.5: Implementation of IPsec and GSAM Databases

range of source addresses	range of destination addresses	IP protocol number
bypass, discard or protect	range of source ports	range destination ports
IPsec mode	link to incoming SAD	link to outgoing SAD

Table 7.1: The Format of SPD Entries

direction of the packet	SPI value	encrption function

Table 7.2: The Format of SAD Entries

7.5 GSAM Protocol Module

Figure 7.6 shows the implementation of this module. This module also has only one class *ns3::GsamL4Protocol*, which defines the operation of message exchanges in GSAM. The class *ns3::GsamL4Protocol* inherits *ns3::Object* but not *ns3::IpL4Protocol* because GSAM is an application protocol and it uses UDP.

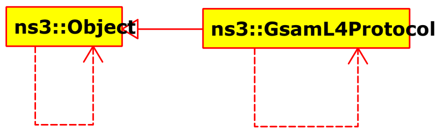


Figure 7.6: Implementation of GSAM Protocol

7.6 GSAM and Simple AH Data Structures Module

This module consists of classes of various data structures of GSAM packets as well as the data structure of a simplified AH.

Figure 7.7 shows the implementation a simplified AH. The class *ns3::SimpleAuthenticationHeader* also inherits ns-3 infrastructure class *ns3::Header*. An object of *ns3::SimpleAuthenticationHeader* will be added to or removed from the packet when a protected packet passes through the IPsec boundary.

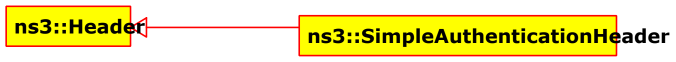


Figure 7.7: Implementation of Simple AH

Figure 7.8 shows the implementation of data structures of GSAM packets as well as IKEv2's, because GSAM reuses some of IKEv2's data structures. A GSAM/IKEv2 packet consists of an object of *ns3::IkeHeader* and an object of *ns3::IkePayload*. They can be directly aggregated into an ns-3 packet since both of the classes inherit *ns3::Header* infrastructure class. Each *ns3::IkePayload* object is composed by an *ns3::IkePayload* object and an object of *ns3::IkePayloadSubstructure*, which has several derived classes used for different message exchanges (sectiion 5.2):

(1) Objects of *ns3::IkeSaPayloadSubstructure* are used to carry the information of GSAM SA in phase one of GSAM.

(2) Objects of *ns3::IkeIdSubstructure* are used to carry the identity information during the auth exchange in phase one of GSAM.

(3) Objects of *ns3::IkeAuthSubstructure* are used to carry authentication information during auth exchange in first phase of GSAM.

(4) Objects of *ns3::IkeNonceSubstructure* are used to carry a random number during the init exchange in phase one of GSAM.

(5) Objects of *ns3::IkeTrafficSelectorSubstructure* are used to carry information of traffic selectors in both phase of GSAM.

(6) Objects of *ns3::IkeGsaPayloadSubstructure* are used to carry the information of GSA during the *GSA Distribution* in the second phase of GSAM.

(7) Objects of *ns3::IkeGroupNotifySubstructure* are used to carry the list of SPIs used on hosts in the second phase of GSAM.

Moreover, the following data structures are also used:

(1) Objects of *ns3::IkeSaProposal* represent the *SA Proposals* attached to *Security Association Payloads*.

(2) Objects of *ns3::IkeGsaProposal* represent the *Group SA Proposals* attached to *Group Security Association Payloads*.

(3) Objects of *ns3::IkeTrafficSelector* are used to store infomation of traffic selectors in *Traffic Selector Payloads*, *Group Security Association Payloads* and *Group Notify Payloads*.

Because of the similarity of format between *Security Association Payload* and *Group Security Association Payload* as well as between *SA Proposal* and *GSA Proposal*. *ns3::IkeSaPayloadSubstructure* is the base class of *ns3::IkeGsaPayloadSubstructure* and so as *ns3::IkeSaProposal* to *ns3::IkeGsaProposal*.

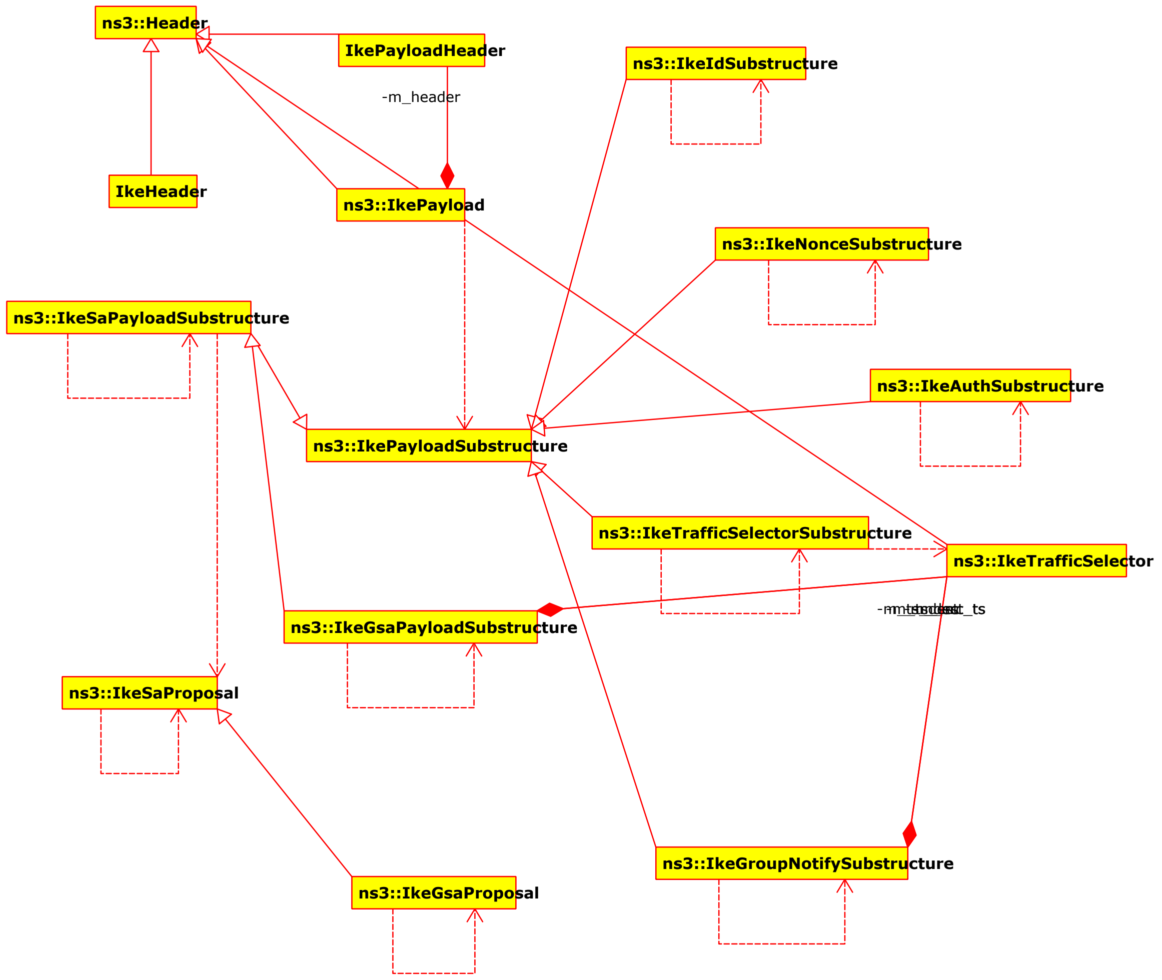

ns3::Header
IkePayloadHeader
-m_header
ns3::IkeIdSubstructure
IkeHeader
ns3::IkePayload
ns3::IkeNonceSubstructure
ns3::IkeSaPayloadSubstructure
ns3::IkeAuthSubstructure
ns3::IkePayloadSubstructure
ns3::IkeTrafficSelectorSubstructure
ns3::IkeTrafficSelector
ns3::IkeGsaPayloadSubstructure
ns3::IkeSaProposal
ns3::IkeGroupNotifySubstructure
ns3::IkeGsaProposal

Figure 7.8: Implementation of Data Structures of GSAM Packets

Chapter 8

Analysis and Evaluation of Performance

8.1 Message Sequences in Different SPI Conflicts

There are three kinds of role in IGMP: Q, NQ and GM. A Q periodically sends *General Query* messages and processes incoming *Report* messages from attached networks. There should always be one Q in a network segment. NQs act as backups of the Q. An NQ also receives and processes incoming *Report* messages but never sends any *Query* messages. A GM sends a *Report* message when it wishes to join a group.

Moreover, the Q is also responsible for distributing GSA pairs and resolving SPI conflicts. To join a secure group in SIGMP, a GM must do a series of message exchanges of GSAM with the Q before sending the IGMP *Report* message.

We compare the process of joining an open group in IGMPv3 and the process of joining a secure group in SIGMP. Though there can be multiple GMs and NQs, the scenario of the comparison is set as the following: In a simple network segment, there are one Q, one NQ and two GMs. One GM (*GM1*) performs a regular IGMPv3 join and the other one performs an SIGMP join. Moreover, depending on the location of SPI conflict, there are five different cases of sequences of message exchanges in our comparison:

