IPSec Based Bluetooth Security Architecture

Developed by

Al-Muthanna Suleiman Al-Hawamdeh
(208 – FAS/MSCS/04)

Adil Khan
(175 – FAS/MSCS/04)

Supervised by
Dr. M. Sikander Hayat Khiyal

Department of Computer Science
International Islamic University, Islamabad.

(2006)

بسم الله الرحمن الرحيم

International Islamic University, Islamabad

Date: March 3, 2007

Final Approval

It is certified that we have read the thesis titled **"IPSec Based Bluetooth Security Architecture"** submitted by Mr. Al-Muthanna Sulieman Al-Hawamdeh Reg. No. 208-FAS/MSCS/04 and Mr. Adil Khan Reg. No. 175-FAS/MSCS/04 which, in our judgment this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad for the award of MS in Computer Sciences.

Committee

External Examiner
Prof. Dr Abdus Sattar
Former D.G,
Pakistan Computer Bureau,
Islamabad

[illegible]

Internal Examiner
Prof. Dr. Syed Afaq Hussain
Head,
Department of Electronics Engineering,
International Islamic University, Islamabad

[illegible]

Supervisor
Dr. M Sikander Hayat Khiyal
Head,
Dept. Computer Science,
International Islamic University, Islamabad.

[illegible]

Dedicated to

Our Parents,

Teachers

&

Friends

ACKNOWLEDGEMENTS

All praise to Allah, the Almighty, the most merciful and compassionate, who granted us perseverance to accomplish this research work.

We express our profound gratitude to our kind supervisor **Dr. M.Sikander Hayat Khiyal.** His motivation guided us to this achievement and kept our morale high with his valuable suggestions and personal encouragement. He was readily available for consultations, without which, we never have been able to develop this software.

We are also thankful to **Dr. Faten Kharbat** (CS Assistant Professor at Zarqa Private University-Jordan) for her inspiring attitude and untiring help during through the project efforts. She did a lot of efforts for our success.

We would like also to thank our parents, teachers, friends and all those who helped us and supported us during the project.

Al-Muthanna Sulieman
208-FAS/MSCS/04

Adil Khan
175-FAS/MSCS/04

DECLARATION

We, hereby declare that this research, neither as a whole nor as a part, has been copied from any source. It is further declared that we have developed this research entirely based on our personel efforts under the able guidence of our supervisor, Dr. M Sikander Hayat Khiyal. If any part of this thesis is proved to be copied or reported at any stage, we accept the responsibility to face the subsequant consequences. No part of this work inscribed in this thesis has either been submitted to any other university for the award of degree/qualification.

Al-Muthanna Suleiman
208-FAS/MSCS/04

Adil Khan
175-FAS/MSCS/04

A dissertation submitted to the
Department of Computer Science,
International Islamic University, Islamabad,
As a partial fulfillment of the requirements
For the award of the degree of
MS in Computer Science

Project in Brief

Project Title:	**IPSec Based Bluetooth Security Architecture**
Objective:	The primary goal of the research was to develop an understanding between IPSec Protocol and Bluetooth technology and study the performance of the new developed architecture.
Organization:	Department of Computer Science, International Islamic University, Islamabad.
Undertaken By:	Al-Muthanna Suleiman (208 - FAS/MSCS/04) Adil Khan (175 - FAS/MSCS/04)
Supervised By:	Dr. M.Sikander Hayat Khiyal Dead, Department of Computer Science, International Islamic University, Islamabad.
Technologies:	BNEP (Bluetooth Network Encapsulation Protocol)
Language:	C, C++
Tool Used:	Network Simulator NS-2, UCBT
Operating System:	Windows XP
System Used:	Pentium III
Date Started:	February 2006
Date Completed:	September 2006

Table of Contents

Chapter No **Contents** **Page No**

1. INTRODUCTION ....
1.1 Bluetooth System Basics 1
1.1.1 Background 1
1.1.2 Trade-OffsS 1
1.1.3 Bluetooth Protocol StackS 2
1.1.4 Physical Layer 3
1.1.5 Baseband 4
1.1.6 Link Manager Protocol 8
1.1.7 Logical Link Control & Adaptaion Protocol 11
1.1.8 Host Control Interface 11
1.1.9 Profiles 12
1.2 Bluetooth Security Basics 14
1.2.1 User Scenarios 14
1.2.2 Notions and Terminology 16

2. LITERATURE SURVEY ....
2.1 Key Types 21
2.2 Pairing and User Ineraction 22
2.3 Authentication 24
2.4 Link Privacy 24
2.4.1 Protect The Link 25
2.4.2 Encryption Algorithm 25
2.4.3 Mode Of Operations 27
2.4.4 Unicast and Broadcast 28
2.5 Communication security Policies 29
2.5.1 Security Modes 30
2.5.2 Security Policy Managment 33

3. IPSEC PROTOCOL ....
3.1 System Overview 36
3.1.1 What IPSec Does 36
3.1.2 How IPSec Works 37
3.2 Authentication Header(AH) 38
3.2.1 AH Modes 38

3.2.2 Intergrity Protection Process 38
3.2.3 AH Header 39
3.2.4 How AH Works 40
3.2.5 AH Version 3 41
3.2.6 AH Summary 40
3.3 Encapsulating Security Payload(ESP) 42
3.3.1 ESP Modes 42
3.3.2 Encryption Process 43
3.3.3 ESP Packet Fields 43
3.3.4 How ESP Works 45
3.3.5 ESP Version 3 46
3.3.6 ESP Summary 46
3.4 MD-5 Algorithm 47
3.4.1 Summary 49
3.5 SHA-1 Algorithm 50
3.5.1 Message Padding 50
3.5.2 Functions and Constants used 52
3.5.3 Computing the Message Digest 52
3.5.3.1 Method 1 52
3.5.3.2 Method 2 53
3.6 DES Algorithm 53
3.7 Triple DES Algorithm 61
3.7.1 In Depth 62
3.7.2 Modes of Operations 63
3.7.2.1 Triple ECB (Electronic Code Book) 63
3.7.2.2 Triple CBC (Cipher Block Chaining) 63

4. METHODOLOGY ....
4.1 Security Weaknesses 65
4.1.1 Security Depends on Security of PIN 65
4.1.2 Unit Key 65
4.1.3 Encryption Algorithms 66
4.1.4 Denial Of Service Attacks 66
4.1.5 Location Privacy 67
4.1.6 Other Security Problems 67
4.2 Bluetooth Network Encapsulation Protocol 68
4.2.1 BNEP Header Format 68
4.2.2 Packet Encapsulation 69
4.3 Suggested Solution 70

4.3.1 AH and BNEP....71
4.3.2 ESP and BNEP....71
4.4 System Design....72
4.4.1 Security Policy & Association Managment....75
4.4.1.1 Basic Concept of Security Association....75
4.4.1.2 SPD Outbound Processing....76
4.4.1.3 SPD/SAD Inbound Processing....76
4.4.2 AH Processing....78
4.4.2.1 AH Inbound Processing....78
4.4.2.2 AH Outbound Processing....79
4.4.3 ESP Processing....79
4.4.3.1 ESP Inbound Processing....80
4.4.3.2 ESP Outbound Processing....80

5. IMPLEMENTATION....
5.1 Network Simulator NS-2....83
5.1.1 Introduction....83
5.1.2 Functionality....83
5.2 UCBT....84
5.2.1 Introduction....84
5.2.2 Functionality....84
5.2.3 Interface & Implementation....86

6. TESTING....
6.1 Structural (White Box) Testing....89
6.1.1 Why Structural Testing....89
6.1.2 How We Implemanted Structural Testing....89
6.2 Functional (Black Box) Testing....90
6.2.1 Why Funtional Testing....90
6.2.2 How We Implemanted Functional Testing....90
6.3 Interoperability Testing....92
6.3.1 Testing Enviroments....92
6.3.1.1 IPSec_Tunnel for Linux....92
6.3.1.2 FreeS/WAN....93
6.3.1.3 Windows 2000....94
6.3.1.4 PGPnet....94
6.3.2 Security Tests....94
6.3.2.1 Packet Was Altered During Transmission....94
6.3.2.2 Non-IPSec Packet , Which Should be one....95

6.3.2.3 Packet are Resent....95
6.3.2.4 Packet that are out of the Window....95
6.3.2.5 Packet With Bad SPI....96
6.4 Specification Testing....96
6.5 Regression Testing....97
6.6 Acceptance Testing....97
6.7 Assertion Testing....97
6.8 Unit Testing....97
6.9 System Testing....97
6.10 System Evaluation....97

7. PERFORMANCE....
7.1 Features....99
7.2 Analysis & Results....100
7.3 Conclusion....105

8. REFRENCES & BIBLIOGRAPHY....107

9. APPENDICES....
9.A User Manual....111
9.B Code....125

10. PUBLICATION....

CHAPTER 1

Introduction

1. Introduction

Bluetooth wireless technology is gradually becoming a popular way to replace existing wire line connections with short-range wireless interconnectivity. It is also an enabling technology for new types of applications. In this chapter we give a short background and a condensed description of how the Bluetooth system works. We will focus on details that directly or indirectly relate to security issues and on the functionality that is important in order to understand the concept of the technology. The reference documentation for Bluetooth wireless technology is given in [1].

1.1 Bluetooth system basics

1.1.1 Background

Bluetooth wireless technology is a short-range radio technology that is designed to fulfill the particular needs of wireless interconnections between different personal devices, which are very popular in today's society. The development of Bluetooth started in the mid-1990s, when a project within Ericsson Mobile Communications required a way to connect a keyboard to a computer device without a cable. The wireless link turned out to be useful for many other things, and it was developed into a more generic tool for connecting devices. A synchronous mode for voice traffic was added and support for up to seven slaves was introduced. In order to gain momentum for the technology and to promote acceptance, the Bluetooth Special Interest Group (SIG) was founded in 1998.

The group consists of many companies from various fields. By joining forces, the SIG members have evolved the radio link to what is now known as Bluetooth wireless technology.

1.1.2 Trade-offs

Bluetooth wireless technology is targeting devices with particular needs and constraints. The main issues are, as with all battery-powered consumer electronics, cost and power consumption. Consequently, certain design trade-offs have been made between the cost and power consumption on one side and overall performance on the other. For instance, some of the specified requirements for the radio (particularly the sensitivity numbers) are chosen to be so relaxed that it is possible to implement a rather cheap one-chip radio with very few external components (such as filters).

The price paid is in a shortening of the range, as it will decrease with decreased sensitivity. On the other hand, some requirements are quite stringent (e.g. adjacent channel rejection) in order to handle interference at frequencies near the intended signal. This helps to keep up the aggregated throughput when many links are running simultaneously. One major design goal is to have the system quite robust in noisy environments. This is because interference rather than range is expected to be the limiting factor of the perceived performance.

In contrast to most other well-known radio standards used for data communication [e.g., Institute of Electrical and Electronics Engineers (IEEE) 802.11b and HIPERLAN], the specification has been written from the beginning with use cases for handheld personal devices in mind. In particular, there is no need to have an infrastructure (i.e., base stations) in place. The flexible Bluetooth master-slave concept was introduced to fit well in a dynamically changing constellation of devices that communicate with each other. Furthermore, due to the wide range of requirements for the traffic types for different applications, Bluetooth can handle various data transport channels: asynchronous, isochronous, and synchronous. It is even possible for a device to mix asynchronous (data) and synchronous (voice) traffic at the same time.