(1) No SPI conflict takes place after *GSA Distribution*, shown in Figure 8.1

(2) SPI conflict from GM, shown in Figure 8.2

(3) SPI conflict of only *GSA R* from NQs, shown in Figure 8.3

(4) SPI conflict of *GSA Q* or both from NQs, shown in Figure 8.4

(5) SPI conflict from both GM and NQs, shown in Figure 8.5

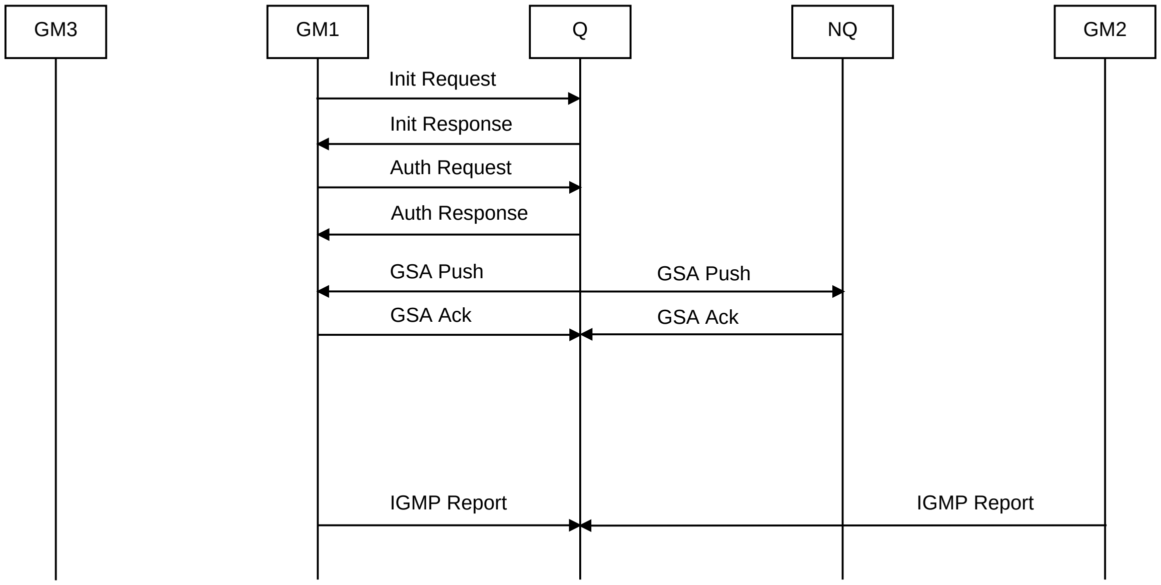


Figure 8.1: Packet Sequence When No SPI Conflict Happens

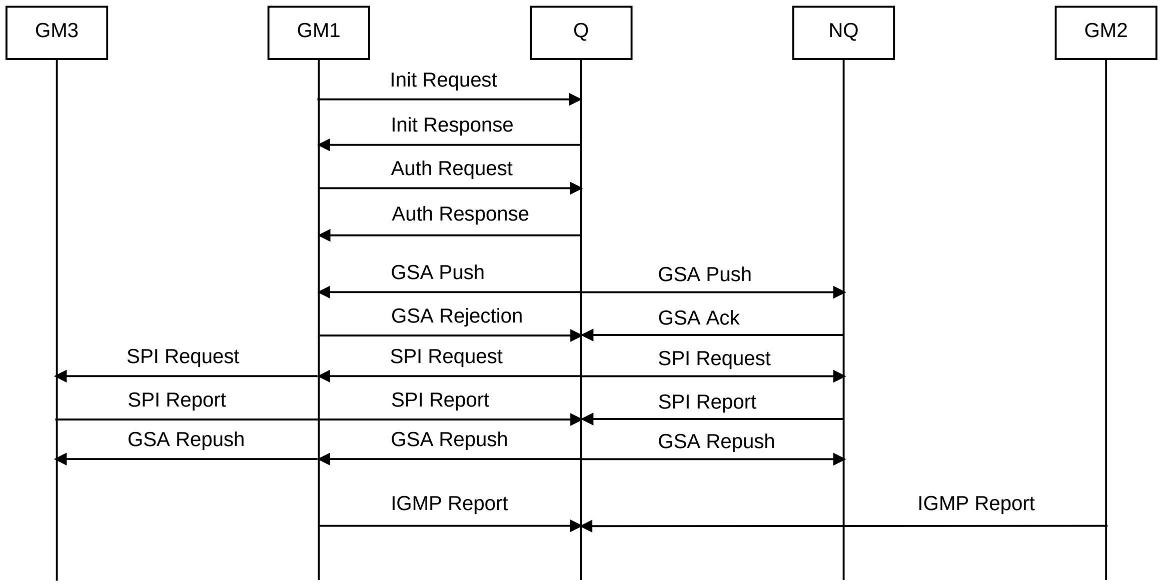


Figure 8.2: Packet Sequence When GSA Q Is Rejected by the GM

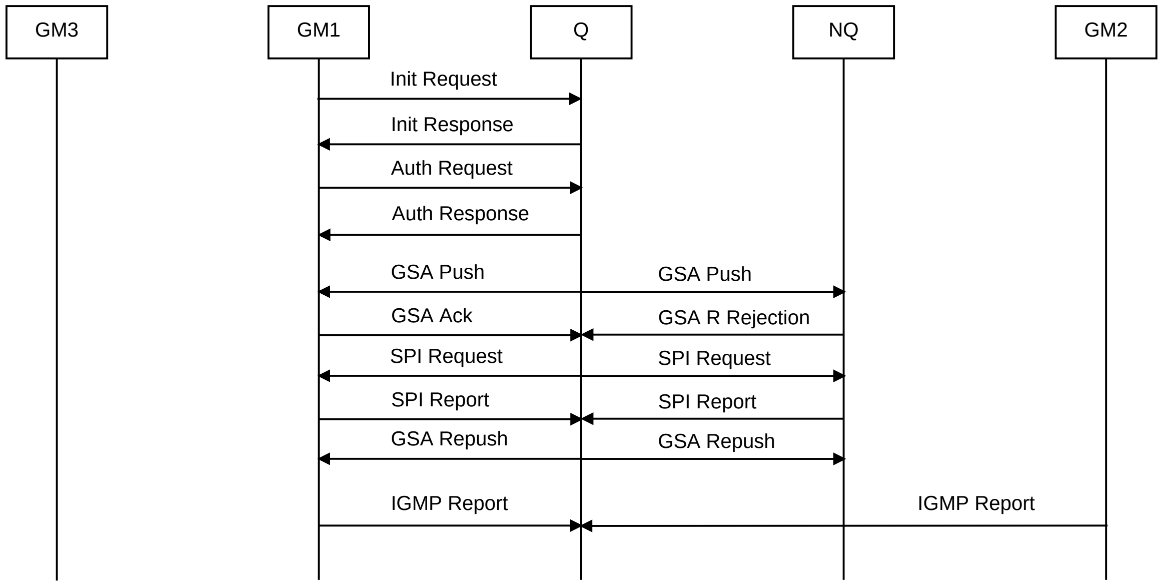


Figure 8.3: Packet Sequence When GSA R Is Rejected by NQs

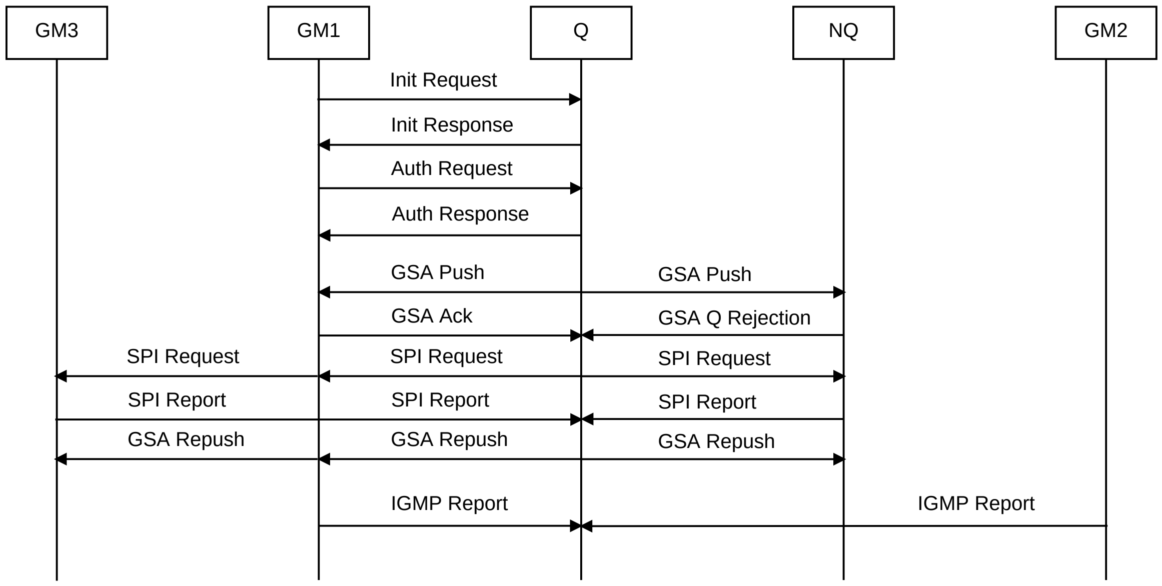


Figure 8.4: Packet Sequence When GSA Q or Both GSAs Are Rejected by NQs

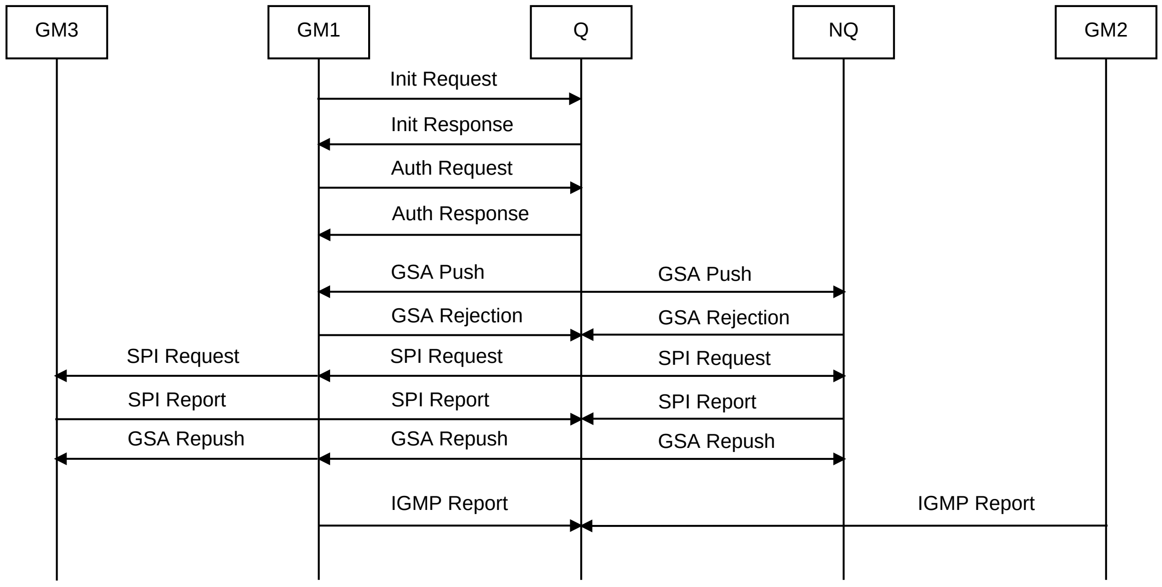


Figure 8.5: Packet Sequence When There Are SPI conflicts from both GM and NQs

There are five roles in these figures of our comparison:

(1) *GM1* represents a host who wishes to join a secure group through SIGMP and GSAM.

(2) *GM2* represents a host who wishes to join an open group through regular IGMPv3.

(3) *Q* represents the *Querier*.

(4) *NQ* represents a *Non-Querier*.

(5) *GM3* represents a group member who has already joined a group which *GM1* is joining.

Assuming no packet loss and *GM1* never joined any secure group, in all cases, the message sequence for *GM1*'s joining starts with two pairs of phase one message exchanges, followed by phase two exchanges, and ends with reception of *GM1*'s *Report* message by *Q*. *GM2*'s message sequence starts with sending the IGMP *Report* message and ends when *Q* receives that message. Phase one exchanges only involve *Q* and *GM1*. In all four cases of our comparison, message exchanges in phase one between *GM1* and *Q* always consists of a pair of *Init* messages and a pair of *Auth* messages. The sending of *Auth Response* message marks the ending of phase one. The *GSA Push* messages sent after it mark the beginning of phase two.

Phase two exchanges involve *Q*, *GM1*, *NQ* and *GM3*. Between *Q* and another participant, a GM or a NQ, it starts with a pair of *GSA Distribution* messages. Particularly, in case one, the pair of *GSA Distribution* messages consists of a *GSA Push* message and an *GSA Ack* message. In case two, three, four and five, there will be three additional messages, which consist of a pair of *SPI Resolve* messages and a *GSA Repush* message, if either *GM1* or *NQ* rejects the GSA pair from *Q* because of SPI conflict. In cases of SPI conflict, *Q* will initiate the *SPI Resolve Exchange* whenever it receives a *GSA Rejection* message. However, *Q* will not generate a new resolved SPI for the GSA or send *GSA Repush* message until it receives *SPI Report* messages from all members involved in the process.

Generally, *GM1* sends the IGMP (SIGMP) *Report* to the network segment after finishing message exchanges of GSAM. Specifically, in case one, *GM1* sends the IGMP *Report* message following the transmission of the *GSA Ack* message. In case two, three, four, the IGMP *Report* message is sent when *GM1* receives the *GSA Repush* message.

As to *GM2*, its group joining process remains identical in all four cases.

8.2 Factors that Influence Performance

In the best-case scenario (*case 1*, shown in Figure 8.1), there is a total of seven messages during the joining process, consisting of four first phase messages, two second phase messages between *GM1* and the *Q*, two second phase messages between the *Q* and *NQ* and one IGMP message. In the worst-case scenario (*case 4*, shown in Figure 8.4 and *case 5*, shown in Figure 8.5), the combination of messages is four first phase messages, five second phase messages between *GM1* and the *Q*, five second phase messages between the *Q* and *NQ*, three second phase messages between the *Q* and *GM3* and one IGMP message, which is eighteen messages in total.

In summary, when a host joins a secure multicast group in a network segment that has m NQs and n GMs that have joined the same group, the minimal number of messages needed in the whole process is seven and the maximal number is $10 + 5m + 3n$.

For performance evaluation, we measure the additional delay for joining a secure group using SIGMP and GSAM. The additional delay consists of:

(1) Transmission time of messages, which can be affected by:

  (a) The number of NQs

  (b) The number of GMs that joined the group

  (c) The probability of SPI conflict

(2) Transition time between two phases of GSAM and the one between SIGMP and GSAM

Firstly, transmission time of messages is affected by:

(1) The number of messages

(2) The speed and latency of the link

Secondly, since the Q is the ending point of phase one and the starting point of phase two, a parameter in GSAM is needed to determine when phase two starts. Similarly, another parameter is also needed for the transition between receiving *GSA Repush* message and sending IGMP Report. We name these parameters:

(1) *TIME_TRANSISTION_P1P2*

(2) *TIME_TRANSISTION_ACKREPORT*

8.3 Assumptions

While we model the boundary mechanism of IPsec, the duration of running encryption is not implemented because ns-3 is an event driven simulator and it can not measure the how long a function runs. Furthermore, studies (CALOMEL, 2017) (IoT Business Unit, ARM, 2015) show the duration of running encryption is negligible compared to the factors we analyzed above. Additionally, we make the following settings and assumptions:

(1) All hosts reside in a CSMA/CD ethernet segment with $100Mbps$ speed and $10ms$ latency, which is reasonable to home environment.

(2) *TIME_TRANSISTION_P1P2* and *TIME_TRANSISTION_ACKREPORT* is set to 0

(3) IGMPv3's parameters are set to default values by IETF

(4) GMs periodically join a random group every second, which we assume it is large enough to avoid congestion.

8.4 Initial Simulation

In ns-3, we run an initial simulation of comparing join time between IGMPv3 and SIGMP with different probability of SPI conflict in following additional settings and assumptions:

(1) In the network segment, there are one Q, two NQs and 10 GMs.

(2) No IGMP packet loss: A GM may have to wait for a long general query interval for reporting membership if loss of IGMP *Report* messages.

Figure 8.6 shows the result of initial simulation:

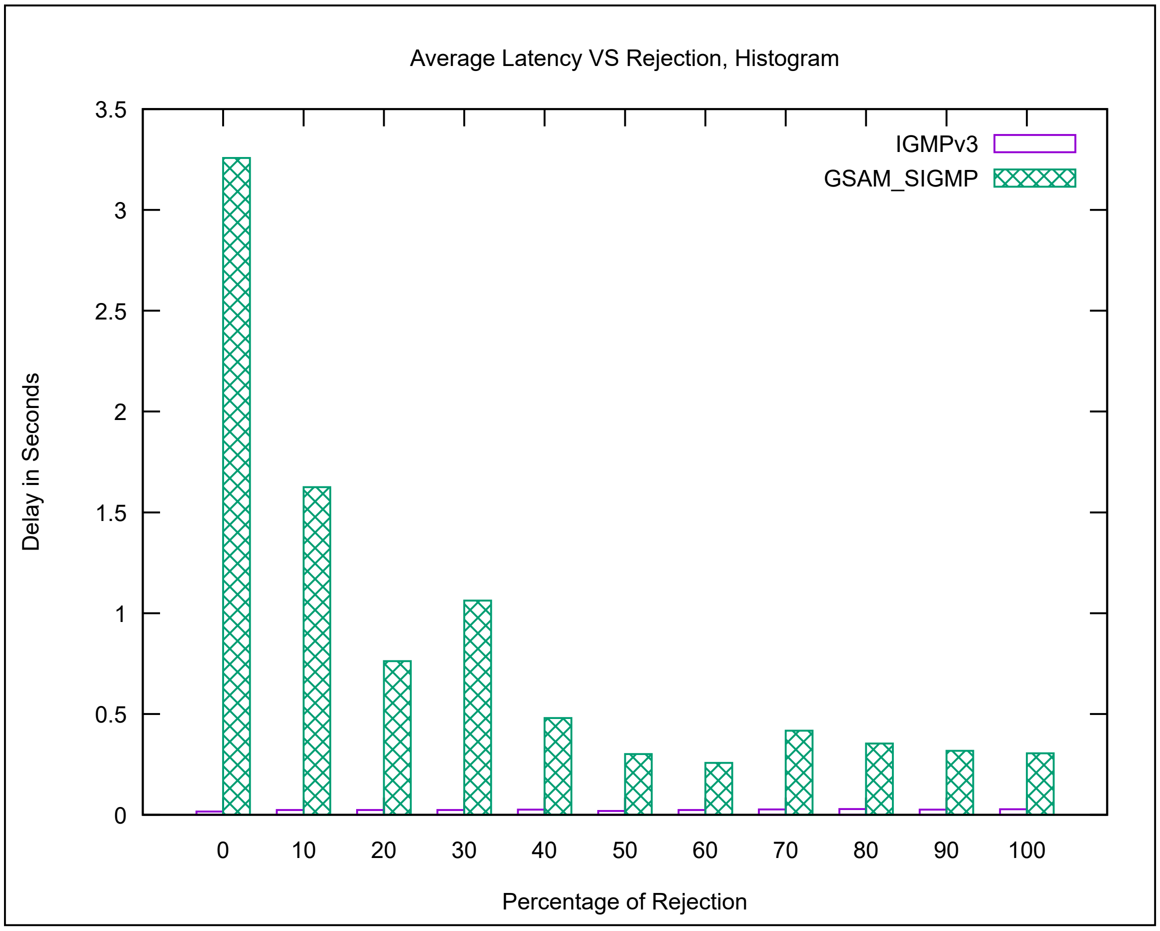


Figure 8.6: The Result of Initial Simulation

As we can see from the result, the average delay of joining a group using IGMPv3 is low. And the average latency when joining a secure group using SIGMP and GSAM is many times more no matter what the rejection percentage is. Moreover, we can observe an abnormality here. The transmission time of messages in SIGMP and GSAM should increase when there are more SPI conflicts. However, the result shows the opposite trend.

After examining the log of packet transmission, we found the cause to this abnormality. Since the Q would only install a GSA after receiving *GSA Ack* messages from everyone, if the IGMP *Report* message reaches the Q before the *GSA Ack* message does, the IGMP *Report* message would be discarded by the IPsec system. Then the Q has to wait for retransmission of the IGMP *Report* message or a *Report* message in response to a *General Query* message. Either of them takes up to several seconds.

8.5 Revision of Message Exchanges and its Result

We simply fix this issue by making the Q send an IGMP *Group Specific Report* message for the secure group that *GM1* is joining when it receives all the *GSA Ack* messages and finishes installing the GSA. The revised message sequence of *case 1* is shown in Figure 8.7:

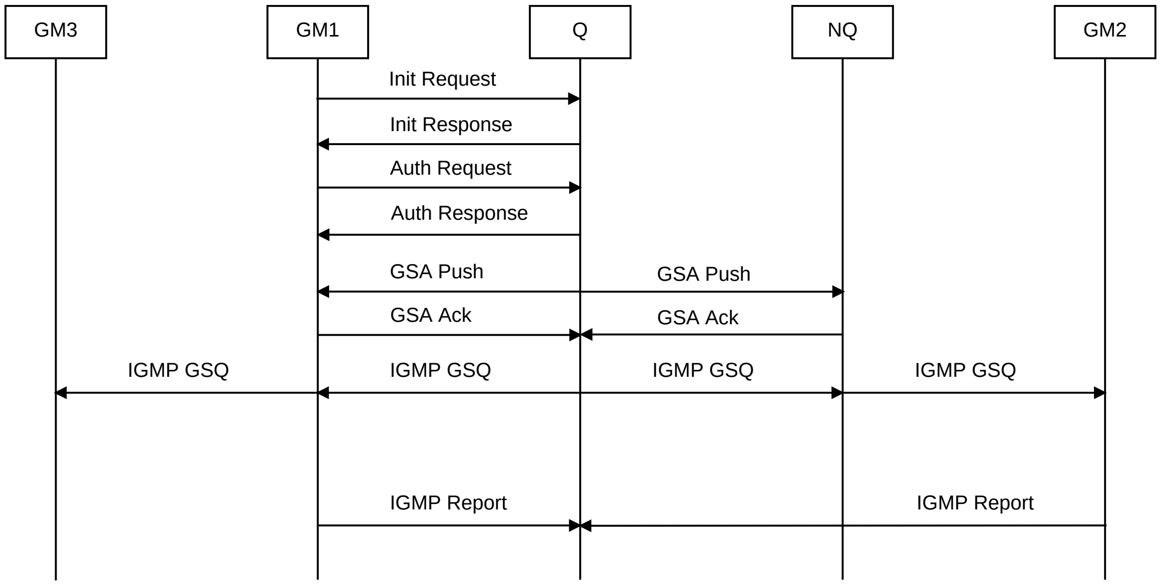


Figure 8.7: Revised Packet Sequence When No SPI Conflict Happens

The simulation result after the revision of message exchanges is shown in Figure 8.8:

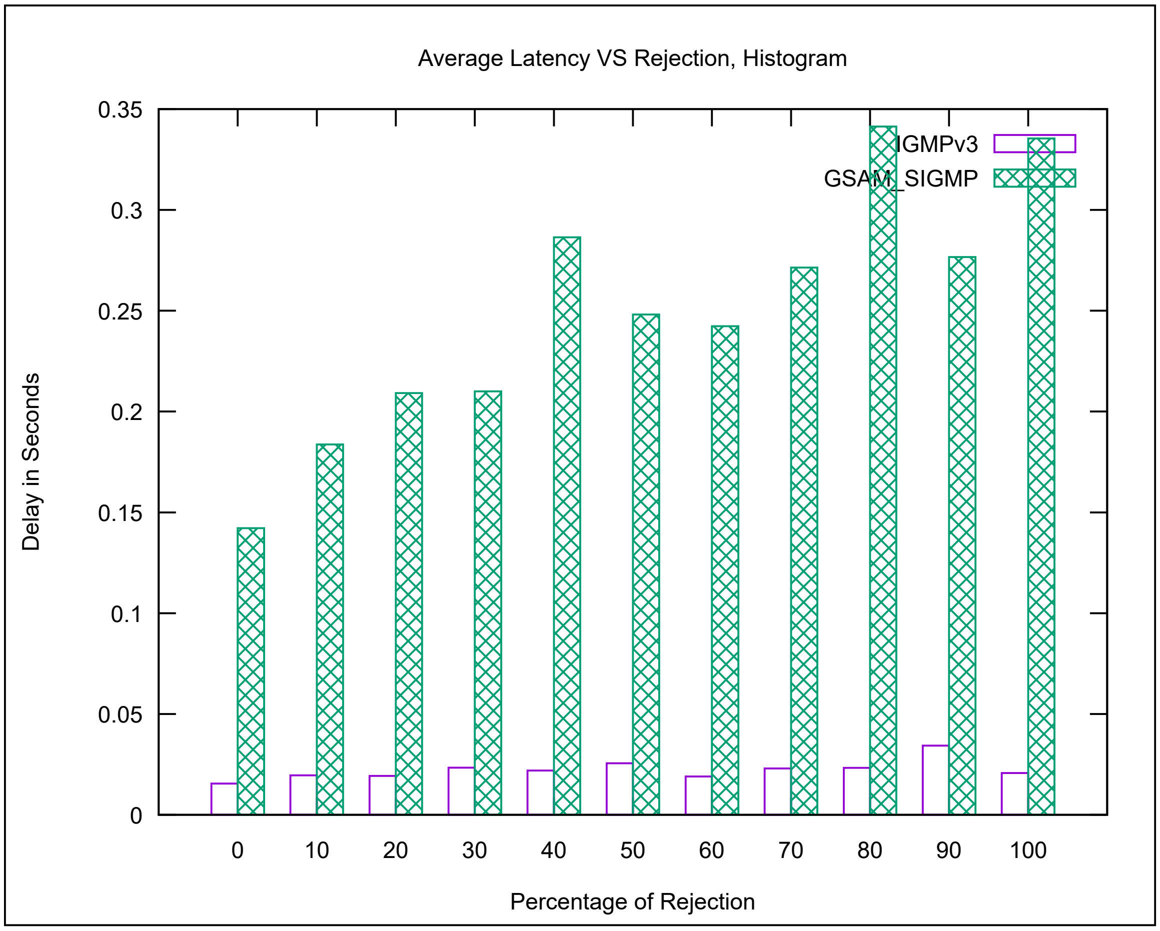


Figure 8.8: The Result after Revising Packet Sequence When No SPI Conflict Happens

As we can see from Figure 8.8, compared to the previous simulation, the average latency is greatly improved in cases of low percentage of *SPI Rejection*, from seconds to hundreds of milliseconds. However, it is still much higher than the one in IGMPv3.

8.6 Simulations with Different Numbers of NQs and GMs

After fixing the abnormality of the protocol, we continue the evaluation of the performance. As we analyzed that the numbers of NQs and GMs also impact the number of messages to be exchanged during the process of GSAM, we conduct two more experiment with different sets of numbers of NQs and GMs.

Firstly, we show the result of the second simulation with different numbers of NQs in Figure 8.9:

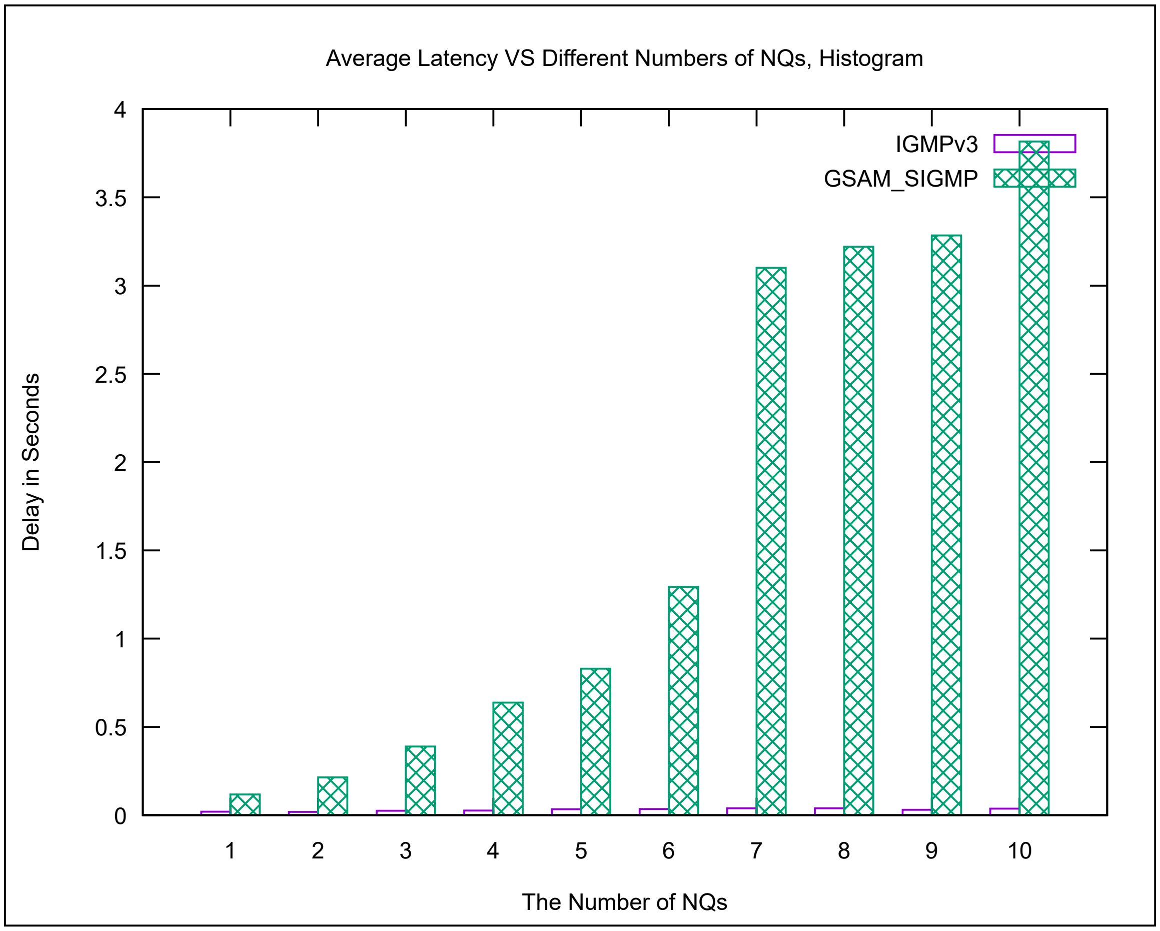


Figure 8.9: The Result of Second Simulation

In the experiment, the number of GMs is set to 10, an the *SPI Rejection Rate* is set to 20%. The rest of the parameters are identically set as they are in the initial simulation. As we can see from Figure 8.9, the average latency of joining a group in SIGMP and GSAM protocols bundle drastically increases as the number of NQs grows, specially when the number is larger than 4.

The first result of third experiment is shown in Figure 8.10:

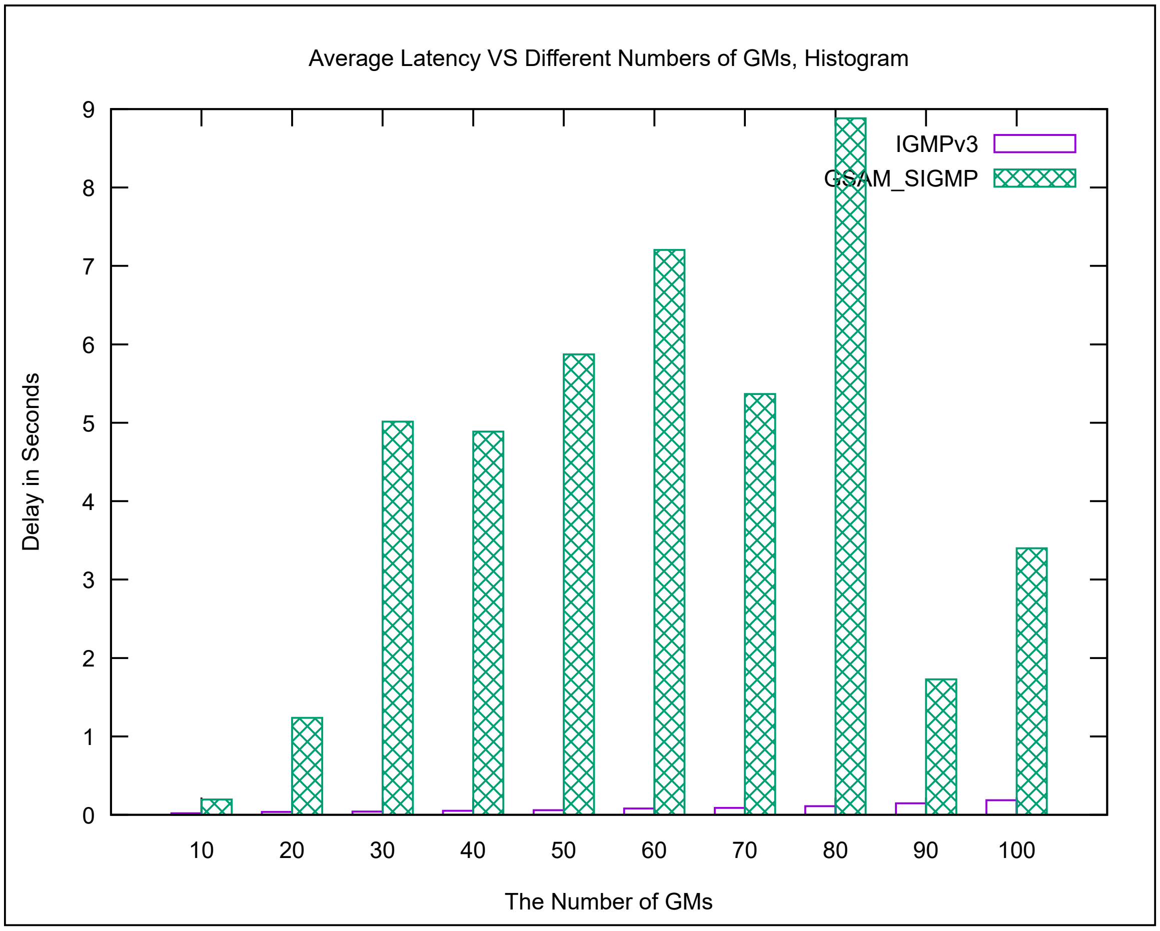


Figure 8.10: The First Result of Third Simulation

In the experiment, the number of NQs is set to 2. The rest of the paraments remain unchanged. As we can see from Figure 8.10. The average latency fluctuates as the number of GMs increases. We can also observe that the latency of joining groups in IGMPv3 also greatly increased. Therefore, we speculate it is caused by the congestion of the network. In above shown tests, each GM joins a group every one second. We increase the interval between joining events to 10 seconds and the result is shown in Figure 8.11:

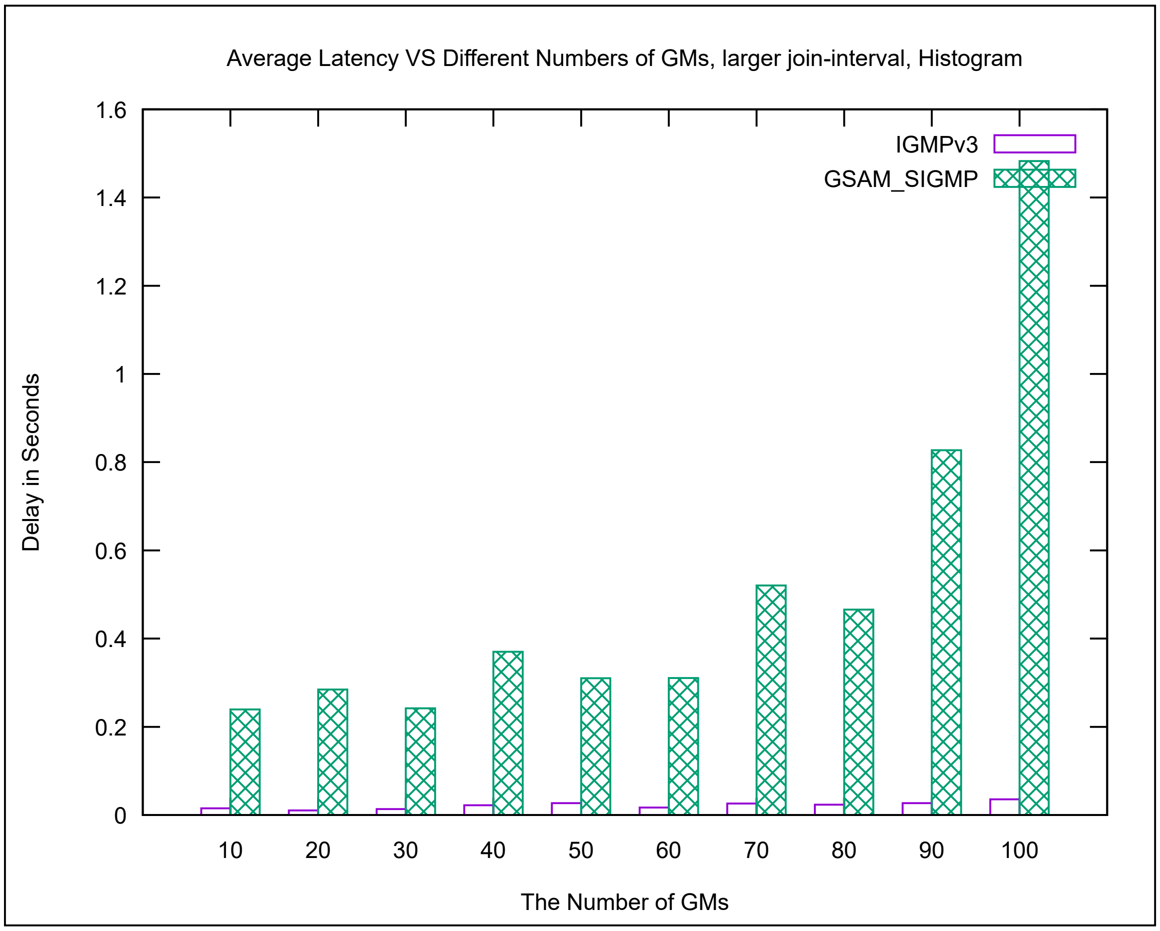


Figure 8.11: The Second Result of Third Simulation

As we can see from Figure 8.11, the average latency stays around $200ms$ to $400ms$ before the network starts becoming overcrowd when the number of GMs is larger than 60.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this thesis, we analyze the performance of SIGMP and GSAM protocol bundle and compare its latency of joining a group with the one of IGMPv3 by simulation in ns-3. In our analysis, we present all possible cases of packet sequence and the factors that affect the performance. The first simulation result shows SIGMP and GSAM protocol bundle has much inferior performance when compared to IGMPv3. Moreover, the result also shows abnormal performance, which is that its average latency decreases as *SPI Rejection Rate* increases.