In a radio environment where communication links are set up on request rather than by default (without the need for a centralized infrastructure, as in cellular networks) and where any node is able to communicate with any other node, networking is usually called *ad hoc networking* or *ad hoc connectivity*. As we will discuss later in the thesis, ad hoc connections impose special requirements for the security functionality for the system. Bluetooth wireless technology is particularly well suited for ad hoc usage scenarios.

1.1.3 Bluetooth protocol stack

The Bluetooth system stack is layered according to Figure 1.1. At the bottom is the *physical layer*, which is basically the modem part. This is where the radio signals are processed. The fundamental limits on sensitivity (range) and interference rejection are set by the radio front end (noise figure) and filters implemented in this layer.

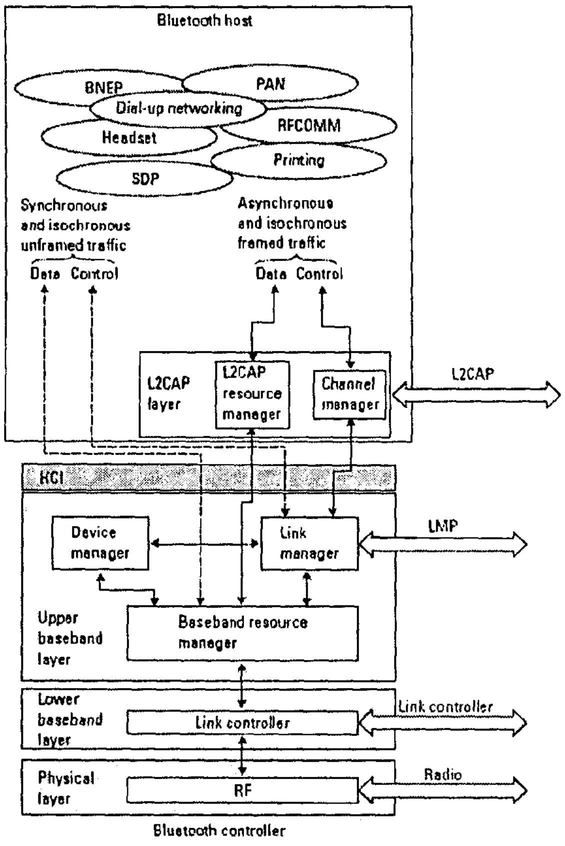


Figure 1.1 A schematic view of the Bluetooth protocol stack architecture [1].

Above the physical layer is the *baseband layer*, which is divided into lower and upper parts. In the following, we will not differentiate between these, but simply refer to them as the baseband. It is at this layer that packets are formatted: creation of headers, checksum calculations, retransmission procedure, and, optionally, encryption and decryption are handled. The *link controller* (LC) is the entity that implements the baseband protocol and procedures.

Bluetooth links are managed by the *link manager* (LM). The devices set up links, negotiate features, and administer connections that are up and running using the *link manager protocol* (LMP).

Large chunks of user data need to be reformatted into smaller units before they can be transmitted over the Bluetooth link. It is the responsibility of the *logical link communication and adaptation protocol* (L2CAP) to take care of this.

At this layer it is possible to ask for certain *quality-of-service* (QoS) values one would like to reserve for the link. In many cases, the Bluetooth functionality is to be integrated into a host entity that has computational power but lacks the radio part. For this purpose, *Bluetooth modules* handling only the lower layers exist. The entity handling the functionality of these layers is sometimes referred to as the *Bluetooth controller*.

For instance, a laptop that is perfectly capable of handling higher protocol layers can embed a module that handles radio, baseband, and L2CAP. In such a setup, the higher layers that are implemented in the host entity will communicate with the lower layers of the module through the *host controller interface* (HCI).

1.1.4 Physical layer

Bluetooth radio operates in the license-free and globally available *industrial, scientific, and medical* (ISM) band at 2.4 GHz. Because the ISM band is free, Bluetooth has to share this frequency band with many other systems. Various wireless communication systems operate in this band (besides Bluetooth, IEEE 802.11b, most notably). Other systems may be defined in the future. One other common device emitting radio frequency power in this band is found in almost all homes: the microwave oven. Even though the vast majority of the radiation is absorbed by the food inside the oven, some of it leaks and will appear outside as interference. Actually, the leakage may be as much as 1,000 times more powerful than the signal one tries to capture, so this interference cannot be neglected.

Fortunately, the interference is not there all the time (loosely speaking, the radiation cycle follows the frequency of the power supply) and is not over the entire frequency spectrum (approximately 15 to 20 MHz of the frequency band is affected by the microwave oven).

All in all, it is very hard to predict what kind of interference to expect in the ISM band. To combat this, Bluetooth deploys a *frequency hopping* (FH) spread spectrum technology. There are 79 channels used, each with a bandwidth of 1 MHz. During communication, the system makes 1,600 hops per second evenly spread over these channels according to a pseudorandom pattern. The idea is that if one transmits on a bad channel, the next hop, which is only 625 μs 6 Bluetooth Security later, will hopefully, be on a good channel. In general, faster hopping between frequencies gives more spreading, this improves on protection from other interference. However, the improved performance comes at the cost of increased complexity. The hopping rate chosen for Bluetooth is considered to be a good trade-off between performance and complexity.

The signal is transmitted using binary *Gaussian frequency shift keying*. The raw bit rate is 1 Mbps, but due to various kinds of protocol overhead, the user data rate cannot exceed 723 Kbps. Following regulatory bodies in different parts of the world, the maximum transmit power is restricted to 100 mW (or, equivalently, 20 dBm). It is expected that this will give a range of 100m at line of sight.

Another power class, where the output power is restricted to 1 mW (0 dBm), is also defined. Radios of this power class are more common in handheld devices, and they will have a range of approximately 10m at line of sight.

One should notice that the specification defines the sensitivity level for the radio such that the raw *bit error rate* (BER) 10−3 is met, which translates into the range numbers given above within the specified link budget. It is around this raw BER that a voice link without error-correcting capabilities becomes noticeably distorted. This is a major reason for the choice of the BER 10−3 as a benchmark number for the radio specification. However, for data traffic, Bluetooth applies *cyclic redundancy check* (CRC) as well as optional error correction codes.

Thus, if the receiver detects a transmission error, it will request a retransmission. The result is that when operating at BER 10−3 (and even worse, to some extent), a data link will function quite well anyway. Depending on payload lengths and packet types, the decrease in throughput may even be unnoticed by the user.

This is, of course, good for the users, but also for potential eavesdroppers, who may be able to choose a position at a safe distance beyond the specified range for their purposes.

1.1.5 Baseband

Addressing and setting up connections Each Bluetooth radio comes with a unique, factory preset 48-bit address. This address, known as the *Bluetooth device address* (*BD_ADDR*), constitutes the basis for identification of devices when connections are established. Before any connection can be set up, the *BD_ADDR* of the addressee must be known to the side that initiates a connection. For first-time connections, this is accomplished by having the initiating side collect the device addresses of all nearby units and then individually address the one of interest. This step is known as the *inquiry procedure*. Naturally, once this has been done, the information gathered can be reused without the need for another inquiry at the next connection attempt to one of the known devices.

The first step in finding other devices is to send an inquiry message. This message is repeatedly transmitted following a well-defined, rather short hop sequence of length 32. Any device that wants to be visible to others (also known as being *discoverable*) frequently scans the inquiry hop sequence for inquiry messages.

This procedure is referred to as *inquiry scan*. A scanning device will respond to inquiries with its *BD_ADDR* and the current value of its native clock. The inquiry message is anonymous and there is no acknowledgment to the response, so the scanning device has no idea who made the inquiry, nor if the inquirer received the response correctly.

The inquirer gathers responses for a while and can, when so desired, reach a particular device through a *page* message. This message is sent on another length 32 hop sequence determined from the 24 least significant bits of the *BD_ADDR* [these are denoted by *lower address part* (LAP)] of the target device.

A device listens for page messages when it is in the *page scan* state. The phase (i.e., the particular position) of the FH sequence is determined from the device's native clock. The paging device has knowledge of this from the inquiry response; thus it is possible for the paging device to hit the correct frequency of the paged device fairly quickly. As already has been stated, the inquiry part can be bypassed when two units have set up a connection before and want to connect again. If a long time has passed since the previous connection, the clocks of the devices may have drifted, causing the estimate of the other unit's native clock to be inaccurate. The only effect of this is that the connection set-up time may increase because of the resulting misalignment of their respective phase in the page hop sequence.

When a page response is received, a rough FH synchronization has been established between the pager and the paged device. By definition, the pager is the *master* and the paged device is the *slave*. The meaning of these terms will be discussed in the next section. Before the channel can be set up, some more information must be exchanged between the devices. The FH sequence, the timing, and the *channel access code* (CAC) are all derived from the master device. In order to fine tune the FH synchronization, the slave needs the *BD_ADDR* and the native clock of the master. This information is conveyed in a special packet sent from the master to the slave. With all information at hand at the slave side, the master and slave can switch from the page hopping sequence (defined by the slave) to the basic channel hopping sequence determined by the master's parameters. Details on this process can be found in [2].

1.1.5.1 Topology and medium access control

Networks are formed using a star topology in Bluetooth. Not more than eight simultaneous devices can participate in one of these *piconets*. The central node of the piconet is called a *master* and the other nodes are called *slaves*. Thus, a piconet will have exactly one master and at least one but at most seven slaves. The 8 Bluetooth Security simplest form of piconet is illustrated in Figure 1.2(a). Information exchange within the piconet is done by sending packets back and forth between devices.

Full duplex is accomplished using a *time division duplex* mode; that is, the channel access is divided into time slots assigned to the communicating parties. Who gets access to the channel is determined by the piconet master simply by addressing a slave, which will then have the right to send in the next time slot.

Being in connection state, the piconet devices follow a long deterministic FH sequence determined from the master's LAP and native clock. The length of this sequence is 223, which roughly corresponds to a 23-hour cycle. Following from the fact that a device can only be master of one piconet at a time, every piconet will have different FH sequences. To stay tuned to its piconet, each slave member must continuously adjust for clock drift to the master by monitoring the traffic sent over the channel. Only master-to-slave and slave-to-master communication is possible.

Consequently, slave-to-slave traffic must be relayed via the master. If one particular device is involved in all traffic, there is a risk that it becomes a bottleneck for the data transfer. This property is suboptimal with respect to the aggregated system throughput. However, an important concept in Bluetooth is that all devices have the ability to take the role of either slave or master, so the slaves may choose to create another piconet. Doing so is better for the aggregated throughput, since quite many piconets can actually be operated in parallel before mutual interference cancels the benefits inherent in the parallelism. This principle is shown in Figure 1.2(b).

In principle, a Bluetooth device is allowed to participate in more than one piconet simultaneously, as illustrated in Figure 1.2(c). This is accomplished using time sharing between the different piconets. To accommodate for this, the low-power modes *hold*, *park*, and *sniff* can be used. Without going into detail, these modes make it possible for a device to temporarily leave a piconet to do something else (e.g., to sleep to save power or join another piconet). Thus, by having one device is a member of two piconets, it is possible to exchange information between piconets by relaying traffic via the common node. There are, of course, practical problems with this—such as timing issues and fulfilling quality of service when a device is absent from the piconet—but the possibility is given in the specification. One limitation is that a device can only be the master in at most one of the piconets of which it is a member.