Therefore, we analyze the log file of the first simulation and discover the cause of the issue. Furthermore, we revise the packet sequence of a specific case of GSAM's operation and conduct a second simulation. The result of the revised approach of SIGMP and GSAM protocol bundle shows the anticipated behavior from our performance analysis. Yet, its performance is still many times worse than IGMPv3, an average latency of hundreds of milliseconds versus dozens of milliseconds in IGMPv3.

Then, we conduct two more expriments with other two factors that influence the performance: the number of NQs and the number of GMs. The results show that the average latency will become very high when the number of NQs is large. When the network is not congested, the number of GMs has very limited impact on the performance.

At last, taking video streaming as the use case, the latency of the IGMP process is relatively smaller compared to others such as video streaming buffering and synchronization, which can take up to two seconds. For better revenue collection, we conclude from the results of our simulations that when the number of NQs is small and the local network is not congested, the additional hundreds of milliseconds latency is a reasonable overhead for the security and access control provided by SIGMP and GSAM.

9.2 Future Work

Though the performance of the protocol bundle has been measured, there are details undefined and space for improvement. Firstly, a secure but not authenticated channel established in Init exchange of phase one between Q and NQ will be shared during joining processes of multiple groups by multiple GMs. That is to say, the Q may issue simultaneous requests to NQ using a single *Init SA*. Therefore, it requires a sliding-window mechanism for processing multiple incoming requests and their retransmissions on the endpoint of NQs. Currently, in the simulation of this thesis, NQ always treats incoming requests as new requests and increase its counters of message id accordingly. Secondly, the issue of what content should be included in the *SPI Report* messages. To generate a new SPI value whenever there is a SPI conflict, the Q needs to gather used SPI values from everyone who is involved in joining process of the group by sending *SPI Request* messages. Whoever receives such message should respond accordingly. In the current simulation, a host will enclose every SPI valus used in its database. This is not efficient when the list of SPI values is large.

Appendix A

Class Diagrams of the Implemention

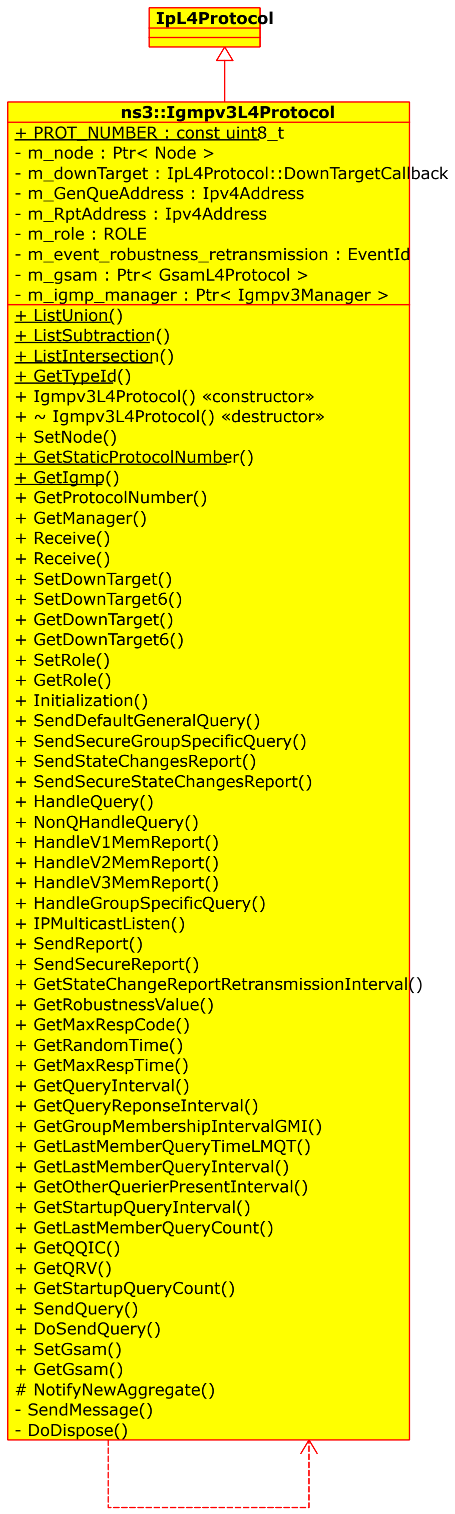

IpL4Protocol
ns3::Igmpv3L4Protocol
+ PROT_NUMBER : const uint8_t
- m_node : Ptr< Node >
- m_downTarget : IpL4Protocol::DownTargetCallback
- m_GenQueAddress : Ipv4Address
- m_RptAddress : Ipv4Address
- m_role : ROLE
- m_event_robustness_retransmission : EventId
- m_gsam : Ptr< GsamL4Protocol >
- m_igmp_manager : Ptr< Igmpv3Manager >
+ ListUnion()
+ ListSubtraction()
+ ListIntersection()
+ GetTypeId()
+ Igmpv3L4Protocol() «constructor»
+ ~ Igmpv3L4Protocol() «destructor»
+ SetNode()
+ GetStaticProtocolNumber()
+ GetIgmp()
+ GetProtocolNumber()
+ GetManager()
+ Receive()
+ Receive()
+ SetDownTarget()
+ SetDownTarget6()
+ GetDownTarget()
+ GetDownTarget6()
+ SetRole()
+ GetRole()
+ Initialization()
+ SendDefaultGeneralQuery()
+ SendSecureGroupSpecificQuery()
+ SendStateChangesReport()
+ SendSecureStateChangesReport()
+ HandleQuery()
+ NonQHandleQuery()
+ HandleV1MemReport()
+ HandleV2MemReport()
+ HandleV3MemReport()
+ HandleGroupSpecificQuery()
+ IPMulticastListen()
+ SendReport()
+ SendSecureReport()
+ GetStateChangeReportRetransmissionInterval()
+ GetRobustnessValue()
+ GetMaxRespCode()
+ GetRandomTime()
+ GetMaxRespTime()
+ GetQueryInterval()
+ GetQueryReponseInterval()
+ GetGroupMembershipIntervalGMI()
+ GetLastMemberQueryTimeLMQT()
+ GetLastMemberQueryInterval()
+ GetOtherQuerierPresentInterval()
+ GetStartupQueryInterval()
+ GetLastMemberQueryCount()
+ GetQQIC()
+ GetQRV()
+ GetStartupQueryCount()
+ SendQuery()
+ DoSendQuery()
+ SetGsam()
+ GetGsam()
# NotifyNewAggregate()
- SendMessage()
- DoDispose()