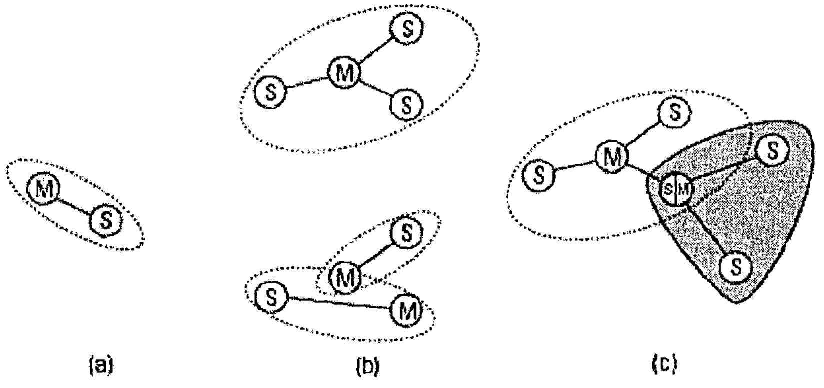


Figure 1.2.Three different piconet constellations: (a) two devices, (b) master relaying versus two separate piconets, and (c) interpiconet scheduling using time sharing.

1.1.5.2 Traffic types

Bluetooth wireless technology is designed to handle quite different types of traffic scenarios. Data may be sent without any QoS requirements (referred to as *best effort* traffic); thus, no bandwidth needs to be reserved and there are no requirements for latency or delay. Typically, file transfer and data synchronization fall into this category. Sometimes this traffic is called *asynchronous*. For real-time, two-way communication, the round-trip delay must be kept small, as do variations in the inter arrival time of data samples. If not, the quality will be perceived as unacceptable. This type of traffic is referred to as *synchronous*.

Typical examples are speech and video conversations. Streaming audio and video falls somewhere in between these categories. Small time variations between data samples are still important, but latency and roundtrip delays are of less interest. Such traffic is called *isochronous*. Bluetooth can handle all these traffic types—it is even possible to mix asynchronous and synchronous traffic between the master and a slave at the same time.

A synchronous link in Bluetooth is referred to as a *synchronous connection oriented* (SCO) link. It is a point-to-point link between the master and a slave where traffic is sent on slots reserved at regular intervals. Another logical link that carries traffic on reserved slots is called *enhanced synchronous connection-oriented* (eSCO) link. Both these logical links provide constant rate data services by carrying fixed-sized packets on reserved slots over the physical channel. The eSCO link (introduced in Bluetooth version 1.2) is more flexible than the SCO link in that it offers more freedom in choosing bit rates and it is more reliable, as a limited number of retransmissions can take place in between the reserved time slots.

The *asynchronous connection-oriented (logical transport)* (ACL) link is a point-to-multipoint link between the master and all the slaves on the piconet.
No reserved slots are used. The master can address an arbitrary slave at any slot not reserved for SCO/eSCO traffic, even one that has a SCO/eSCO logical link running with the master.

1.1.5.3 Packet structure

A baseband packet consists of an *access code*, a *packet header*, and the *payload*. The access code, which comes first in each packet, is used to trigger and synchronize the receiver. Each piconet uses a unique access code derived from the *BD_ADDR* of the master. Thus, by inspecting the access code, a receiver can determine if a packet is for another piconet. In that case, processing the rest of the packet can be aborted, which will help it save some power. Moreover, as the access code defines where a slot boundary is, it is used to time-synchronize the slave to the master clock.

This is necessary, as time drift is inevitable between different devices due to differences in their respective crystal frequencies. Consequently, each slave of a piconet must continuously adjust its clock offset relative to the master clock; otherwise it will eventually lose connection with the master.

The packet header is used to address individual slaves of a piconet. For this purpose, a 3-bit field denoted by *logical transport address* (*LT_ADDR*) is used. The master assigns nonzero addresses to slaves at connection setup, while the all-zero address is reserved for broadcast messages. Apart from this, the packet header conveys information regarding the type of data traffic, flow control, and the retransmission scheme. To increase the robustness of the packet header, it is encoded with a rate *R* ▯ 1/3 repetition code (i.e., each bit is repeated three times).

User data is carried by the payload. The length of this field can vary depending on the type of traffic—from zero bytes (for acknowledgment of received data when nothing needs to be sent in the reverse direction) up to 339 bytes (plus 4 bytes of payload header and CRC). The packet format is depicted in Figure 1.3.

A baseband packet may occupy up to 1, 3, or 5 slots, depending on its type. This allows for having asymmetric data rates in the forward and reverse directions without the overhead penalty that one-size packets would cause. Error detection may be applied through a 16-bit CRC code. Furthermore, it is possible to apply an error correcting code to the payload—either a rate *R* ▯ 1/3 repetition code, or a (15, 10) shortened Hamming code [3] (which has rate *R* ▯ 2/3)—when link conditions are bad. In the Bluetooth specification, one uses the notion *forward error correction* (FEC) for this.

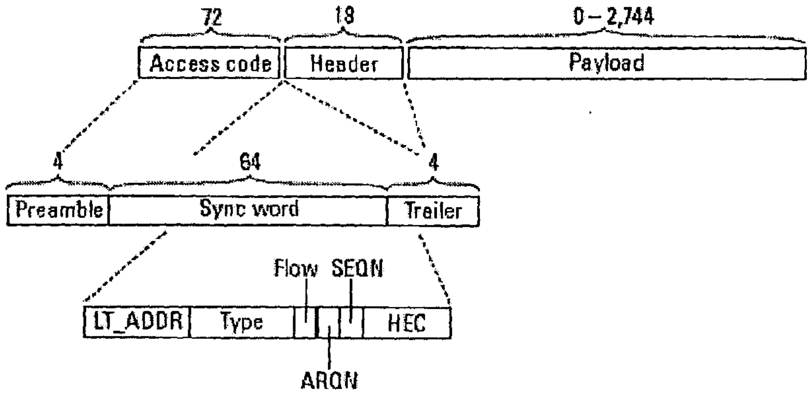


Figure 1.3 Packet format used in Bluetooth..

Best effort traffic (i.e., ACL links) without an error correcting code are carried over packets denoted by DH1, DH3, and DH5, where D indicates data, H stands for high rate, and the number is the maximum number of slots occupied by the packet. Similarly, there are DM1, DM3, and DM5 packets (where M stands for medium rate) for packets utilizing the shortened Hamming code. Using these packet types, it is possible to have user data rates ranging from 108.8 Kbps (symmetric, DM1) to 723.2 Kbps (forward) and 57.6 Kbps (reverse) for DH5 packets.

The achievable data rates using ACL packets are summarized in Table 1.1. For synchronous traffic, there are the HV1, HV2, and HV3 [where H stands for high-quality (referring to the relatively high bit rate available for speech coding) and V stands for voice] packets of 10, 20, and 30 information bytes, respectively. These one-slot packets have no CRC applied to the payload and are typically used to carry voice traffic. The achievable rate for all HV packets is 64 Kbps. The HV1 packet is protected by the rate *R* ▯. 1/3 repetition code, the HV2 packet is protected by the rate *R* ▯. 2/3 Hamming code, and the HV3 packet has no error correcting code applied.

Table 1.1
Summary of ACL Packets and Their Achievable Data Rates (in Kbps)

					Asymmetric Max. Rate	
Type	**Payload (Information Bytes)**	**FEC**	**CRC**	**Symmetric Max. Rate**	**Forward**	**Reverse**
DM1	0–17	2/3	Yes	108.8	108.8	108.8
DH1	0–27	No	Yes	172.8	172.8	172.8
DM3	0–121	2/3	Yes	258.1	387.2	54.4
DH3	0–183	No	Yes	390.4	585.6	86.4
DM5	0–224	2/3	Yes	286.7	477.8	36.3
DH5	0–339	No	Yes	433.9	723.2	57.6
AUX1	0–29	No	No	185.6	185.6	185.6

There is also a DV packet which consists of two parts—one carrying 10 bytes of voice data (no CRC) and one carrying asynchronous user data (0 to 9 bytes) for which CRC is applied. The voice part also offers 64 Kbps. In addition to these, the eSCO logical transport is mapped on EV3, EV4, and EV5 packets. All these have a CRC, which implies that retransmission is possible if no acknowledgment has been received within the retransmission window. The EV4 also applies the error correcting code to the payload. For these packets, the achievable rates are 96, 192, and 288 Kbps, respectively. The rates that are supported for synchronous traffic are summarized in Table 1.2.

1.1.6 Link manager protocol

It is the link manager that is responsible for the control of the Bluetooth link. That includes all tasks related to the setup, detachment, or configuration of a link. The

LM is also responsible for exchanging security-related messages. The LMs in different units exchange control messages using the LMP. A large set of control messages or LMP *protocol data units* (PDU) have been defined. Many of these are security related and some PDUs are used to carry the information needed at pairing and authentication, and for enabling of encryption.

Table 1.2
Summary of Synchronous Packets and Their Achievable Data Rates (in Kbps)

Type	Payload (Information Bytes)	FEC	CRC	Symmetric Max. Rate
HV1	10	1/3	No	64
HV2	20	2/3	No	64
HV3	30	No	No	64
DV	10 + (0–9)*	2/3*	Yes*	64 + 57.6*
EV3	1–30	No	Yes	96
EV4	1–120	2/3	Yes	192
EV5	1–180	No	Yes	288

*Marked items of the DV packet are only relevant to the data part of the payload.

The LMP PDUs are transferred in the payload instead of ordinary data. They are always sent as single-slot packets and they can be carried in two different types of data packets. In order to distinguish LMP packets from other packets, a special type code is used in the packet header of all LMP messages. To avoid overflow in the receiving packet buffer, flow control is normally applied to the asynchronous data packet in Bluetooth. However, no flow control applies to LMP PDUs. The LMP PDU payload format is shown in Figure 1.4. The PDU format can be considered as one byte header followed by the LM data.

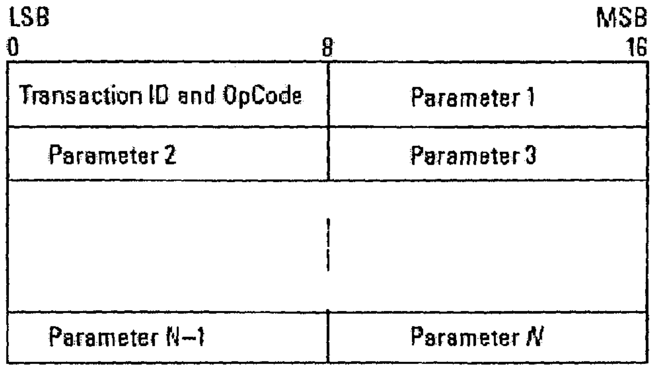


Figure 1.4 The LMP PDU format.

The header has two fields. The first field is only 1 bit long and contains the transaction *identifier* (ID). The second field is 7 bits long and contains the *operation* *code* (OpCode). The operation code tells which type of LMP PDU that is being sent. Each LMP message has its unique OpCode.

As we have described, the LMP is used to control and set up the link. A typical PDU flow example at connection creation is shown in Figure 1.5. The connection establishment always starts with the master unit paging the slave unit. After the basic baseband page and page response messages have been exchanged, the setup of the link can start. Before the master sends a connection request, it might request information from the slave regarding its clock, version of the link manager protocol, LMP features, and the name of the slave units. A set of LMP PDUs has been defined for this purpose. The connection setup procedure then really starts with the master sending the LMP connection request message.

Next, the security-related message exchange takes place. Finally, the peers complete the connection setup by exchanging LMP setup complete messages.
Special security related PDUs have been defined in order to accomplish:

- Pairing;
- Authentication;
- Encryption;
- Changing the link key.

The details of principles and usage are described in Chapters 2 and 3. In addition to the different LM functions we have mentioned previously, the LM is also responsible for performing role change (master-slave switch), controlling multislot packet size, and power control.

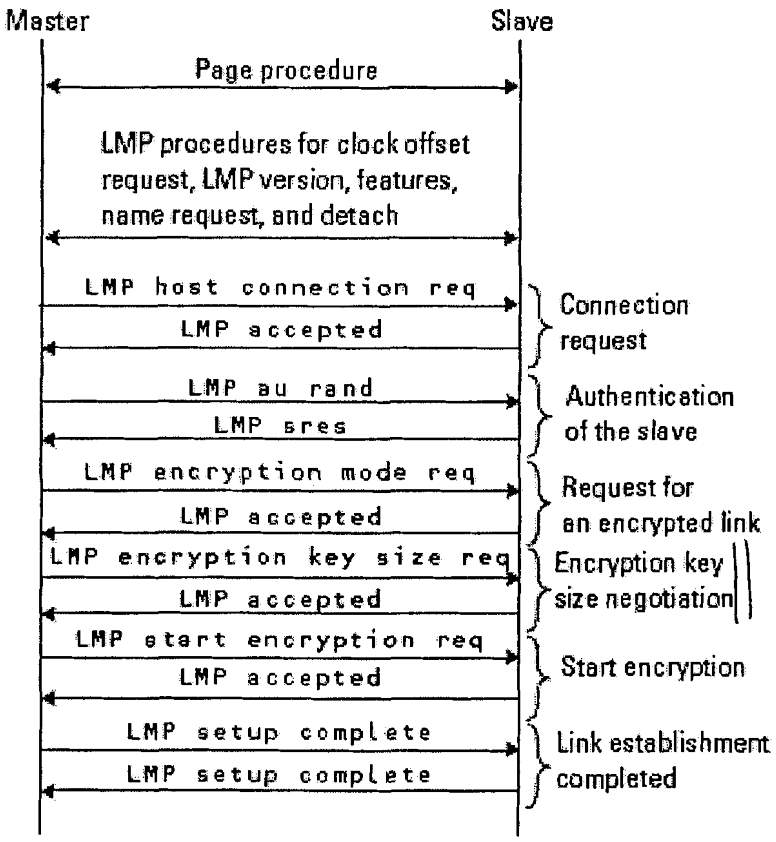


Figure 1.5 Connection establishment examples, LMP PDU flow.

1.1.7 Logical link control and adaptation protocol

The L2CAP takes care of datagram segmentation and reassembly, multiplexing of service streams, and quality-of-service issues. The L2CAP constitutes a filter between the Bluetooth independent higher layers running on the host and the lower layers belonging to the Bluetooth module. For instance, *transmission control protocol/internet protocol* (TCP/IP) traffic packets are too large to fit within a baseband packet. Therefore, such packets will be cut into smaller chunks of data before they are sent to the baseband for further processing. On the receiving side, the process is reversed; baseband packets are reassembled into larger entities before being released to higher layers.

1.1.8 Host control interface

The HCI is a common standardized interface between the upper and lower layers in the Bluetooth communication stack. As we described in Section 1.1.3, the HCI provides the capability of separating the radio hardware-related functions from higher layer protocols, which might run on a separate host processor. By using the HCI, it is possible to use one Bluetooth module for several different hosts and applications. Similar, upper-layer applications implemented in one host can use any Bluetooth module supporting the HCI.

Figure 1.6 provides an overview of the lower Bluetooth layers and the HCI interface. The HCI commands for the Bluetooth module are handled by the HCI firmware that accesses the baseband and link manager. Not all Bluetooth implementations run the lower and higher layer processing on different processors. Integrated implementations are also possible. Consequently, the HCI is an optional feature and only products that benefit from the separation use it.

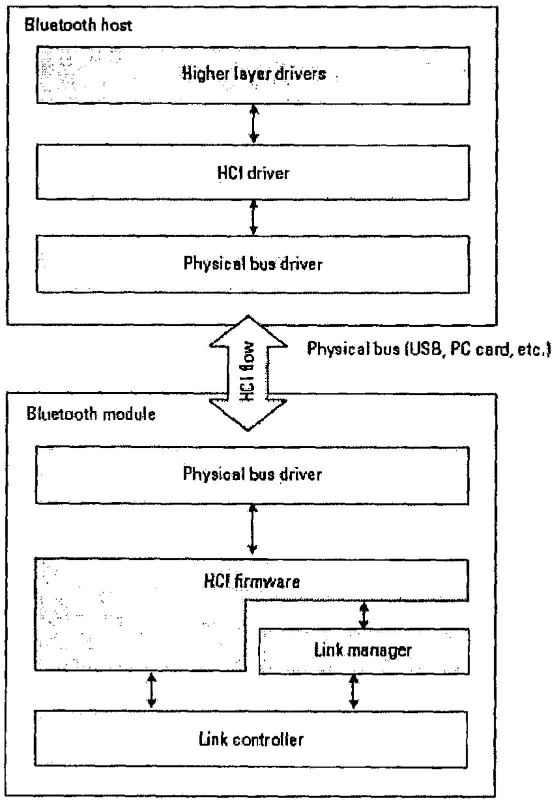


Figure 1.6 Overview of the lower software layers and the position of the HCI stack.

The HCI commands are transported between the Bluetooth module and host by some physical bus. This can, for example, be a *universal serial bus* (USB) or PC card connection. Three physical transport media have been defined [4]: USB, RS232, and universal asynchronous receiver/transmitter (UART). The host exchanges data with the module by using *command packets,* and the module gives responses to these requests or sends its own commands to the hosts, which are called *event packets*. Data to be passed over a Bluetooth link is transported in *data packets*.

To prevent buffer overflow in the host controller, flow control is used in the direction from the host to the host controller. The host keeps track of the size of the buffer all the time. At initialization, the host issues the Read Buffer Size command. The host controller then continuously informs the host of the number of completed transmitted packets through the Number of Completed Packet event.

The command packets can be divided into six different subgroups:

1. Link control commands.
2. Link policy commands.
3. Host controller and baseband commands.
4. Read information commands.
5. Read status commands.
6. Test commands.

The link control commands are used to control the link layer connections to other Bluetooth devices. Control of authentication and encryption as well as keys and pass-key commands belong to this subgroup. The policy commands are used to control how the link manager manages the piconet. The host controller and baseband commands are used to read and write into several different host controller registers. This includes reading and writing keys and pass-keys to or from the host controller, as well as reading and writing the general link manager authentication and encryption policy (see Section 2.5).

The read information commands are used to get information about the Bluetooth device and the capabilities of the host controller. Information on connection states and signal strength can be obtained through the read status commands. Finally, the test commands are used to test various functionalities of the Bluetooth hardware.

1.1.9 Profiles

The Bluetooth standard is not limited to specific use cases or applications. However, in order to offer interoperability and to provide support for specific applications, the Bluetooth SIG has developed a set of so-called *profiles*. A profile defines an unambiguous description of the communication interface between two units for one particular service. Both basic profiles that define fundamental procedures for Bluetooth connections and profiles for distinct services have been defined.

A new profile can be built on existing ones, allowing efficient reuse of existing protocols and procedures. This gives raise to a hierarchical profiles structure as outlined in Figure 1.7. The most fundamental definitions, recommendations, and requirements related to modes of operation and connection and channel setup are given in the *generic access profile* (GAP). All other existing Bluetooth profiles make use of the GAP.

The very original purpose of the Bluetooth standard was short-range cable replacement. Pure cable replacement through RS232 emulation is offered by the *serial port profile*. Several other profiles, like the *personal area network* (PAN) and *local positioning profile* make use of the *serial port profile*. One level deeper in the profiles hierarchy is the *general object exchange profile*. The purpose of this profile is to describe how the IrDA *object exchange* (OBEX) layer is used within Bluetooth. OBEX can be used to any higher layer object exchange, such as synchronization, file transfer, and push services.

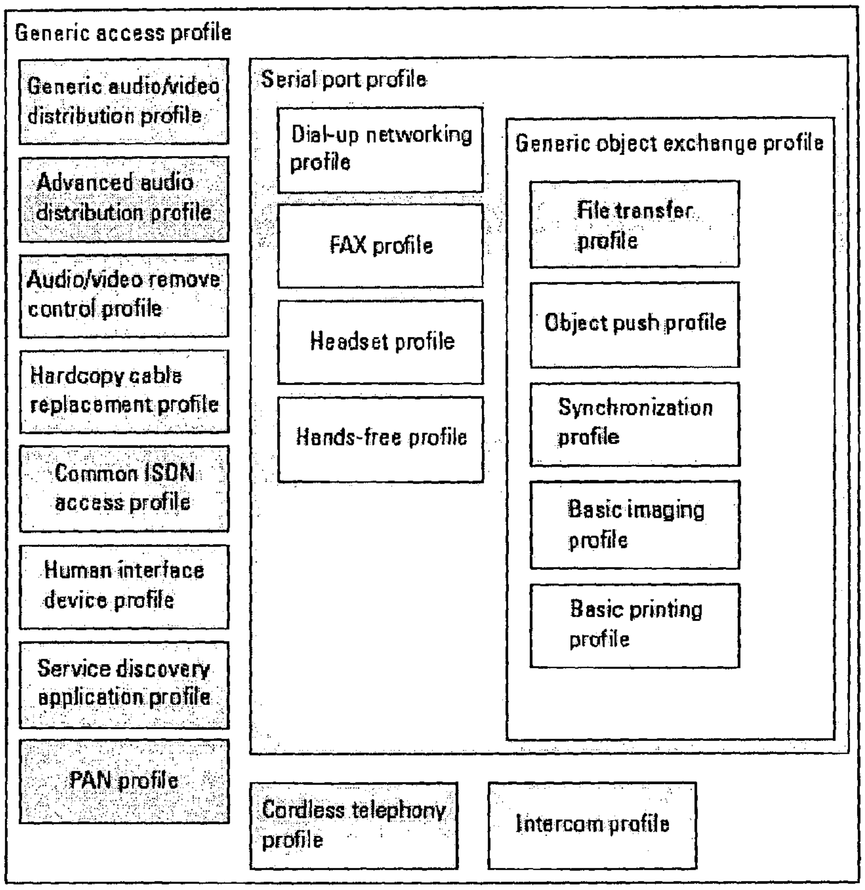

Generic access profile
Generic audio/video distribution profile
Advanced audio distribution profile
Audio/video remove control profile
Hardcopy cable replacement profile
Common ISDN access profile
Human interface device profile
Service discovery application profile
PAN profile
Serial port profile
Dial-up networking profile
FAX profile
Headset profile
Hands-free profile
Generic object exchange profile
File transfer profile
Object push profile
Synchronization profile
Basic imaging profile
Basic printing profile
Cordless telephony profile
Intercom profile

Figure 1.7 Bluetooth profiles.

Different services have different security requirements. In Section 10 we discuss the security requirements and solutions for a selection of Bluetooth profiles.

Most profiles benefit from using the baseband security functions. It is important, though, that the mechanisms are correctly understood and that application providers are aware of the strength as well as limitations of the link level security services. New profiles are constantly being developed, and some existing profiles may become replaced as others covering the same or similar functionality are added. Profiles are released independently of the core specification release schedule. In Figure 1.7 we have included the profiles that were adopted at the time of (November 2003).

1.2 Bluetooth security basics

Security issues surfaced from the beginning in the design of the Bluetooth system. It was decided that even for the simplest usage scenarios, the Bluetooth system should provide security features. To find the correct level of security when a new communication technology is defined is a nontrivial task, as it depends on usage. Bluetooth is versatile, which further increases the difficulties in finding the correct level one anticipates for the system. We start this section by discussing some typical user scenarios for Bluetooth applications.