Figure A.1: The Class Diagrams of IGMPv3 and SIGMP Protocols Implementation

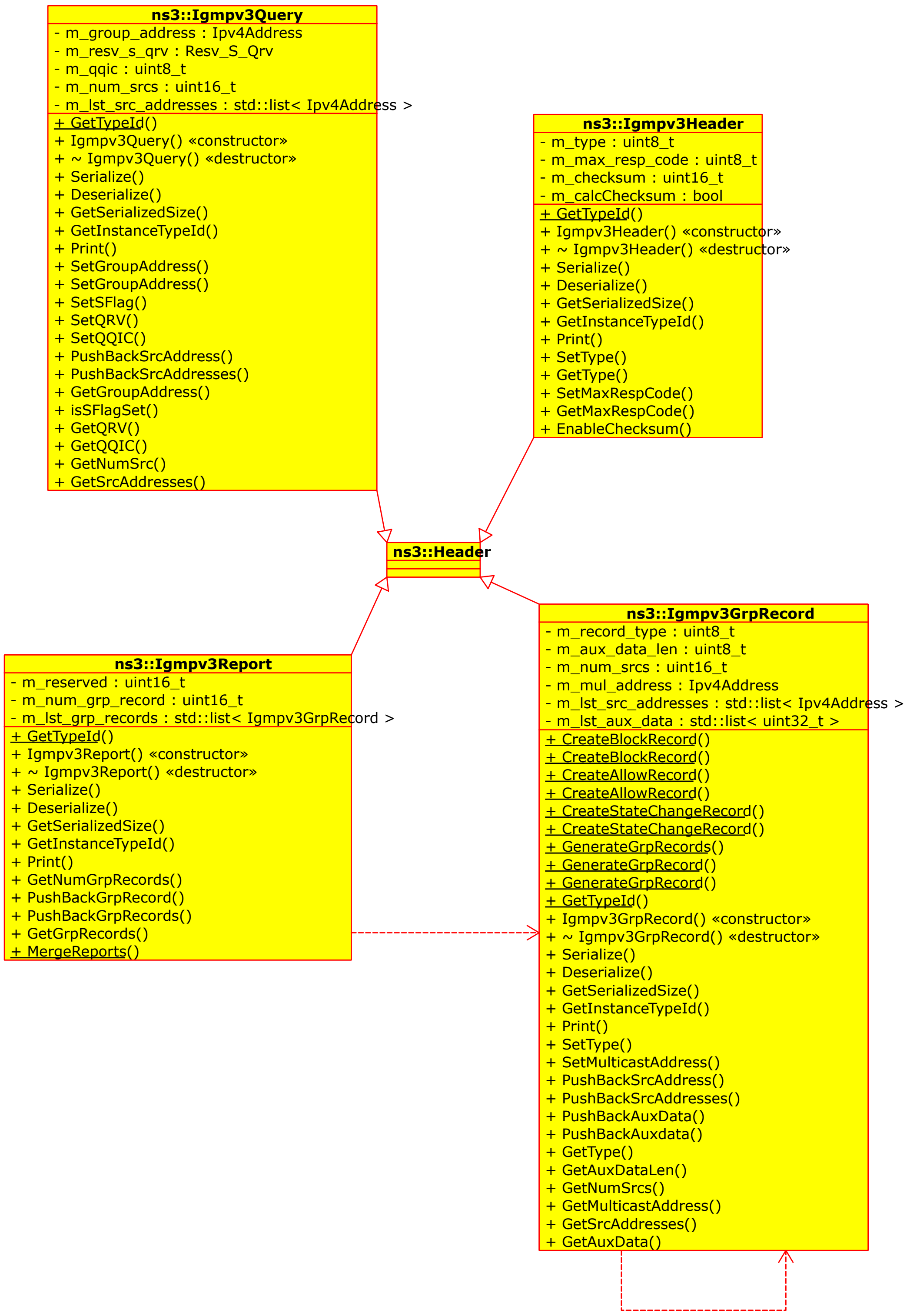


Figure A.2: The Class Diagrams of IGMPv3 Packet Data Structures Implementation

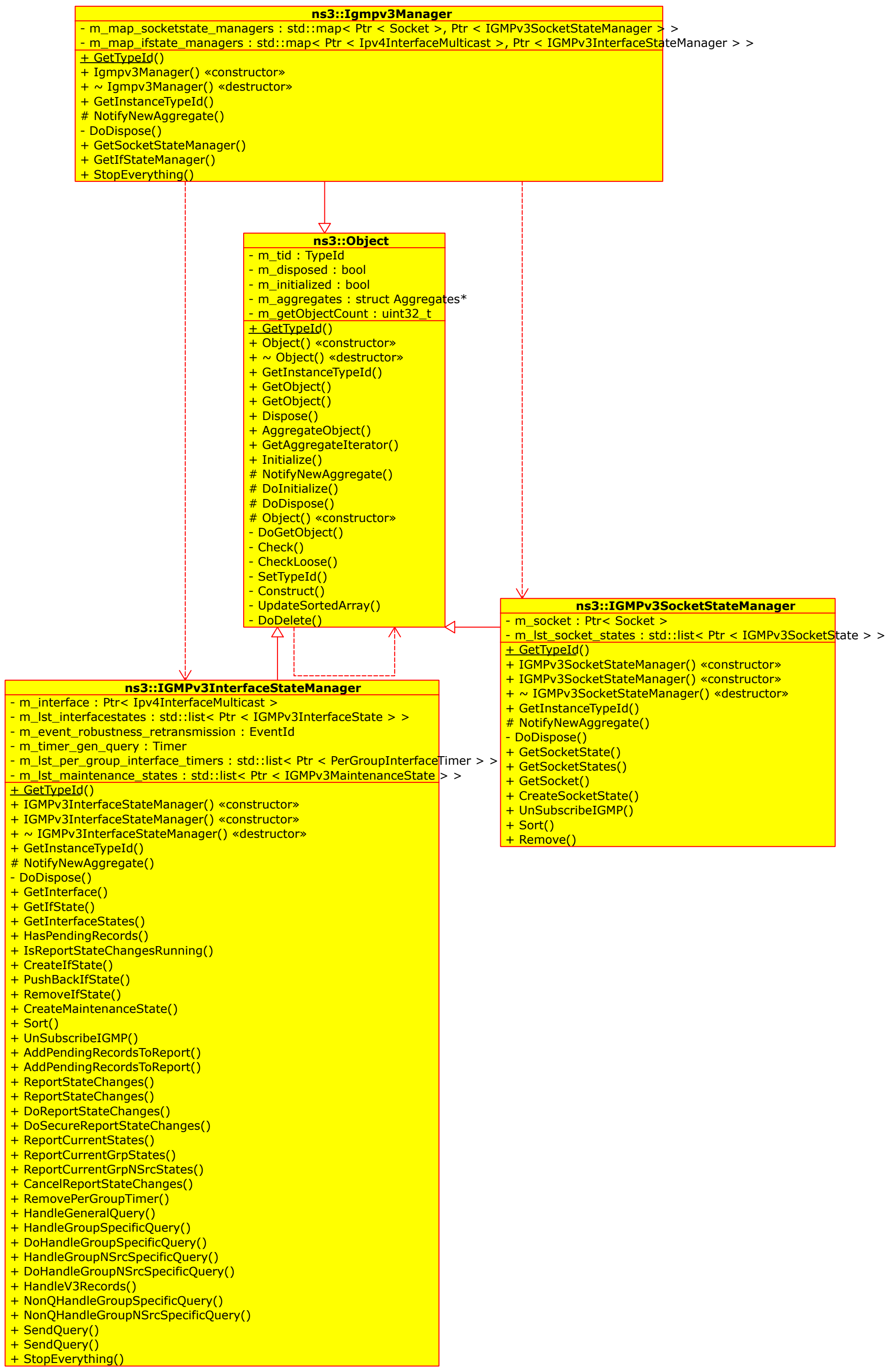

ns3::Igmpv3Manager
- m_map_socketstate_managers : std::map< Ptr < Socket >, Ptr < IGMPv3SocketStateManager > >
- m_map_ifstate_managers : std::map< Ptr < Ipv4InterfaceMulticast >, Ptr < IGMPv3InterfaceStateManager > >
+ GetTypeId()
+ Igmpv3Manager() «constructor»
+ ~ Igmpv3Manager() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ GetSocketStateManager()
+ GetIfStateManager()
+ StopEverything()
ns3::Object
- m_tid : TypeId
- m_disposed : bool
- m_initialized : bool
- m_aggregates : struct Aggregates*
- m_getObjectCount : uint32_t
+ GetTypeId()
+ Object() «constructor»
+ ~ Object() «destructor»
+ GetInstanceTypeId()
+ GetObject()
+ GetObject()
+ Dispose()
+ AggregateObject()
+ GetAggregateIterator()
+ Initialize()
# NotifyNewAggregate()
# DoInitialize()
# DoDispose()
# Object() «constructor»
- DoGetObject()
- Check()
- CheckLoose()
- SetTypeId()
- Construct()
- UpdateSortedArray()
- DoDelete()
ns3::IGMPv3SocketStateManager
- m_socket : Ptr< Socket >
- m_lst_socket_states : std::list< Ptr < IGMPv3SocketState > >
+ GetTypeId()
+ IGMPv3SocketStateManager() «constructor»
+ IGMPv3SocketStateManager() «constructor»
+ ~ IGMPv3SocketStateManager() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ GetSocketState()
+ GetSocketStates()
+ GetSocket()
+ CreateSocketState()
+ UnSubscribeIGMP()
+ Sort()
+ Remove()
ns3::IGMPv3InterfaceStateManager
- m_interface : Ptr< Ipv4InterfaceMulticast >
- m_lst_interfacestates : std::list< Ptr < IGMPv3InterfaceState > >
- m_event_robustness_retransmission : EventId
- m_timer_gen_query : Timer
- m_lst_per_group_interface_timers : std::list< Ptr < PerGroupInterfaceTimer > >
- m_lst_maintenance_states : std::list< Ptr < IGMPv3MaintenanceState > >
+ GetTypeId()
+ IGMPv3InterfaceStateManager() «constructor»
+ IGMPv3InterfaceStateManager() «constructor»
+ ~ IGMPv3InterfaceStateManager() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ GetInterface()
+ GetIfState()
+ GetInterfaceStates()
+ HasPendingRecords()
+ IsReportStateChangesRunning()
+ CreateIfState()
+ PushBackIfState()
+ RemoveIfState()
+ CreateMaintenanceState()
+ Sort()
+ UnSubscribeIGMP()
+ AddPendingRecordsToReport()
+ AddPendingRecordsToReport()
+ ReportStateChanges()
+ ReportStateChanges()
+ DoReportStateChanges()
+ DoSecureReportStateChanges()
+ ReportCurrentStates()
+ ReportCurrentGrpStates()
+ ReportCurrentGrpNSrcStates()
+ CancelReportStateChanges()
+ RemovePerGroupTimer()
+ HandleGeneralQuery()
+ HandleGroupSpecificQuery()
+ DoHandleGroupSpecificQuery()
+ HandleGroupNSrcSpecificQuery()
+ DoHandleGroupNSrcSpecificQuery()
+ HandleV3Records()
+ NonQHandleGroupSpecificQuery()
+ NonQHandleGroupNSrcSpecificQuery()
+ SendQuery()
+ SendQuery()
+ StopEverything()

Figure A.3: The Class Diagrams of IGMPv3 Databases Implementation Overview

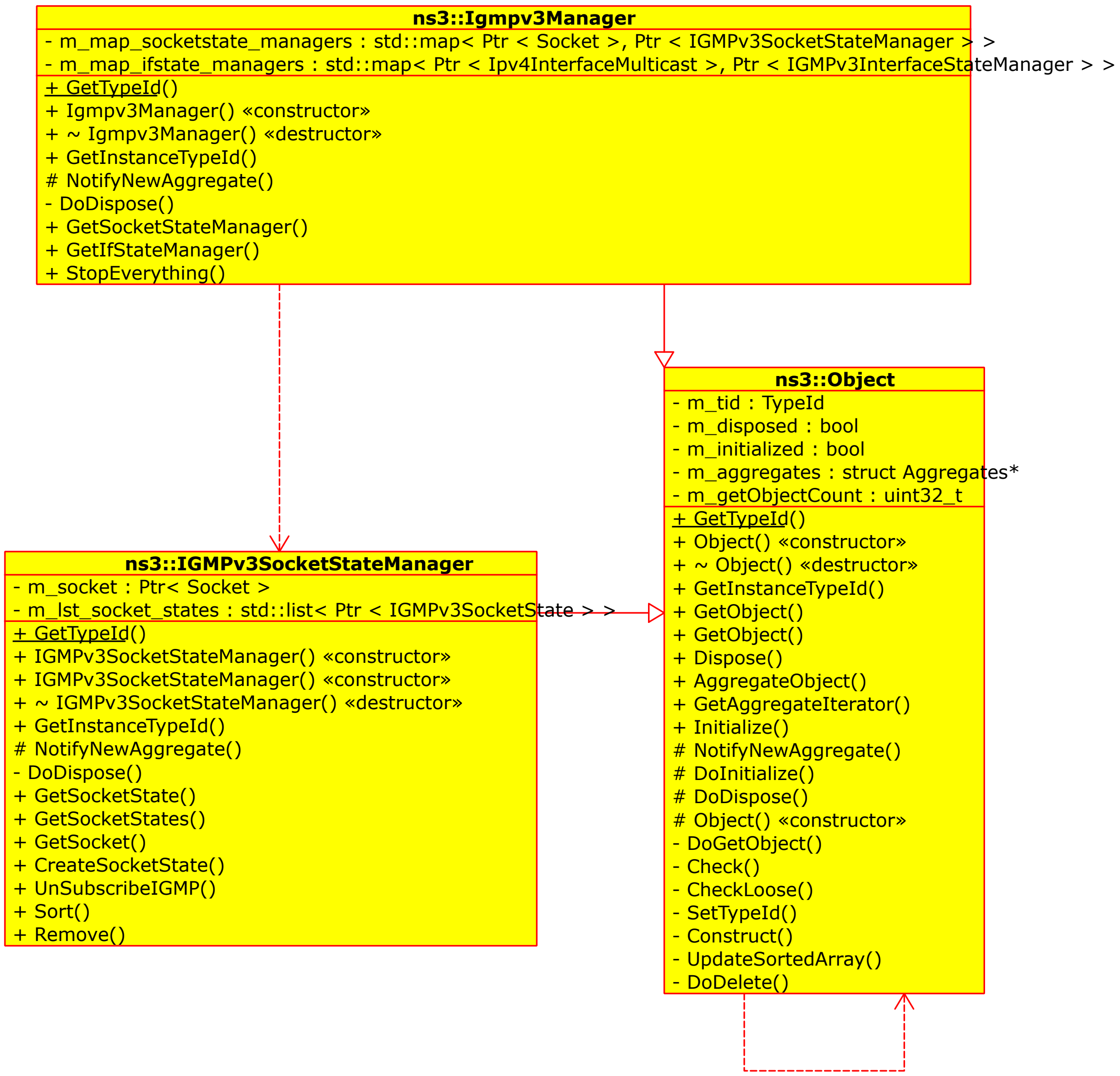


Figure A.4: The Class Diagrams of IGMPv3 Socket State Database Implementation

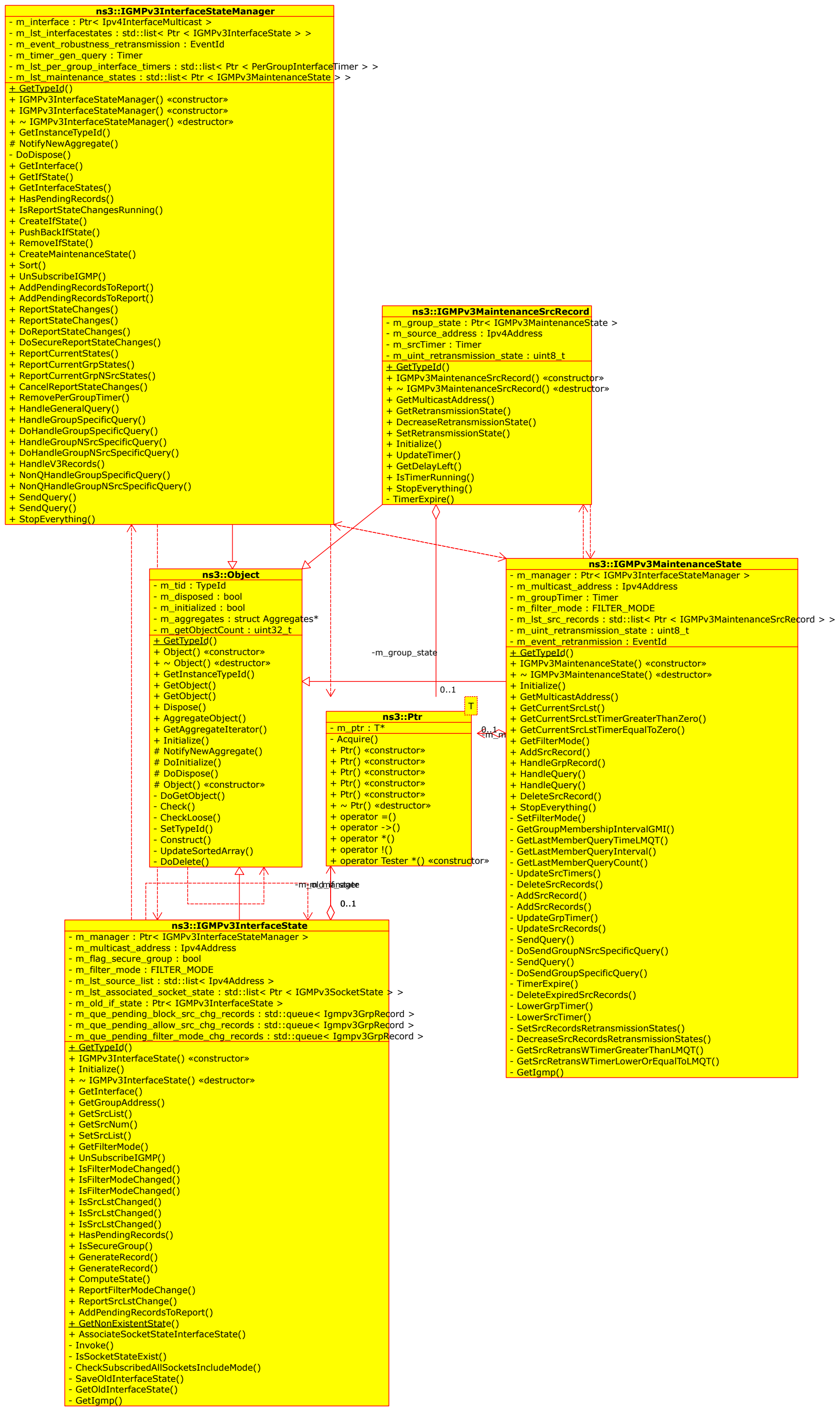

ns3::IGMPv3InterfaceStateManager
- m_interface : Ptr< Ipv4InterfaceMulticast >
- m_lst_interfacestates : std::list< Ptr < IGMPv3InterfaceState > >
- m_event_robustness_retransmission : EventId
- m_timer_gen_query : Timer
- m_lst_per_group_interface_timers : std::list< Ptr < PerGroupInterfaceTimer > >
- m_lst_maintenance_states : std::list< Ptr < IGMPv3MaintenanceState > >
+ GetTypeId()
+ IGMPv3InterfaceStateManager() «constructor»
+ IGMPv3InterfaceStateManager() «constructor»
+ ~ IGMPv3InterfaceStateManager() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ GetInterface()
+ GetIfState()
+ GetInterfaceStates()
+ HasPendingRecords()
+ IsReportStateChangesRunning()
+ CreateIfState()
+ PushBackIfState()
+ RemoveIfState()
+ CreateMaintenanceState()
+ Sort()
+ UnSubscribeIGMP()
+ AddPendingRecordsToReport()
+ AddPendingRecordsToReport()
+ ReportStateChanges()
+ ReportStateChanges()
+ DoReportStateChanges()
+ DoSecureReportStateChanges()
+ ReportCurrentStates()
+ ReportCurrentGrpStates()
+ ReportCurrentGrpNSrcStates()
+ CancelReportStateChanges()
+ RemovePerGroupTimer()
+ HandleGeneralQuery()
+ HandleGroupSpecificQuery()
+ DoHandleGroupSpecificQuery()
+ HandleGroupNSrcSpecificQuery()
+ DoHandleGroupNSrcSpecificQuery()
+ HandleV3Records()
+ NonQHandleGroupSpecificQuery()
+ NonQHandleGroupNSrcSpecificQuery()
+ SendQuery()
+ SendQuery()
+ StopEverything()
ns3::IGMPv3MaintenanceSrcRecord
- m_group_state : Ptr< IGMPv3MaintenanceState >
- m_source_address : Ipv4Address
- m_srcTimer : Timer
- m_uint_retransmission_state : uint8_t
+ GetTypeId()
+ IGMPv3MaintenanceSrcRecord() «constructor»
+ ~ IGMPv3MaintenanceSrcRecord() «destructor»
+ GetMulticastAddress()
+ GetRetransmissionState()
+ DecreaseRetransmissionState()
+ SetRetransmissionState()
+ Initialize()
+ UpdateTimer()
+ GetDelayLeft()
+ IsTimerRunning()
+ StopEverything()
- TimerExpire()
ns3::Object
- m_tid : TypeId
- m_disposed : bool
- m_initialized : bool
- m_aggregates : struct Aggregates*
- m_getObjectCount : uint32_t
+ GetTypeId()
+ Object() «constructor»
+ ~ Object() «destructor»
+ GetInstanceTypeId()
+ GetObject()
+ GetObject()
+ Dispose()
+ AggregateObject()
+ GetAggregateIterator()
+ Initialize()
# NotifyNewAggregate()
# DoInitialize()
# DoDispose()
# Object() «constructor»
- DoGetObject()
- Check()
- CheckLoose()
- SetTypeId()
- Construct()
- UpdateSortedArray()
- DoDelete()
-m_group_state
0..1
T
ns3::Ptr
- m_ptr : T*
- Acquire()
+ Ptr() «constructor»
+ Ptr() «constructor»
+ Ptr() «constructor»
+ Ptr() «constructor»
+ Ptr() «constructor»
+ ~ Ptr() «destructor»
+ operator =()
+ operator ->()
+ operator *()
+ operator !()
+ operator Tester *() «constructor»
0..1
-m_m
-m_old_if_state
-m_manager
0..1
ns3::IGMPv3MaintenanceState
- m_manager : Ptr< IGMPv3InterfaceStateManager >
- m_multicast_address : Ipv4Address
- m_groupTimer : Timer
- m_filter_mode : FILTER_MODE
- m_lst_src_records : std::list< Ptr < IGMPv3MaintenanceSrcRecord > >
- m_uint_retransmission_state : uint8_t
- m_event_retranmission : EventId
+ GetTypeId()
+ IGMPv3MaintenanceState() «constructor»
+ ~ IGMPv3MaintenanceState() «destructor»
+ Initialize()
+ GetMulticastAddress()
+ GetCurrentSrcLst()
+ GetCurrentSrcLstTimerGreaterThanZero()
+ GetCurrentSrcLstTimerEqualToZero()
+ GetFilterMode()
+ AddSrcRecord()
+ HandleGrpRecord()
+ HandleQuery()
+ HandleQuery()
+ DeleteSrcRecord()
+ StopEverything()
- SetFilterMode()
- GetGroupMembershipIntervalGMI()
- GetLastMemberQueryTimeLMQT()
- GetLastMemberQueryInterval()
- GetLastMemberQueryCount()
- UpdateSrcTimers()
- DeleteSrcRecords()
- AddSrcRecord()
- AddSrcRecords()
- UpdateGrpTimer()
- UpdateSrcRecords()
- SendQuery()
- DoSendGroupNSrcSpecificQuery()
- SendQuery()
- DoSendGroupSpecificQuery()
- TimerExpire()
- DeleteExpiredSrcRecords()
- LowerGrpTimer()
- LowerSrcTimer()
- SetSrcRecordsRetransmissionStates()
- DecreaseSrcRecordsRetransmissionStates()
- GetSrcRetransWTimerGreaterThanLMQT()
- GetSrcRetransWTimerLowerOrEqualToLMQT()
- GetIgmp()
ns3::IGMPv3InterfaceState
- m_manager : Ptr< IGMPv3InterfaceStateManager >
- m_multicast_address : Ipv4Address
- m_flag_secure_group : bool
- m_filter_mode : FILTER_MODE
- m_lst_source_list : std::list< Ipv4Address >
- m_lst_associated_socket_state : std::list< Ptr < IGMPv3SocketState > >
- m_old_if_state : Ptr< IGMPv3InterfaceState >
- m_que_pending_block_src_chg_records : std::queue< Igmpv3GrpRecord >
- m_que_pending_allow_src_chg_records : std::queue< Igmpv3GrpRecord >
- m_que_pending_filter_mode_chg_records : std::queue< Igmpv3GrpRecord >
+ GetTypeId()
+ IGMPv3InterfaceState() «constructor»
+ Initialize()
+ ~ IGMPv3InterfaceState() «destructor»
+ GetInterface()
+ GetGroupAddress()
+ GetSrcList()
+ GetSrcNum()
+ SetSrcList()
+ GetFilterMode()
+ UnSubscribeIGMP()
+ IsFilterModeChanged()
+ IsFilterModeChanged()
+ IsFilterModeChanged()
+ IsSrcLstChanged()
+ IsSrcLstChanged()
+ IsSrcLstChanged()
+ HasPendingRecords()
+ IsSecureGroup()
+ GenerateRecord()
+ GenerateRecord()
+ ComputeState()
+ ReportFilterModeChange()
+ ReportSrcLstChange()
+ AddPendingRecordsToReport()
+ GetNonExistentState()
+ AssociateSocketStateInterfaceState()
- Invoke()
- IsSocketStateExist()
- CheckSubscribedAllSocketsIncludeMode()
- SaveOldInterfaceState()
- GetOldInterfaceState()
- GetIgmp()