1.2.1 User scenarios

In Section 1.1.9 we touched upon Bluetooth profiles. The overview of the profiles shows that the technology can be used in a large number of different applications. The overview also demonstrates that very different devices with very different capabilities might utilize the local connectivity provided by Bluetooth. However, most applications are characterized by two things: *personal area usage* and *ad hoc connectivity*. The Bluetooth link level security mechanisms have been designed with these two characteristics in mind, and below we describe what we mean by personal area networks and ad hoc connectivity.

1.2.1.1 Personal area networks

The personal area network concept is a vision shared among a large number of researchers and wireless technology drivers. A PAN consists of a limited number of units that have the ability to form networks and exchange information. The units can be under one user's control (i.e., personal computing units) or they can be controlled by different users or organizations. Bluetooth is used as a local connection interface between different personal units, such as mobile phones, laptops, *personal digital assistants* (PDA), printers, keyboards, mouses, headsets, and loudspeakers. Hence, Bluetooth is a true enabling technology for the PAN vision. The devices are typically (but not at all limited to) consumer devices.

Different consumer devices have different manufacturers, and the personal usage of a device will vary from person to person. Hence, in order to provide interoperability between the different personal devices, the security must to some extent be configured by the user. Bluetooth security solutions have been designed with the principles in mind that any ordinary user should be able to configure and manage the necessary security actions needed to protect the communication links.

The information exchanged over Bluetooth might very well be sensitive and vulnerable to eavesdropping. In addition, users of mobile phones or laptops would like to be sure that no unauthorized (by the users) person is able to connect to their personal devices. Another issue is location privacy. People would like to use their Bluetooth devices anywhere they go without fearing that somebody can track their movements. To ensure that, device anonymity is an important user expectation.

To sum up, there are four fundamental security expectations for Bluetooth:

1. Easy-to-use and self-explanatory security configuration.
2. Confidentiality protection.
3. Authentication of connecting devices.
4. Anonymity.

Bluetooth provides link encryption and authentication. If the expectation for easy-to-use and self-explanatory security configuration has also been fulfilled is hard to say—at least the system has been designed with this goal in mind.

1.2.1.2 Ad hoc connectivity

As discussed previously, Bluetooth has been designed to support the wireless PAN vision. Sometimes the relations between the devices are fixed, like the connection between a desktop computer and the keyboard or the mouse. Another example is the connection between a mobile phone and a headset. However, sometimes one wishes to set up connections on the fly with another device that just happens to be nearby. This is ad hoc connectivity. To illustrate an ad hoc connectivity scenario, we give an example. Let us consider a business meeting where two persons, an employee and a visitor, meet in a room equipped with a video projector, illustrated in Figure 1.8.

The two persons in the room are each carrying one laptop. The laptops contain presentation information that the users would like to present to each other using the video projector. Furthermore, after the presentation, the visitor would like to send a presentation to the employee. We assume that the video projector and the laptops support Bluetooth for local connectivity. Hence, we have a PAN scenario with three different Bluetooth-enabled devices:

1. A video projector.
2. A visitor laptop.
3. An employee laptop.

The ad hoc nature of these connections stems from the fact that no prior relation can be assumed between the visitor's laptop and the projector or between the visitor and employee laptop. Hence, in order to provide security (authentication and encryption) on the communication links, the security relations must be set up on the fly and often by the users themselves. The original Bluetooth pairing mechanism provides the possibility of setting up ad hoc security relations. However, one would like to minimize the load on the user and find alternative methods to manual procedures. In this book we revisit these issues several times and discuss features needed to make ad hoc connectivity as secure and, at the same time, as user friendly as possible. In the next chapter we will give an overview of the Bluetooth security architecture. But first we review some frequently used notions and terminology.

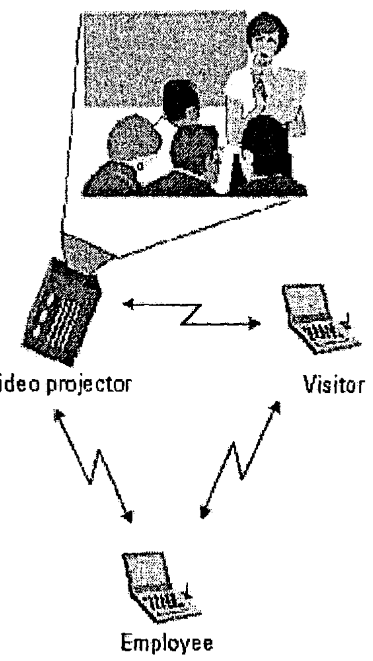


Figure 1.8 Bluetooth meeting room ad hoc connectivity scenario.

1.2.2 Notions and terminology

We already mentioned that security expectations for Bluetooth are related to the following four aspects (1) easy-to-use and self-explanatory security configuration, (2) confidentiality protection, (3) authentication of connecting devices, and (4) anonymity. These aspects describe what we mean by security in this book. When considering general information systems, security is understood to encompass the following three aspects [5]: confidentiality, integrity, and availability.

The mechanisms that address the confidentiality aspects should provide the means to keep user information private. Integrity mechanisms address the capability to protect the data against unauthorized alterations or removal. Finally, availability deals with the aspect that the system should be available as expected. Availability is therefore closely related to reliability and robustness.

Comparing this with what we said within the context of Bluetooth, we see that the aspects of confidentiality and availability appear in the four security expectations, although it may be argued that anonymity is an aspect on its own. The Bluetooth standard does not currently include any data integrity protection mechanism. In the sections that follow, we discuss first the meaning of confidentiality and integrity in more detail. We then continue to give a very compact description of cryptographic mechanisms that are used to achieve security.

1.2.2.1 Confidentiality

Confidentiality of data can be achieved by transforming the original data, often called the *plaintext*, into a new text, the *ciphertext*, that does not reveal the content of the plaintext. The transformation should be (conditionally) reversible, allowing the recovery of the plaintext from the ciphertext. To avoid that the transformation itself has to be kept secret to prevent a recovery of the plaintext, the transformation is realized as a parameterized transformation, where only the controlling parameter is kept secret. The controlling parameter is called the *key* and the transformation is called *encryption*.

A good encryption mechanism has the property that unless the key value is known, it is practically infeasible to recover the plaintext or the key value from the ciphertext. What actually "practically infeasible" means is not exactly defined. Moreover, what is infeasible today may be feasible tomorrow. A good measure of the quality of an encryption mechanism is that even if very many plaintext and corresponding ciphertext messages are known, the amount of work to break a cipher (e.g., recover the key) is in the same order as the number of key combinations. In other words, breaking the cipher is equivalent to a complete search through the key space.

1.2.2.2 Integrity

The second aspect of security, that is, integrity, is about ensuring that data has not been replaced or modified without authorization during transport or storage. Integrity should not be confused with peer authentication or identification (see the explanation below), which can be used to verify the communication peer during connection setup. Peer authentication only guarantees that a connection is established with the supposed peer, while message integrity is about authenticity of the transmitted messages. Integrity protection of transmitted data is not part of the Bluetooth standard.

1.2.2.3 Symmetric and asymmetric mechanisms

Cryptographic mechanisms are distinguished as being either *symmetric key* or *asymmetric key*. Symmetric mechanisms are mechanisms for which the communicating parties share the same secret key. There is, so to speak, a symmetric situation among the parties. If the mechanism concerns the encryption of files, say, then the receiver is not only able to decrypt the files received from the transmitter, but in fact the receiver is able to decrypt encrypted files that were generated by the receiver itself. Thus, a receiver cannot claim that the decrypted data indeed was sent by the sender.

Symmetric mechanisms (we sometimes also use the word *schemes*) are also called *secret-key mechanisms*. An important property of symmetric mechanisms is that the transportation of the key from the sending to the receiving party needs to be realized in such a way that no information about the key is leaked to outsiders. This need for key transfer constitutes the core problem in key management. Encryption of large data blocks is often realized through symmetric encryption mechanisms because they are faster than the asymmetric mechanisms. Secret-key mechanisms have a long history, and many variants are known and in use. The main two types of secret-key mechanisms are block and stream ciphers.

Asymmetric mechanisms are mechanisms that realize an encryption and decryption transformation pair for which the keys for the respective transformations are not the same. In fact, one demands that one of the keys cannot be recovered from the other. Hence, the keys at the sending and receiving sides have an asymmetry in their properties. Asymmetric mechanisms are also called *public-key mechanisms*. This naming stems from the fact that for asymmetric mechanism, one speaks about a private- and public-key pair. The private key is kept secret from everyone else and the public key is made accessible to everybody (i.e., it is made public). Asymmetric mechanisms solve some of the key distribution problems that arise in the activation of symmetric mechanisms. This advantage of asymmetric mechanisms is, however, often spoiled by the need to have proofs of the binding between a public key and an entity who claims to be the owner (of the private key). A widespread solution to this is the use of so-called *certificates*. Such certificates bind a public key to an identity and are issued by a common trusted agent.

Public-key schemes are asymmetric cryptographic mechanisms. The two keys that relate to a pair of encryption and decryption transformations are called the public key and private key, respectively. Together they form a public- and private-key pair. In public-key schemes, the private key cannot be recovered by practical means from the public key or any other publicly known information for that matter.

The best known public-key schemes are the Rivest, Shamir, and Adleman (RSA) and Diffie-Hellman schemes. Both date back to the beginning of publickey cryptography in the 1970s. Diffie-Hellman is used for key establishment, while RSA is for key transport, encryption, or digital signatures. For more information and a historical overview, see [6].

1.2.2.4 Block and stream ciphers

Block ciphers are symmetric cryptographic mechanisms that transform a fixed amount of plaintext data (a block) to a block of ciphertext data using a key, and that have an inverse transformation using the same key (as used for the encryption transformation). See Figure 1.9(a). Block ciphers are very useful as building blocks to obtain other cryptographic mechanisms, such as authentication mechanisms. In Bluetooth, the SAFER block cipher is used in this manner. *Stream ciphers* are the other main type of symmetric cryptographic mechanisms. Here a stream (sequence) of plaintext symbols is transformed symbol by symbol in a sequence of ciphertext symbols by adding, symbol by symbol, a so-called *key stream* to the sequence of plaintext symbols. See Figure 1.9(b). Stream ciphers have a trivial inverse transformation. Just generate the same key stream and subtract its symbols from the stream of cipher symbols. Bluetooth uses the
*E*0 stream cipher to encrypt the data sent via the radio links.

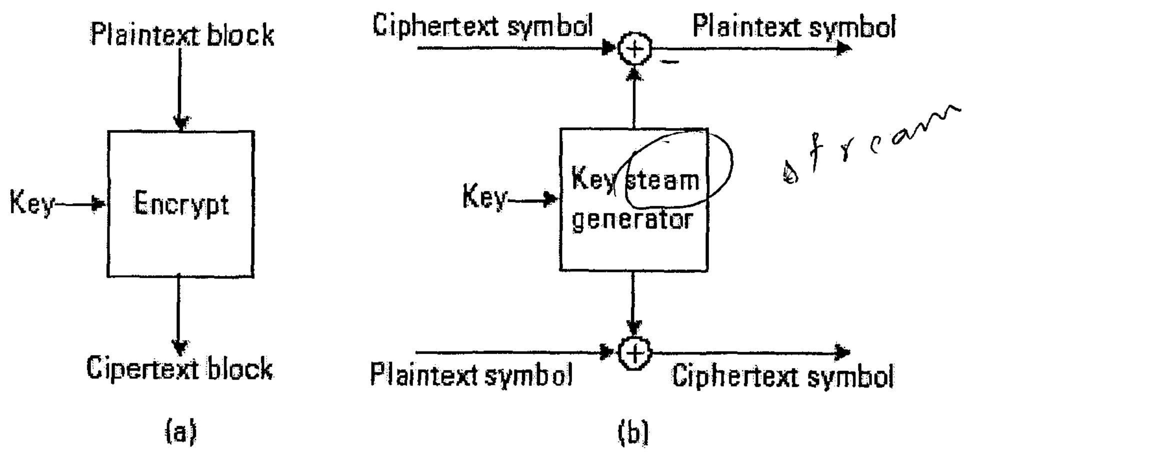


Figure 1.9 (a) Block cipher, and (b) stream cipher.