Figure A.5: The Class Diagrams of IGMPv3 Interface State and Reception State Databases Implementation

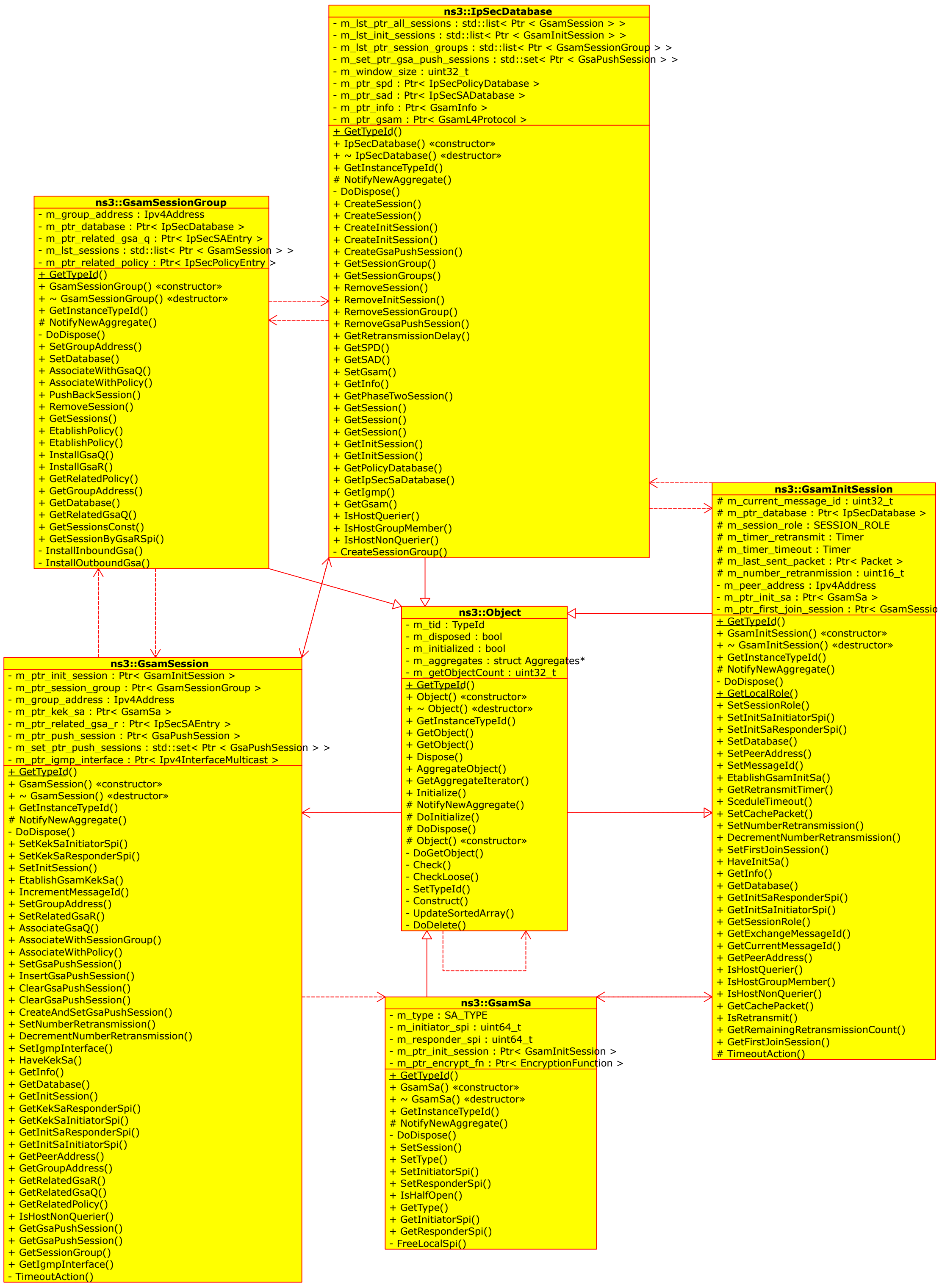


Figure A.6: The Class Diagrams of GSAM Database Implementation

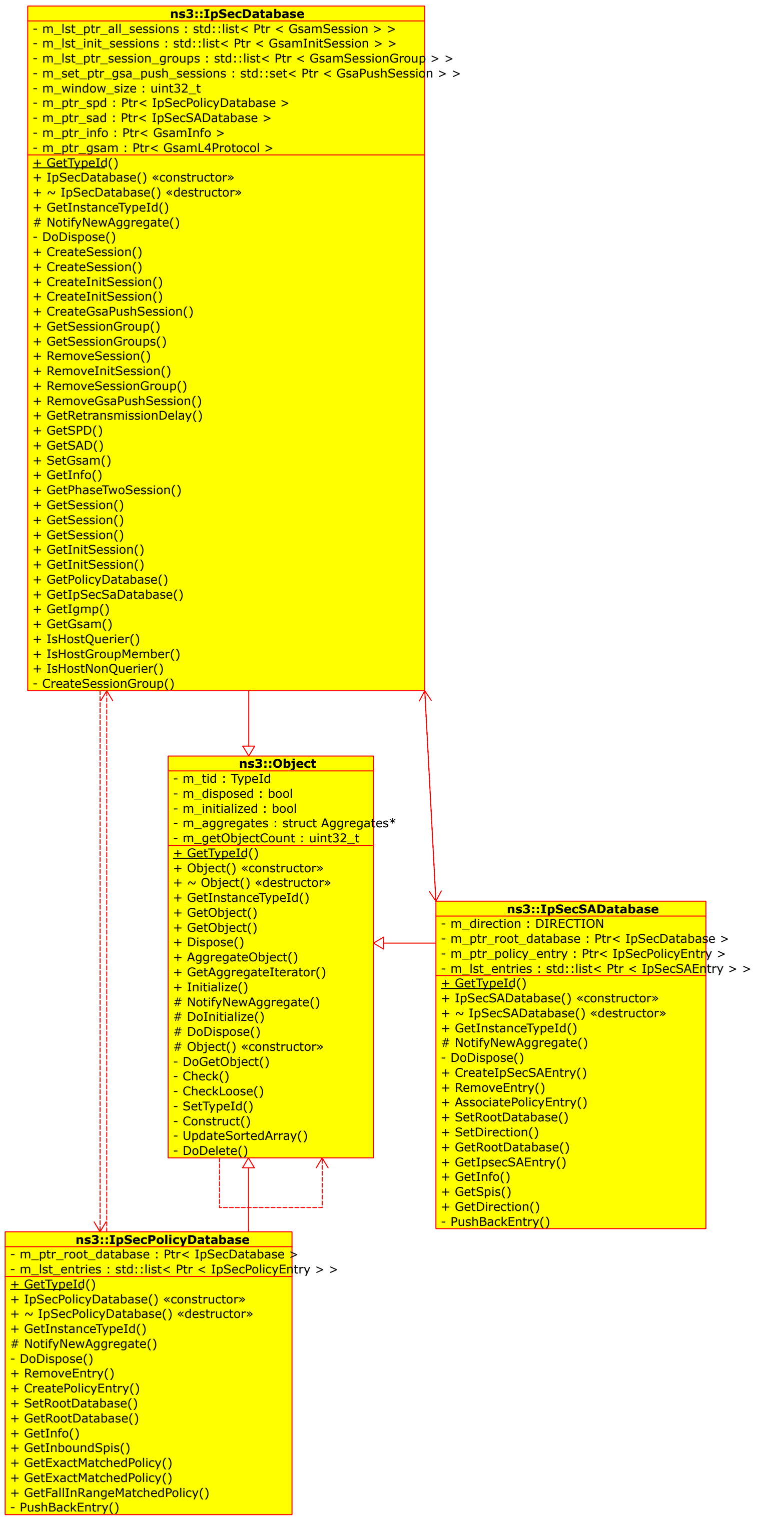
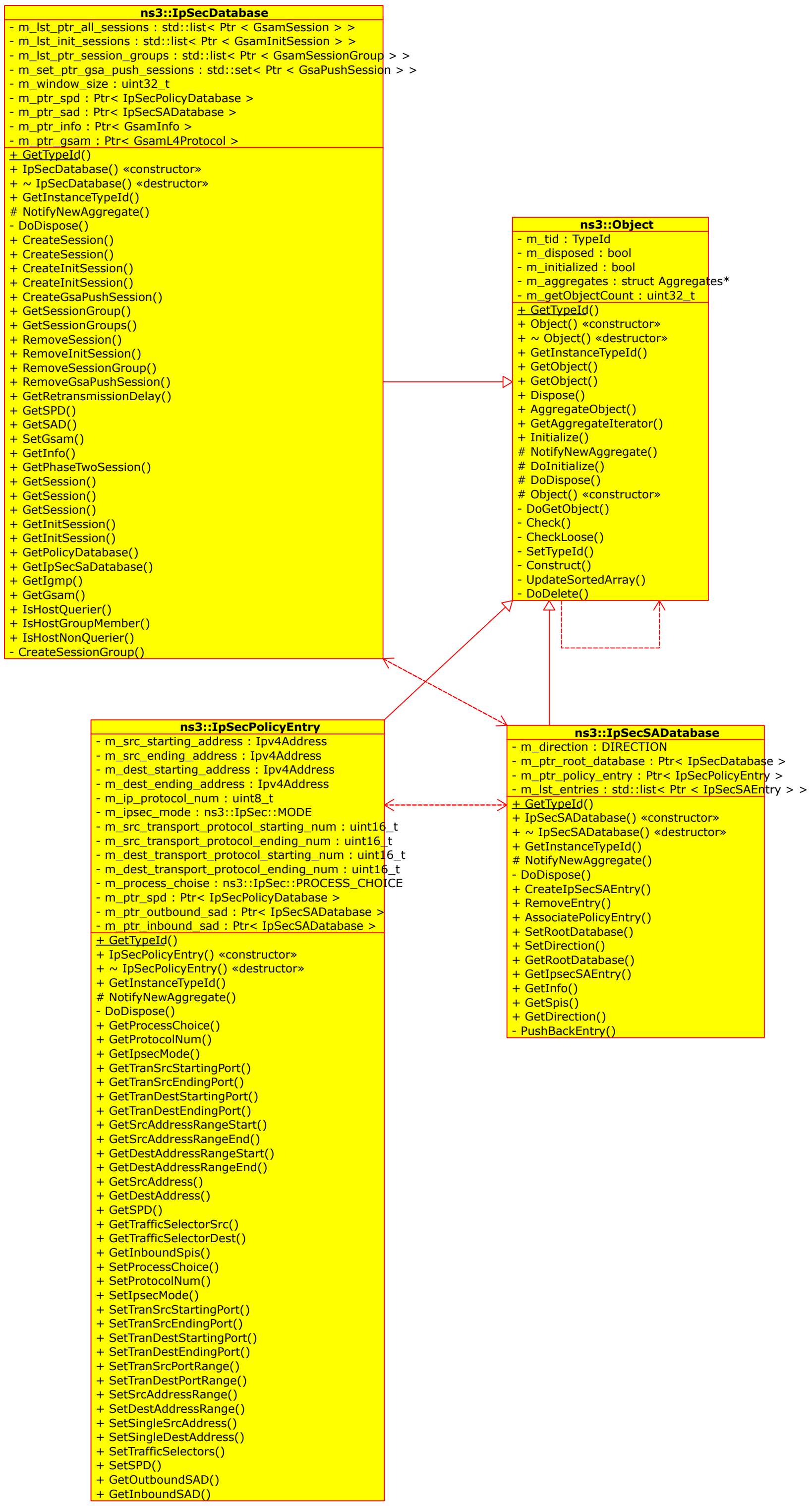


Figure A.7: The Class Diagrams of IPsec Databases Implementation Overview



ns3::IpSecDatabase
- m_lst_ptr_all_sessions : std::list< Ptr < GsamSession > >
- m_lst_init_sessions : std::list< Ptr < GsamInitSession > >
- m_lst_ptr_session_groups : std::list< Ptr < GsamSessionGroup > >
- m_set_ptr_gsa_push_sessions : std::set< Ptr < GsaPushSession > >
- m_window_size : uint32_t
- m_ptr_spd : Ptr< IpSecPolicyDatabase >
- m_ptr_sad : Ptr< IpSecSADatabase >
- m_ptr_info : Ptr< GsamInfo >
- m_ptr_gsam : Ptr< GsamL4Protocol >
+ GetTypeId()
+ IpSecDatabase() «constructor»
+ ~ IpSecDatabase() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ CreateSession()
+ CreateSession()
+ CreateInitSession()
+ CreateInitSession()
+ CreateGsaPushSession()
+ GetSessionGroup()
+ GetSessionGroups()
+ RemoveSession()
+ RemoveInitSession()
+ RemoveSessionGroup()
+ RemoveGsaPushSession()
+ GetRetransmissionDelay()
+ GetSPD()
+ GetSAD()
+ SetGsam()
+ GetInfo()
+ GetPhaseTwoSession()
+ GetSession()
+ GetSession()
+ GetSession()
+ GetInitSession()
+ GetInitSession()
+ GetPolicyDatabase()
+ GetIpSecSaDatabase()
+ GetIgmp()
+ GetGsam()
+ IsHostQuerier()
+ IsHostGroupMember()
+ IsHostNonQuerier()
- CreateSessionGroup()
ns3::Object
- m_tid : TypeId
- m_disposed : bool
- m_initialized : bool
- m_aggregates : struct Aggregates*
- m_getObjectCount : uint32_t
+ GetTypeId()
+ Object() «constructor»
+ ~ Object() «destructor»
+ GetInstanceTypeId()
+ GetObject()
+ GetObject()
+ Dispose()
+ AggregateObject()
+ GetAggregateIterator()
+ Initialize()
# NotifyNewAggregate()
# DoInitialize()
# DoDispose()
# Object() «constructor»
- DoGetObject()
- Check()
- CheckLoose()
- SetTypeId()
- Construct()
- UpdateSortedArray()
- DoDelete()
ns3::IpSecPolicyEntry
- m_src_starting_address : Ipv4Address
- m_src_ending_address : Ipv4Address
- m_dest_starting_address : Ipv4Address
- m_dest_ending_address : Ipv4Address
- m_ip_protocol_num : uint8_t
- m_ipsec_mode : ns3::IpSec::MODE
- m_src_transport_protocol_starting_num : uint16_t
- m_src_transport_protocol_ending_num : uint16_t
- m_dest_transport_protocol_starting_num : uint16_t
- m_dest_transport_protocol_ending_num : uint16_t
- m_process_choise : ns3::IpSec::PROCESS_CHOICE
- m_ptr_spd : Ptr< IpSecPolicyDatabase >
- m_ptr_outbound_sad : Ptr< IpSecSADatabase >
- m_ptr_inbound_sad : Ptr< IpSecSADatabase >
+ GetTypeId()
+ IpSecPolicyEntry() «constructor»
+ ~ IpSecPolicyEntry() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ GetProcessChoice()
+ GetProtocolNum()
+ GetIpsecMode()
+ GetTranSrcStartingPort()
+ GetTranSrcEndingPort()
+ GetTranDestStartingPort()
+ GetTranDestEndingPort()
+ GetSrcAddressRangeStart()
+ GetSrcAddressRangeEnd()
+ GetDestAddressRangeStart()
+ GetDestAddressRangeEnd()
+ GetSrcAddress()
+ GetDestAddress()
+ GetSPD()
+ GetTrafficSelectorSrc()
+ GetTrafficSelectorDest()
+ GetInboundSpis()
+ SetProcessChoice()
+ SetProtocolNum()
+ SetIpsecMode()
+ SetTranSrcStartingPort()
+ SetTranSrcEndingPort()
+ SetTranDestStartingPort()
+ SetTranDestEndingPort()
+ SetTranSrcPortRange()
+ SetTranDestPortRange()
+ SetSrcAddressRange()
+ SetDestAddressRange()
+ SetSingleSrcAddress()
+ SetSingleDestAddress()
+ SetTrafficSelectors()
+ SetSPD()
+ GetOutboundSAD()
+ GetInboundSAD()
ns3::IpSecSADatabase
- m_direction : DIRECTION
- m_ptr_root_database : Ptr< IpSecDatabase >
- m_ptr_policy_entry : Ptr< IpSecPolicyEntry >
- m_lst_entries : std::list< Ptr < IpSecSAEntry > >
+ GetTypeId()
+ IpSecSADatabase() «constructor»
+ ~ IpSecSADatabase() «destructor»
+ GetInstanceTypeId()
# NotifyNewAggregate()
- DoDispose()
+ CreateIpSecSAEntry()
+ RemoveEntry()
+ AssociatePolicyEntry()
+ SetRootDatabase()
+ SetDirection()
+ GetRootDatabase()
+ GetIpsecSAEntry()
+ GetInfo()
+ GetSpis()
+ GetDirection()
- PushBackEntry()

Figure A.8: The Class Diagrams of SPD Implementation

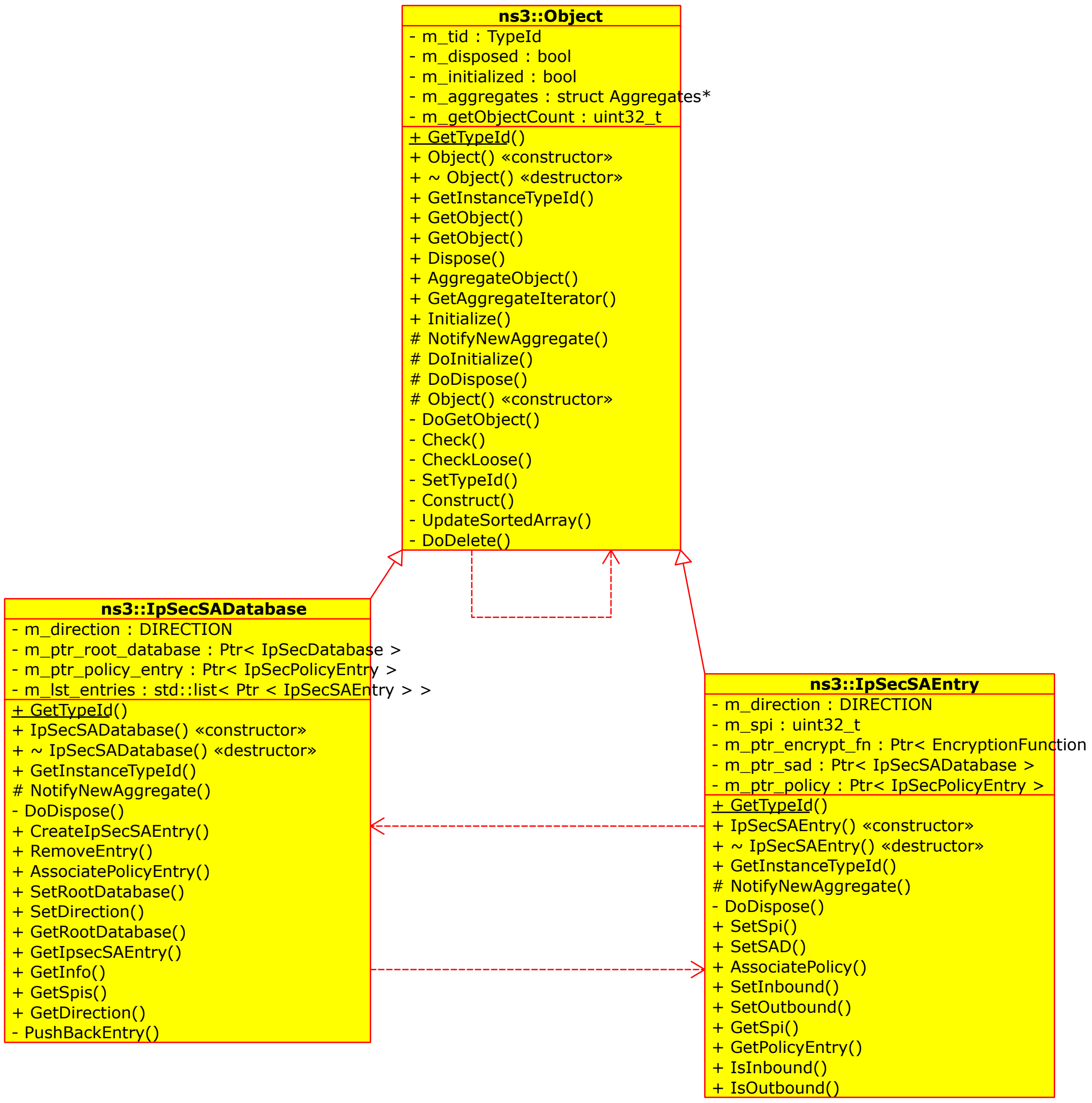


Figure A.9: The Class Diagrams of SAD Implementation

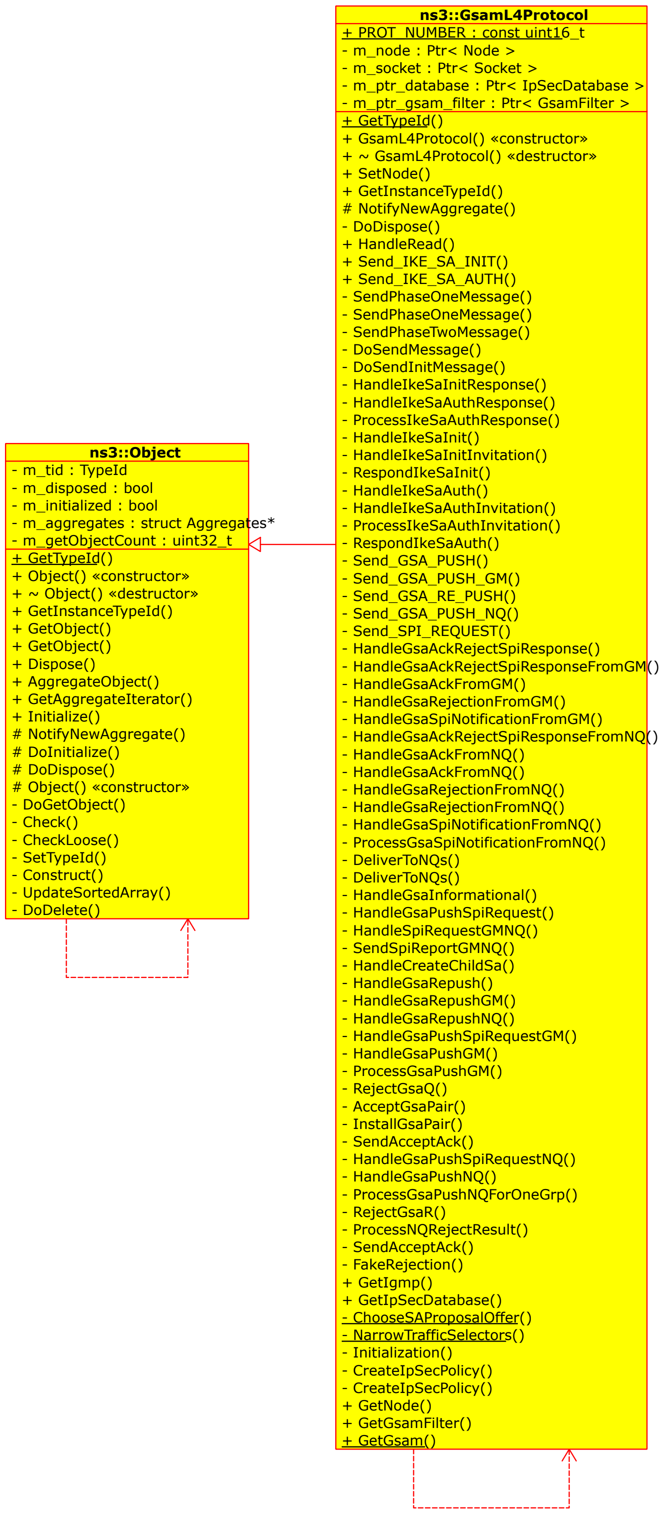


Figure A.10: The Class Diagrams of GSAM Protocol Implementation

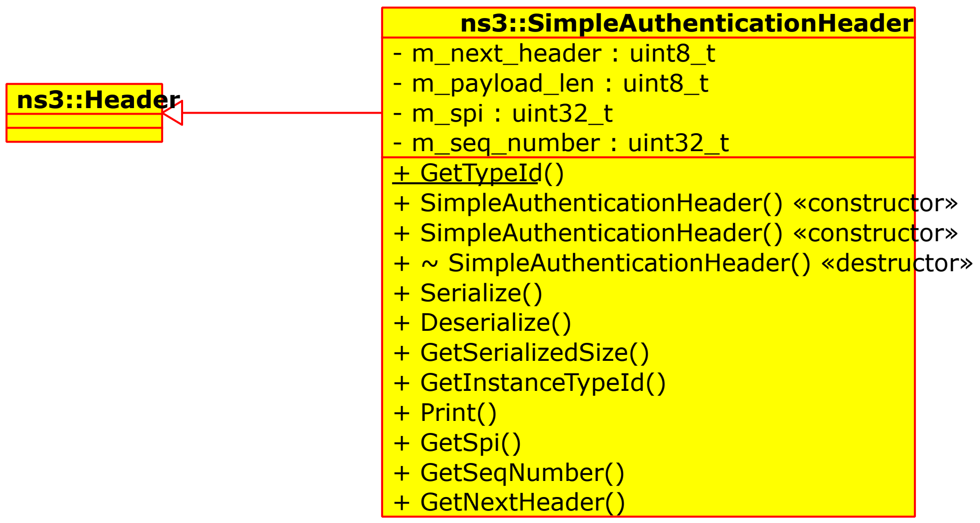

ns3::Header
ns3::SimpleAuthenticationHeader
- m_next_header : uint8_t
- m_payload_len : uint8_t
- m_spi : uint32_t
- m_seq_number : uint32_t
+ GetTypeId()
+ SimpleAuthenticationHeader() «constructor»
+ SimpleAuthenticationHeader() «constructor»
+ ~ SimpleAuthenticationHeader() «destructor»
+ Serialize()
+ Deserialize()
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Print()
+ GetSpi()
+ GetSeqNumber()
+ GetNextHeader()

Figure A.11: The Class Diagram of Simple AH Implementation

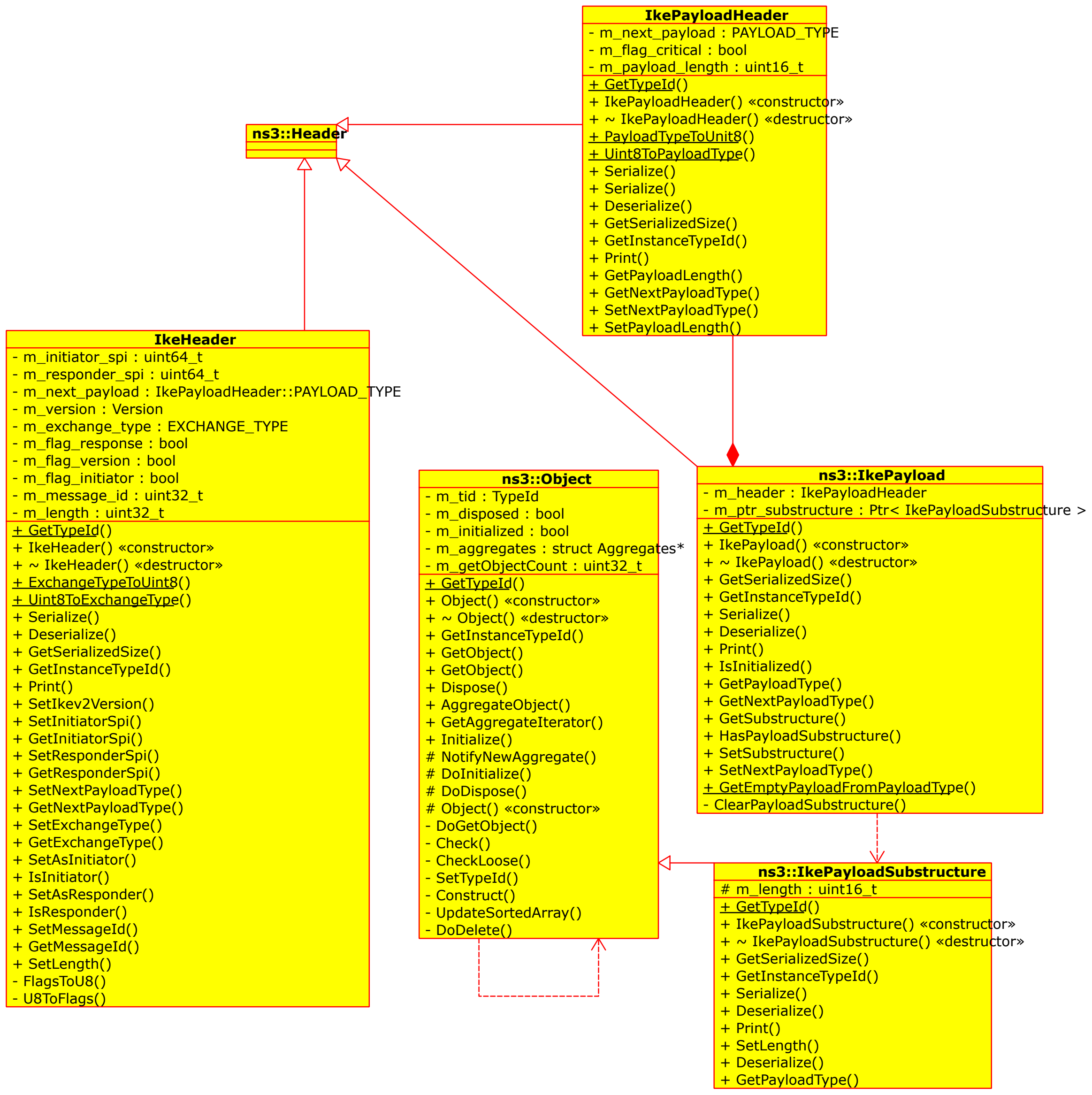


Figure A.12: The Class Diagrams of IKEv2 and GSAM Packet Data Structures Implementation Overview