1.2.2.5 Authentication

Authentication is the procedure by which a unit (the verifier) can convince itself about the (correct) identity of another unit (the claimant) it is communicating with. Note that in cryptography, one often refers to this as the *identification*, and authentication is reserved for referring to (message or data) authenticity, that is, the problem of asserting that a received message is authentic (as sent by the sender). Here we use the definition of authentication that is in use in many (cellular) communication systems [e.g., Global Mobile System (GSM) and wideband code division multiple access (WCDMA)], that is, it refers to the process of verifying the consistency of the link keys in the involved Bluetooth devices exchanged during the pairing procedure.

1.2.2.6 Authorization

Authorization is the process of giving someone permission to do or have access to something. For Bluetooth this means to decide whether a remote device has the right to access a service on the local host and what privileges to gain for it. Usually this involves some form of user interaction. Alternatively, granting access to services can be subject to device-specific settings. Sometimes authorization refers both to administering system permission settings and the actual checking of the permission values when a device is getting access.

CHAPTER 2

Literature Survey

2. Literature Survey

The security demands in the various usage scenarios for Bluetooth differ substantially. For example, a remote-controlled toy and a remote-controlled industrial robot constitute usage cases with essentially different demands on security. The security architecture for Bluetooth is designed to provide built-in security features even for the simplest cases and at the same time provide adequate support to provide security in demanding cases, such as those where Bluetooth devices are used in a network environment.

This chapter gives an overview of the Bluetooth security literature survey, starting with a description of the different key types that are used, how the link encryption is organized, how all the basic features are controlled through security modes to achieve different trust relations and enhanced security suggestions for Bluetooth architecture.

2.1 Key types

The security provided by the Bluetooth core is built upon the use of symmetric key cryptographic mechanisms for authentication, link encryption, and key generation. A number of different key types are used in connection with these mechanisms. In Bluetooth, a link is a communication channel that is established between two Bluetooth devices. To check that a link is established between the correct devices, an authentication procedure between two devices has been introduced. The authentication mechanism in this procedure uses the so-called *link key*. As we will find out later, there are several different types of link keys.

Link keys are not only used for authentication. They are also used for derivation of the key that controls the encryption of the data sent via a link. Through this encryption, confidentiality of the transmitted data is realized. The corresponding encryption mechanism uses the *link encryption key*. Loosely speaking, a link key is used for the authentication between two devices and to derive the link encryption key. A link key is created during the pairing of two devices. Section 2.2 contains more details on the pairing and use of pass-keys.

Before we discuss the pairing mechanism, it is useful to clarify the conditions under which communication between two devices will occur. It is important to distinguish two important states. Firstly, we have the state in which a device wants to establish a connection with a device it has not been paired with.

Secondly, we have the state where a device wants to communicate with a device it has paired with. Of course, a device may, as a result of a malfunction or a forced reset, have lost the pairing information associated with a device. In such a situation, the device should fall back to the unpaired state.

The pairing operation will result in a link key that two devices will use for authentication and link encryption key generation directly after the pairing and at later instances. The Bluetooth system recognizes two types of link keys: *semi permanent* and *temporary keys*. Furthermore, two types of semi permanent (link) keys are distinguished: *unit keys* and *combination keys*. A unit key is a link key that one unit generates by itself and uses as a link key with any other (Bluetooth) device, and a combination key is a key that a device generates in cooperation (combination) with another device. Therefore, any unit key that a specific device has may be known to many other devices, whereas each combination key is only known to itself and the device with which it was generated. Unit keys can only be safely used when there is full trust among the devices that are paired with the same unit key. This is because every paired device can impersonate any other device holding the same unit key. Since Bluetooth version 1.2, the use of unit keys is not recommended. But, for legacy reasons, unit keys have not been completely removed from the specification. Besides the combination and unit keys, two other key types are used: *initialization keys* and *master keys*. These are temporary keys.

The initialization key is a short-lived key that exists during the pairing of two devices. The master key is a link key that the master generates prior to the setup of an encrypted broadcast communication to several slave devices. Besides the link keys, we have three ciphering keys: *the encryption key KC*, the *constrained encryption key* .*'K C* , and the *payload key KP*. The encryption key is the main key that controls the ciphering. Since this key may have a length (in bits) that exceeds legislative constraints on the maximally allowed key length, *KC* is not used directly but is replaced by the constrained encryption key.*'KC* , whose number of independent bits can be selected from 8, 16, . . . , 128 bits.

Currently there is little reason to accept key lengths less than 128 bits because the export regulations have been relaxed since the original design of the Bluetooth system. It is directly derived from *KC*. Finally, the payload key is a ciphering key derived from the constrained encryption key.*'KC*. This key is the initial state of the ciphering engine prior to generating the overlay sequence. A summary of the different key types can be found in Table 2.1. More details on the encryption keys are given in Section 2.4.1.

Table 2.1
Overview of Key Types

Purpose	**Semipermanent**		**Temporary**	
Authentication key generation	Unit key	Combination key	Initialization key	Master key
Ciphering			Encryption key Constrained encryption key	Payload key

2.2 Pairing and user interaction

As indicated earlier, the pairing of two devices is the procedure by which two devices establish a shared secret that they can use when they meet again. The pairing requires user interaction, for example, the entering of a pass-key. See Figure 2.1(a). The Bluetooth system allows the pass-key to be 128 bits long. Such a large pass-key value would be rather user unfriendly for manual input. However, this feature allows the use of a higher level automated key agreement scheme that can "feed" the agreed pass-key into the pairing procedure. See Figure 2.1(b).

The high-level key agreement scheme can be a network or *transport layer security* (TLS) protocol. Examples of such protocols are the Internet Engineering Task Force (IETF) protocols TLS [7] and Internet key exchange (IKE) [8].

There are two kinds of pass-keys in Bluetooth terminology: the *variable* pass-key and the *fixed* pass-key. The first type represents a pass-key that can be arbitrarily chosen at the pairing instance. This requires that some form of user interaction takes place in order to feed the Bluetooth device with the appropriate pass-key value. This interaction is most likely accomplished using a keyboard or numerical keypad. An example of a typical device with a variable pass-key is the mobile phone. In contrast, the fixed pass-key cannot be chosen arbitrarily when it is needed. Instead, a predetermined value must be used. This type of pass-key is used when there is no user interface to input a value to the Bluetooth device. Clearly, for a pairing to work, only one device can have a fixed pass-key (unless, of course, both devices happen to have the same fixed pass-key). Examples of devices in need of fixed pass-keys are Bluetooth-enabled mice and headsets. These gadgets come with a factory preset pass-key when delivered to the customer.

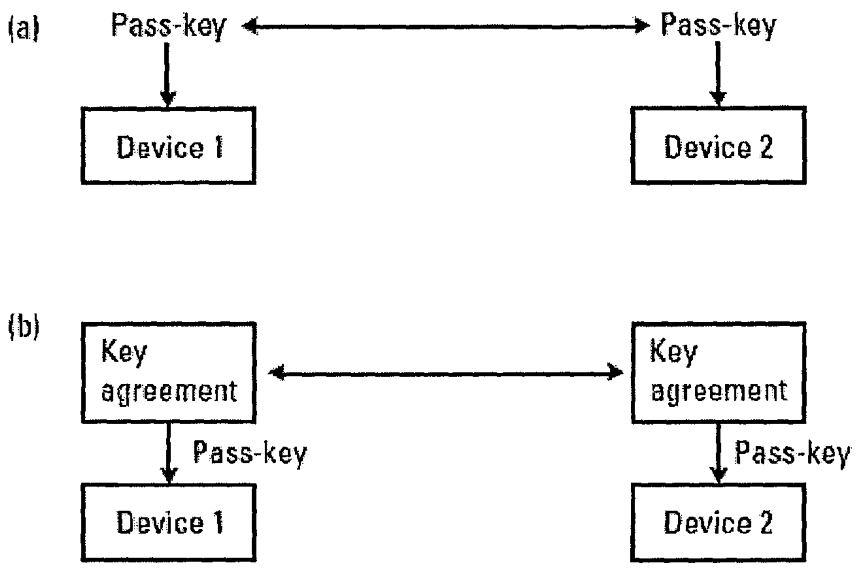


Figure 2.1 (a) Pairing through manual user interaction, and (b) pairing through separate key agreement protocol.

Note that a fixed pass-key need not be "fixed" in the sense that it can never be changed. Preferably, the user is allowed to change the fixed pass- key in some way. In some scenarios, a wired connection could be used, for example, by plugging in an external keyboard and changing the pass-key. This is only feasible if it is difficult for anyone but the rightful owner to have physical access to the Bluetooth device in question. More interesting is to allow the change over Bluetooth using an already paired device (equipped with the necessary user interface) over a *secure connection.* This implies that the user connects to the device with a fixed pass-key, authenticates itself, and requests the link to be encrypted before a fresh pass-key value can be sent to the remote device. The new value replaces the old one and becomes the fixed pass-key to use in subsequent pairings.

2.3 Authentication

A Bluetooth device in a connectable state accepts connection requests from other devices. This means that there is a risk that a connectable device is connected to and attacked by a malicious device. Obvious, this can be avoided by never entering a connectable state. On the other hand, that implies that no Bluetooth connections at all can be established. Accordingly, there is a need to securely identify the other communication peer so that connections from unknown devices can be refused. Device identification is provided through the Bluetooth authentication mechanism.

The authentication procedure is a so-called challenge-response scheme, where the *verifier* device sends a random challenge to the *claimant* device and expects a valid response value in return. The authentication procedure is only one way, and if mutual authentication is needed the procedure must be repeated with the verifier and claimant roles switched [12].

First, the verifier sends the claimant a random number to be authenticated. Then, both participants use the authentication function E1 with the random number, the claimants Bluetooth Device Address and the current link key to get a response. The claimant sends the response to the verifier, who then makes sure the responses match.

The used application indicates who is to be authenticated. So the verifier may not necessarily be the master. Some of the applications require only one way authentication, so that only one party is authenticated. This it not always the case, as there could be a mutual authentication, where both parties are authenticated in turn [17].

If the authentication fails, there is a period of time that must pass until a new attempt at authentication can be made. The period of time doubles for each subsequent failed attempt from the same address, until the maximum waiting time is reached. The waiting time decreases exponentially to a minimum when no failed authentication attempts are made during a time period [17].

2.4 Link privacy

Of all security aspects encountered in wireless scenarios, the easiest to understand is the one relating to confidentiality. Eavesdropping on a radio transmission can be accomplished without revealing anything to the victim. Radio waves are omni directional and travel through walls (at least to some extent). One can easily imagine hiding a small radio receiver close enough to intercept the messages sent by a user, without revealing its presence to anyone not knowing where to look for it. It may even be possible to do this without having physical access to the premises where the Bluetooth devices are used. If the walls surrounding the user area are not completely shielding the radio transmissions, eavesdropping can take place outside this room [12] [17].

Initially, Bluetooth was envisioned as a simple cable replacement technology. For some applications (such as device synchronization), replacing the wire with a radio has implications for confidentiality. It was desirable that the user should not experience any decrease in confidentiality when comparing the wireless with the wired solution. Thus, it was determined to look into what kind of security means were needed in order to give a sufficient degree of protection to Bluetooth communication.

In contrast to what sometimes has been claimed, the frequency hopping scheme used in Bluetooth gives no real protection against eavesdropping.

Firstly, there is no secret involved in generating the sequence of visited channels—it is determined by the master's LAP and native clock. Clearly, these two variables are not secret. Adversaries may have full knowledge of them by following the inquiry/page procedure traffic preceding the connection that they are now eavesdropping on. Alternatively, adversaries can simply connect to the master to automatically get all necessary information. Secondly, there are only 79 channels used. By running this many receivers in parallel (one for each channel) and recording all traffic, an offline attack seems feasible simply by overlaying all 79 recordings.

2.4.1 Protect the link

It is important to understand that Bluetooth specifies security for the link between radio units, not for the entire path from source to destination at the application layer. All protocols and profiles that need end-to-end protection will have to provide for these themselves. The implications are obvious in access point scenarios, where the remote application may be running on a unit located thousands of kilometers away, and traffic routing will involve many unknown links apart from the short radio link between the local unit and the access point. Since the user has no control over this, higher layer security is an understandable prerequisite to ensuring confidentiality all the way. However, even in the case when the source and destination reside on PDAs close to each other and there is only one direct Bluetooth link in between, one should remember that Bluetooth security only addresses the radio link. Who is really in control on the other side? Can malicious software access and control the Bluetooth radio [12].

2.4.2 Encryption algorithm

When it comes to the selection of which encryption algorithm to use, there are some considerations that need to be taken into account:

- Algorithmic complexity;
- Implementation complexity;
- Strength of the cipher.

Algorithmic complexity relates to the number of computations needed for encryption and decryption, while implementation complexity relates to the size of the implementation on silicon. These two items boil down to power consumption and cost—crucial properties for the battery-powered units Bluetooth is designed for. A complex algorithm will almost certainly require a larger footprint on silicon than does a simple algorithm, leading to higher cost. For the implementation, sometimes the speed obtained from dedicated hardware can be traded for flexibility and smaller size using a programmable component such as a *digital signal processor* (DSP) or a small *central processing unit* (CPU). For such solutions, an increased algorithmic complexity will inevitably demand higher clocking frequency, which also increases power consumption.

The last item on the list may be the most important. Should the ciphering algorithm prove to be vulnerable to some "simple" attack, the whole foundation of link privacy falls. Of course, the question of whether an attack is "simple" or not remains to be discussed, but, in general, even the smallest suspicion regarding strength is enough to cast doubts over the system's overall security quality [10] [17].

Do not confuse algorithmic complexity of encryption/decryption with the strength of the cipher. In fact, the goal is to keep the algorithmic complexity low while having the computational complexity for all types of attacks as high as possible.

Bluetooth deploys a stream cipher (see Section 1.2.2) with the desired properties of a small and simple hardware solution while being difficult to break.

A key stream is added modulo 2 to the information sequence. Thus, the scheme is symmetric, since the same key is used for encryption and decryption. This means the same hardware can be used for encryption and decryption, something that will actively keep down the size of the implementation. Moreover, stream ciphers are built efficiently using *linear feedback shift registers* (LFSR), which helps to reduce the die size even further.

The encryption algorithm uses four LFSRs of lengths 25, 31, 33 and 39, with the total length of 128. The initial 128-bit value of the four LFSRs is derived from the key stream generator itself using the encryption key, a 128-bit random number, the Bluetooth device address of the device and the 26-bit value of the master clock. The feedback polynomials used by the LFSRs are all primitive, with the Hamming weight of 5. The polynomials used are (25, 20, 12, 8, 0), (31, 24, 16, 12, 0), (33, 28, 24, 4, 0) and (39, 36, 28, 4, 0). Information on the fundamentals of LFSRs is found in [18].

The encryption/decryption consists of three identifiable parts: initialization of a payload key, generating the key stream bits, and, finally, the actual process of encrypting and decrypting the data. These functions are depicted in Figure 2.2.

The payload key is generated out of different input bits that are "randomized" by running the sequence generating circuitry of the key stream generator for a while.

Then the payload key is used as the starting state for the key stream generator in the encryption process. Since the sequence generating circuitry is used also for generating the payload key, the implementation is mainly concentrated in this part. The last part simply consists of XORing2 the key stream bits with the outgoing data stream (for encryption) or the demodulated received sequence (for decryption).

The choice of a stream cipher was to a large extent based on implementation considerations. Clearly, a key stream generator needs to fulfill a whole range of properties to make it useful for cryptographic purposes. For instance, the sequence must have a large period and a high linear complexity, and satisfy standard statistical and cryptographic tests [10] [11] [12] [17].

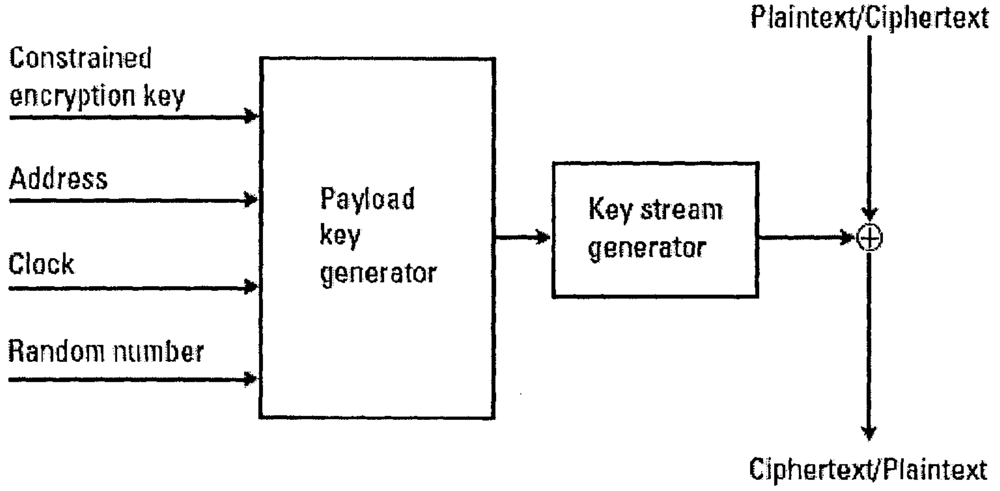


Figure 2.2 Stream cipher usage in Bluetooth.

As can be seen in Figure 2.2, there are some parameters involved in creating the payload key, *KP*. The secret constrained encryption key,.'*K C* , is generated by both units at the time a decision is made to switch encryption on. This key is fixed for the duration of the session or until a decision is made to use a temporary key (which will require a change of the encryption key). Even though the constrained encryption key always consists of 128 bits, its true entropy will vary between 8 and 128 bits (in steps of 8 bits), depending on the outcome of the link key negotiation that the involved units must perform before encryption can be started. The *address* refers to the 48-bit Bluetooth unit address of the master, while the *clock* is 26 bits from the master's native clock. Finally, there is a 128- bit *random number* that is changed every time the encryption key is changed.

This number is issued by the master before entering encryption mode and it is sent in plaintext over the air. The purpose of it is to introduce more variance into the generated payload key.

In Bluetooth, the key stream bits are generated by a method derived from the summation stream cipher generator in Massey and Rueppel [9]. This method is well investigated, and good estimates of its strength with respect to currently known methods for cryptanalysis exist. The summation generator is known to have some weaknesses that can be utilized in correlation attacks, but, thanks to the high resynchronization frequency (see Section 2.4.3) of the generator, these attacks will not be practical threats to Bluetooth.

2.4.3 Mode of operation

Not all bits of a Bluetooth packet are encrypted. The access code, consisting of a preamble, sync word, and a trailer, must be readable to all units in order for them to succeed in their receiver acquisition phase (i.e., in locking onto the radio signal). Furthermore, all units of a piconet must be able to read the packet header to see if the message is for them or not. Therefore, it is only the payload that is encrypted. The ciphering takes place after the CRC is added but before the optional error correcting code is applied. The principle is illustrated in Figure 2.3.
In generating the payload key, bits 1 to 26 of the master clock are used. This implies a change of the resulting key for every slot, since bit 1 toggles every 625 μs. However, the payload key is only generated at the start of a packet; multislot packets will not require a change of the payload key when passing a slot boundary within the packet. Consequently, for every Bluetooth baseband payload, the key stream generator will be initialized with a different starting state. This frequent change of the starting state is a key factor in its resistance to correlation attacks [12].

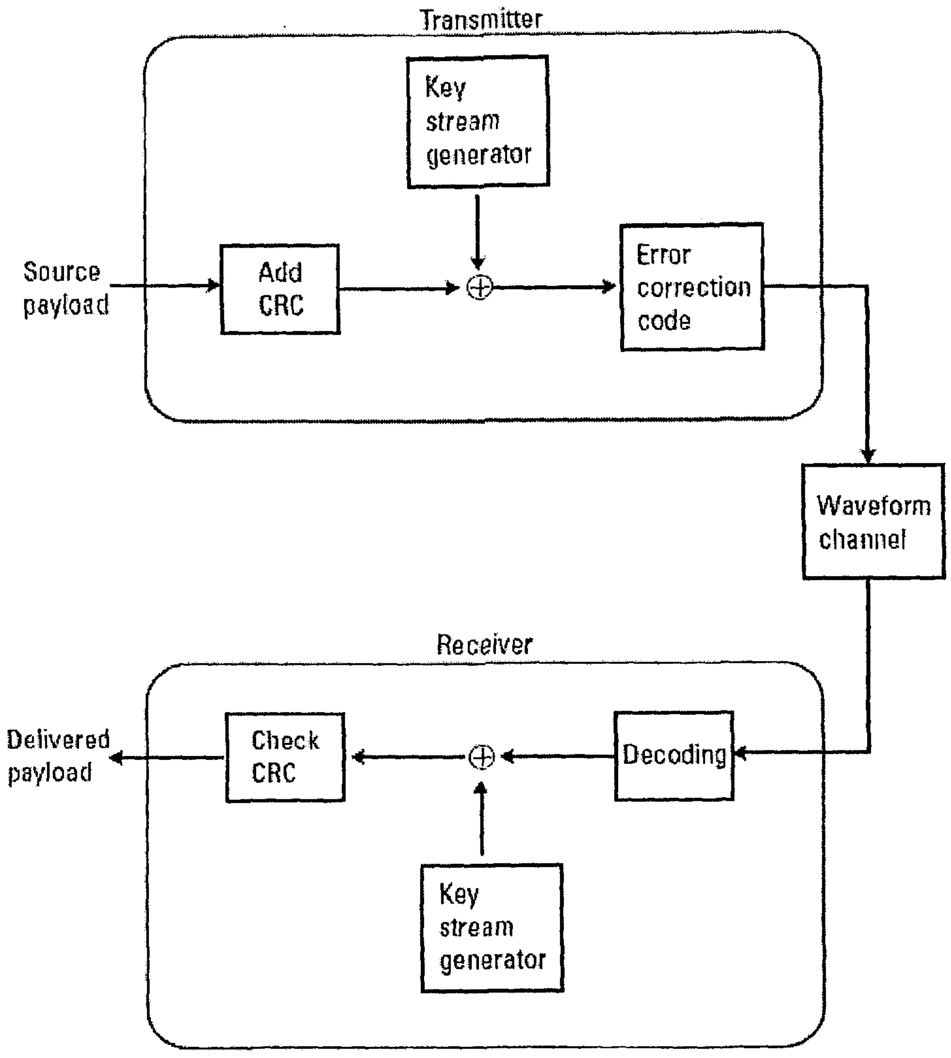


Figure 2.3 How to format encrypted packets.