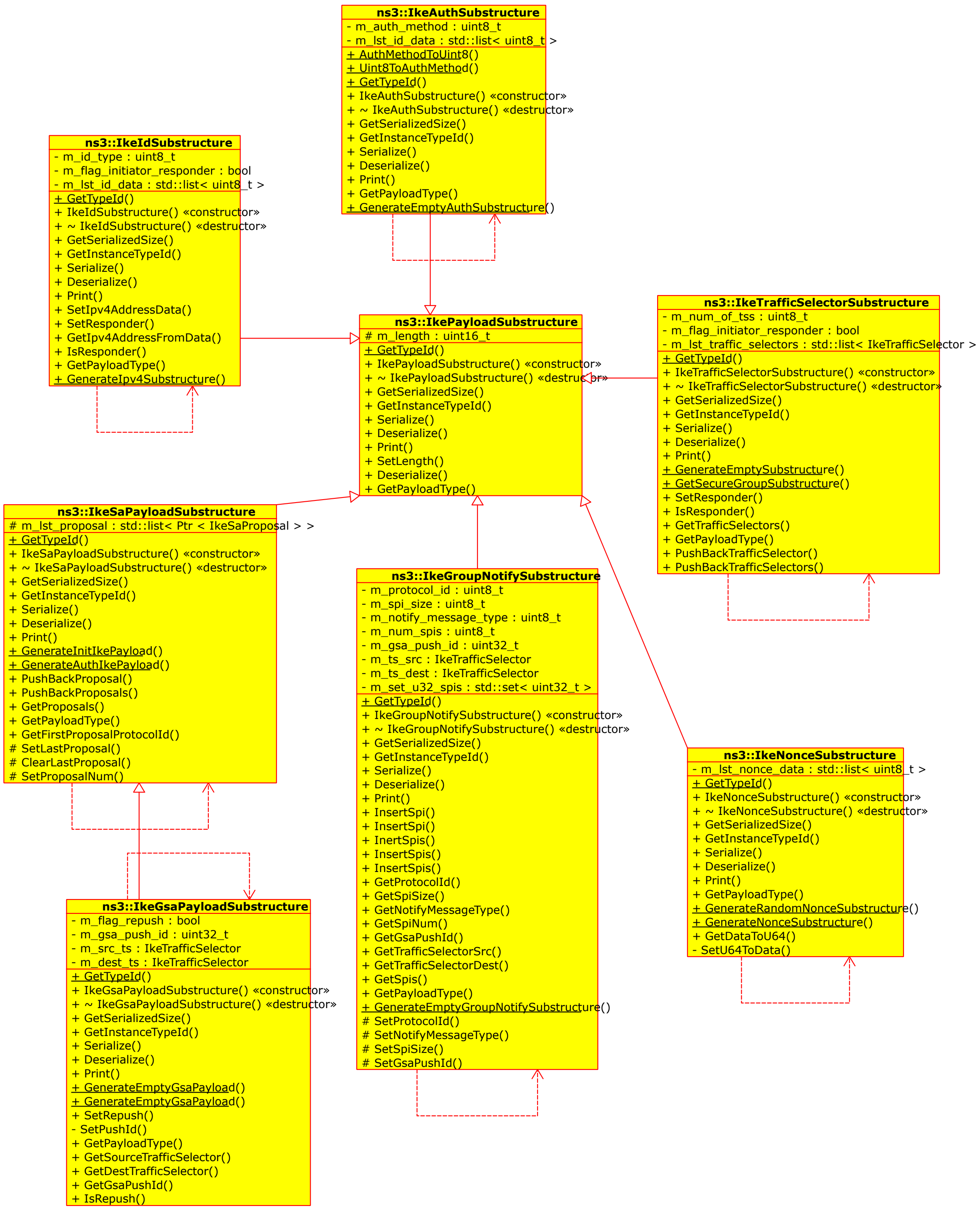

ns3::IkeAuthSubstructure
- m_auth_method : uint8_t
- m_lst_id_data : std::list< uint8_t >
+ AuthMethodToUint8()
+ Uint8ToAuthMethod()
+ GetTypeId()
+ IkeAuthSubstructure() «constructor»
+ ~ IkeAuthSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ GetPayloadType()
+ GenerateEmptyAuthSubstructure()
ns3::IkeIdSubstructure
- m_id_type : uint8_t
- m_flag_initiator_responder : bool
- m_lst_id_data : std::list< uint8_t >
+ GetTypeId()
+ IkeIdSubstructure() «constructor»
+ ~ IkeIdSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ SetIpv4AddressData()
+ SetResponder()
+ GetIpv4AddressFromData()
+ IsResponder()
+ GetPayloadType()
+ GenerateIpv4Substructure()
ns3::IkePayloadSubstructure
# m_length : uint16_t
+ GetTypeId()
+ IkePayloadSubstructure() «constructor»
+ ~ IkePayloadSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ SetLength()
+ Deserialize()
+ GetPayloadType()
ns3::IkeTrafficSelectorSubstructure
- m_num_of_tss : uint8_t
- m_flag_initiator_responder : bool
- m_lst_traffic_selectors : std::list< IkeTrafficSelector >
+ GetTypeId()
+ IkeTrafficSelectorSubstructure() «constructor»
+ ~ IkeTrafficSelectorSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ GenerateEmptySubstructure()
+ GetSecureGroupSubstructure()
+ SetResponder()
+ IsResponder()
+ GetTrafficSelectors()
+ GetPayloadType()
+ PushBackTrafficSelector()
+ PushBackTrafficSelectors()
ns3::IkeSaPayloadSubstructure
# m_lst_proposal : std::list< Ptr < IkeSaProposal > >
+ GetTypeId()
+ IkeSaPayloadSubstructure() «constructor»
+ ~ IkeSaPayloadSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ GenerateInitIkePayload()
+ GenerateAuthIkePayload()
+ PushBackProposal()
+ PushBackProposals()
+ GetProposals()
+ GetPayloadType()
+ GetFirstProposalProtocolId()
# SetLastProposal()
# ClearLastProposal()
# SetProposalNum()
ns3::IkeGroupNotifySubstructure
- m_protocol_id : uint8_t
- m_spi_size : uint8_t
- m_notify_message_type : uint8_t
- m_num_spis : uint8_t
- m_gsa_push_id : uint32_t
- m_ts_src : IkeTrafficSelector
- m_ts_dest : IkeTrafficSelector
- m_set_u32_spis : std::set< uint32_t >
+ GetTypeId()
+ IkeGroupNotifySubstructure() «constructor»
+ ~ IkeGroupNotifySubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ InsertSpi()
+ InsertSpi()
+ InertSpis()
+ InsertSpis()
+ InsertSpis()
+ GetProtocolId()
+ GetSpiSize()
+ GetNotifyMessageType()
+ GetSpiNum()
+ GetGsaPushId()
+ GetTrafficSelectorSrc()
+ GetTrafficSelectorDest()
+ GetSpis()
+ GetPayloadType()
+ GenerateEmptyGroupNotifySubstructure()
# SetProtocolId()
# SetNotifyMessageType()
# SetSpiSize()
# SetGsaPushId()
ns3::IkeNonceSubstructure
- m_lst_nonce_data : std::list< uint8_t >
+ GetTypeId()
+ IkeNonceSubstructure() «constructor»
+ ~ IkeNonceSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ GetPayloadType()
+ GenerateRandomNonceSubstructure()
+ GenerateNonceSubstructure()
+ GetDataToU64()
- SetU64ToData()
ns3::IkeGsaPayloadSubstructure
- m_flag_repush : bool
- m_gsa_push_id : uint32_t
- m_src_ts : IkeTrafficSelector
- m_dest_ts : IkeTrafficSelector
+ GetTypeId()
+ IkeGsaPayloadSubstructure() «constructor»
+ ~ IkeGsaPayloadSubstructure() «destructor»
+ GetSerializedSize()
+ GetInstanceTypeId()
+ Serialize()
+ Deserialize()
+ Print()
+ GenerateEmptyGsaPayload()
+ GenerateEmptyGsaPayload()
+ SetRepush()
- SetPushId()
+ GetPayloadType()
+ GetSourceTrafficSelector()
+ GetDestTrafficSelector()
+ GetGsaPushId()
+ IsRepush()

Figure A.13: The Class Diagrams of IKEv2 and GSAM Packet Payload Substructures Implementation

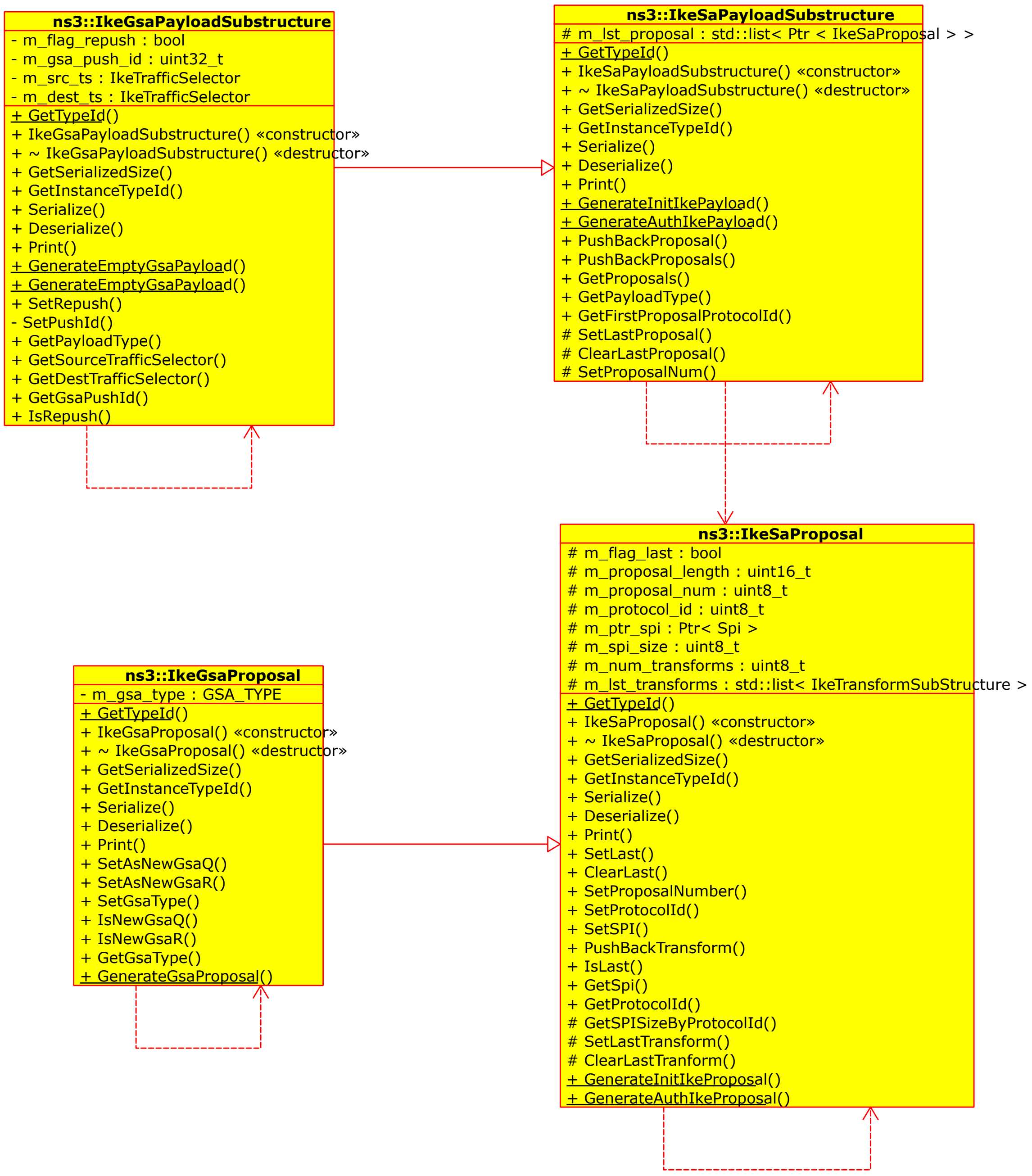


Figure A.14: The Class Diagrams of SA and GSA Payloads Substructures Implementation

References

Atwood, J. W. (2007). An architecture for secure and accountable multicasting. In *Local computer networks, 2007. lcn 2007. 32nd ieee conference on* (pp. 73–78).

Bhattacharyya, S. (2003). An overview of source-specific multicast (SSM), RFC 3569.

CALOMEL. (2017). *AES-NI SSL Performance.* Retrieved from https://calomel.org/aesni_ssl_performance.html ([Online; accessed 2017])

Class diagram. (2017). *Class diagram — Wikipedia, the free encyclopedia.* Retrieved from https://en.wikipedia.org/wiki/Class_diagram ([Online; accessed 2017])

Deering, S. E. (1988). Host extensions for IP multicasting, RFC 1112.

Fenner, W. C. (1997). Internet group management protocol, version 2, RFC 2236.

IoT Business Unit, ARM. (2015). *Performance of State-of-the-Art Cryptography on ARM-based Microprocessors.* Retrieved from http://csrc.nist.gov/groups/ST/lwc-workshop2015/presentations/session7-vincent.pdf ([Online; accessed 2017])

Kent, S. (2005a). IP authentication header (AH), RFC 4302.

Kent, S. (2005b). IP encapsulating security payload (ESP), RFC 4303.

Kivinen, T., Hoffman, P., Kaufman, C., Nir, Y., & Eronen, P. (2014). Internet Key Exchange Protocol Version 2 (IKEv2), RFC 7296.

Kouvelas, I., Cain, B., Fenner, B., Deering, S., & Thyagarajan, A. (2002). Internet group management protocol, version 3, RFC 3376.

Li, B., & Atwood, J. W. (2016). Secure receiver access control for IP multicast at the network level: Design and validation. *Computer Networks*, *102*, 109–128.

Manzato, D. A., & da Fonseca, N. L. (2013). A survey of channel switching schemes for IPTV. *IEEE Communications Magazine*, *51*(8), 120–127.

Nsnam. (2015). WHAT IS NS-3. https://www.nsnam.org/overview/what-is-ns-3/. ([Online; accessed 2017])

Seo, K., & Kent, S. (2005). Security architecture for the internet protocol, RFC 4301.

Unified Modeling Language. (2017). *Unified modeling language — Wikipedia, the free encyclopedia.* Retrieved from https://en.wikipedia.org/wiki/Unified_Modeling_Language ([Online; accessed 2017])

Zion. (2016). IPTV Market for Advertising and Marketing, Media and Entertainment, Gaming, E-Commerce, Healthcare and Medical, Telecommunication & It and Others - Global Industry Perspective, Comprehensive Analysis, and Forecast, 2015 2021. http://www.marketresearchstore.com/report/iptv-market-z59822. ([Online; accessed 2017])