The initialization phase takes some time. In principle, the input parameters are loaded into the shift registers of the key stream generator, which is then run to produce 200 output bits. Of these, the last 128 are retained and subsequently reloaded into the shift registers. These operations put a limit to how fast one can change from one payload key to another. Fortunately, Bluetooth specifies a guard space between the end of a payload and the start of the next of at least3 259 μs. The guard space is there in order to allow for the frequency synthesizer of the radio to stabilize at the next channel used before the start of the next packet. During this time (and, in principle, also during the 72. 54 μs of plaintext access code and packet header), the payload key initialization can be run without interfering with the encryption or decryption process. The principle is shown in Figure 2.4 [10] [12].

2.4.4 Unicast and broadcast

Broadcast encryption poses a slight problem due to the point-to-point paradigm used in Bluetooth. In principle, apart from itself, a slave device is only aware of the piconet master. Thus the slave has no security bonding to other slave members.

Specifically, each link in the piconet uses different encryption keys, since they are all based on their respective link keys. If the master would like to send an encrypted message to all its slaves, it can do these using individually addressed messages (also known as *unicast messages*) which will introduce unnecessary overhead. A better alternative is for the master to change all link keys to a temporary key, the *master key*. Based on this, all devices are able to generate a common encryption key that can be used in broadcast transmissions that address all slaves simultaneously.

One drawback with this approach is that mixing secure unicast traffic and secure broadcast traffic is not possible. The user must settle for one of these at a time. The reason is in the packet structure and required initialization time for the payload key. A broadcast message is identified from the all-zero *LT_ADDR*, while unicast messages have nonzero *LT_ADDR*. This 3-bit address field is part of the payload header. Not until this information has been received and interpreted can the receiver decide whether the payload key should be based on the encryption key used for unicast or broadcast traffic. By then, there is far too little time (less than 48 μs) to generate the payload key before the packet payload is being received unless very fast hardware (i.e., involving high clock frequency) is used. This, however, would put unrealistic requirements on the ciphering hardware and increase cost as well as power consumption. It is, of course, inappropriate to use the broadcast encryption key for unicast traffic also, since all devices within the piconet are able to decipher this.

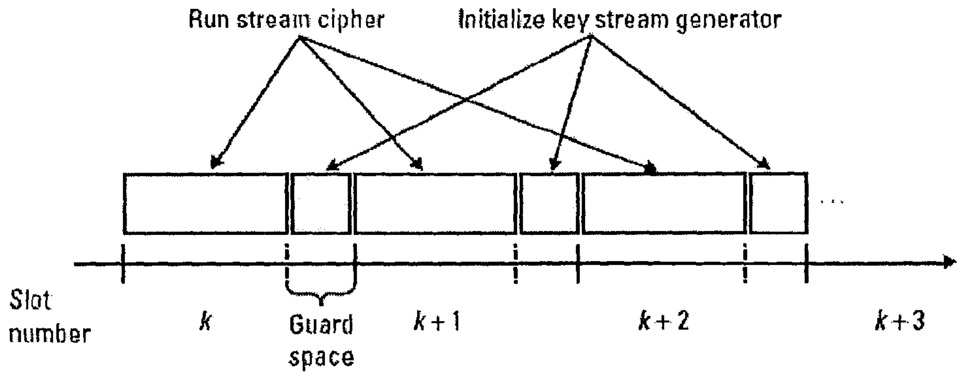


Figure 2.4 Operation of the encryption machinery.

TH-5195

2.5 Communication security policies

Security always comes at the prize of higher complexity. Hence, the security mechanisms should only be used when they are really needed. When and how to use the mechanisms, is determined by the security policies of a device. The Bluetooth standard provides some basic principles for enforcing link-level security and building more advanced security polices through the three defined security modes [11] [12].

One obvious choice for protecting Bluetooth communication is using the built-in link-level security mechanisms. Authentication and encryption is provided at baseband level. Using the built-in mechanisms has the advantage of protecting all layers above the link level (including control messages). The link level security mechanisms can be switched on or off. The security policy determines if a device demands authentication and/or encryption. One very simple approach is to demand maximum link-level security, that is, both authentication and encryption for all connections. This is an "always-on" link-level security policy. Such a simple policy has several advantages. First, the complexity is low. Furthermore, it gives a high level of security for all local connections and it is easy to implement. Finally, it is easy for the user to handle and understand the security policy. This kind of always-on policy and security enforcement is supported by Bluetooth security mode 3 (see Section 2.5.1). In order for this policy to be user convenient, the necessary keys must be present. If one can assume or actually demand that this is the case, the simple, always-on policy can be used and the security mechanisms are very easy to handle. Obviously, this policy also has some drawbacks:

- If the necessary link keys are not present, either a connection cannot be established or the keys need to be generated and exchanged at connection creation.
- If the necessary link keys are not present and the key exchange cannot be done automatically, the users must be involved and they must understand what is happening.

The latter implication can be a serious drawback, when the actual service does not demand any security. In this case, the user will be forced to handle a security procedure for a service that may need to be fast and convenient. Some device might only run services with high security requirements, and consequently this will not cause any problem. On the other hand, devices used at public places for information retrieval or exchange will certainly not have high security requirements for all its connections, and people using such services will probably not accept any tedious security procedures. Hence, a policy that demands link level security for some services and keeps some services totally "open" will be needed. In practice, this implies that a device will need a shared secret with some other device, and at the same time the device must be able to communicate with other devices without sharing any secrets and using link-level security.
In summary, the simple, always-on security policy is not sufficient for all Bluetooth usage scenarios. Better flexibility link-level security mechanism enforcement is necessary. This can be achieved by service level–enabled security (aligned with the access control mechanism). This is the motivation for the introduction of security mode 2 (see Section 2.5.1), which allows service level–enabled link layer security [10] [11] [12] [17].

2.5.1 Security modes

The GAP [16] defines the generic procedure related to the discovery of Bluetooth devices and the link management aspects of connecting to Bluetooth devices. The GAP also defines the different basic security procedures of a Bluetooth device. A connectable device can operate in three different *security modes*:

- *Security mode 1:* A Bluetooth unit in security mode 1 never initiates any security procedures; that is, it never demands authentication or encryption of the Bluetooth link.

- *Security mode 2:* When a Bluetooth unit is operating in security mode 2, it shall not initiate any security procedures, that is, demand authentication or encryption of the Bluetooth link, at link establishment. Instead, security is enforced at channel (L2CAP) or connection (e.g., Service Discovery Protocol (SDP), RFCOMM, and TCS) establishment.

- *Security mode 3:* When a Bluetooth unit is in security mode 3, it shall initiate security procedures before the link setup is completed. Two different security policies are possible: always demand authentication or always demand both authentication and encryption. In the following sections we discuss the different modes and how they are used in Bluetooth applications.

2.5.1.1 Security mode 1

Security mode 1 is the "unsecured" mode in Bluetooth. A unit that offers its service to all connecting devices operates in security mode 1. This implies that the unit does not demand authentication or encryption at connection establishment. For example, an access point that offers information services to anybody is a possible usage scenario for security mode 1[10].

Supporting authentication is mandatory and a unit in security mode 1 must respond to any authentication challenge. However, the unit will never send an authentication challenge itself and mutual authentication is never performed [17].

A unit in security mode 1 that does not support encryption will refuse any request for that. On the other hand, if encryption is supported, the unit should accept a request for switching encryption on [12].

2.5.1.2 Security mode 2

Security mode 2 has been defined in order to provide better flexibility in the use of Bluetooth link-level security. In security mode 2, no security procedures are initiated until a channel or connection request has been received. This means that it is up to the application or service to ask for security. Only when the application or service requires it will the authentication and/or encryption mechanisms be switched on. A sophisticated authentication and encryption policy based on the baseband mechanisms can be implemented using this principle [12] [17].

Security mechanisms enforcement and policy handling must be taken care of by the unit. One possibility is to use a "security manager" to handle this. In Section 2.5.2, we further discuss the role and implementation of a security manager. Security mode 2 comes at the price of higher implementation complexity and the risk of faulty security policies that might compromise the security of the unit [10].

2.5.1.3 Security mode 3

In security mode 3, on the other hand, security procedures (authentication and/or encryption) are enforced at connection establishment. This is a simple, always-on security policy. The implementation is easy and that reduces the risks of any security implementation mistakes. The drawback is the lack of flexibility. The unit will not be generally accessible. All connecting units need to be authenticated [10] [12] [17].

The difference between Security Mode 2 and Security Mode 3 is that in Security Mode 3 the Bluetooth device initiates security procedures before the channel is established [11]

There are also different security levels for devices and services. For devices, there are 2 levels, "trusted device" and "un trusted device". The trusted device obviously has unrestricted access to all services. For services, 3 security levels are defined: services that require authorization and authentication, services that require authentication only and services that are open to all devices [11] [17].

2.5.1.4 Security modes and security mechanisms

The different security modes define how a unit will act at connection establishment. Independent of the current security mode, a unit shall respond to security requests in accordance with what is specified in the link manager protocol (see Section 1.1.6). Hence, a security mode only defines the security behavior of the unit, but the security level for a connection is determined by the security modes of both units. Let one of two units be in security mode 3 and consequently demand encryption. Then the connection will be encrypted if both units support encryption; otherwise the connection will be terminated.

Table 2.2 describes the different security mode options and the resulting security mechanisms, while in Figure 2.5 the channel establishment procedure for different security modes is illustrated. In the figure, the connection and service establishment procedure for a Bluetooth device is shown as a flow diagram.

The process starts with the device that is in connectable mode. If the device is in security mode 3, it will try to authenticate and optionally encrypt the link directly after the link manager receives or makes a connection request. Specific host settings for access can be applied. For instance, devices that are not previously paired may be rejected. A device that is in security mode 1 or 2, on the other hand, will continue with the link setup procedure without any authentication or encryption request. Instead, the device in security mode 2 makes an access control check after a service connection has been requested. Access is only granted for authorized devices. Authorization is either given explicitly by the user or it can be given automatically (trusted and already paired device). For security mode 2, optional encryption can be requested before the connection to the service is finally established [12].

Service level access control can also be implemented by using security mode 3. Then authentication always takes place before the service request. Hence, security mode 2 gives better flexibility, since no security is enforced at channel or connection request. Thus it is possible to allow access to some services without any authentication or encryption and a unit can be totally open to some services while still restricting access to other services.

Table 2.2
The Different Security Mode Options for Master Respective Slave and Resulting Security Mechanism(s)

Slave Security Mode	**Master Security Mode**		
	1	2	3
1	No authentication, no encryption.	If the master application demands authentication (and encryption), then the link will be authenticated (and encrypted).	The link will be authenticated. If the master policy demands it, the link will be encrypted.
2	If the slave application demands it, the link will be authenticated (and encrypted).	If the master or slave application demands it, the link will be authenticated (and encrypted).	The link will be authenticated. If the master policy demands it, or if the slave application demands it, the link will be encrypted.
3	The link will be authenticated. If the slave policy demands it, the link will be encrypted.	The link will be authenticated. If the slave policy demands it, or the master application demands it, the link will be encrypted.	The link will be authenticated. If the slave or the master policy demands it, the link will be encrypted.

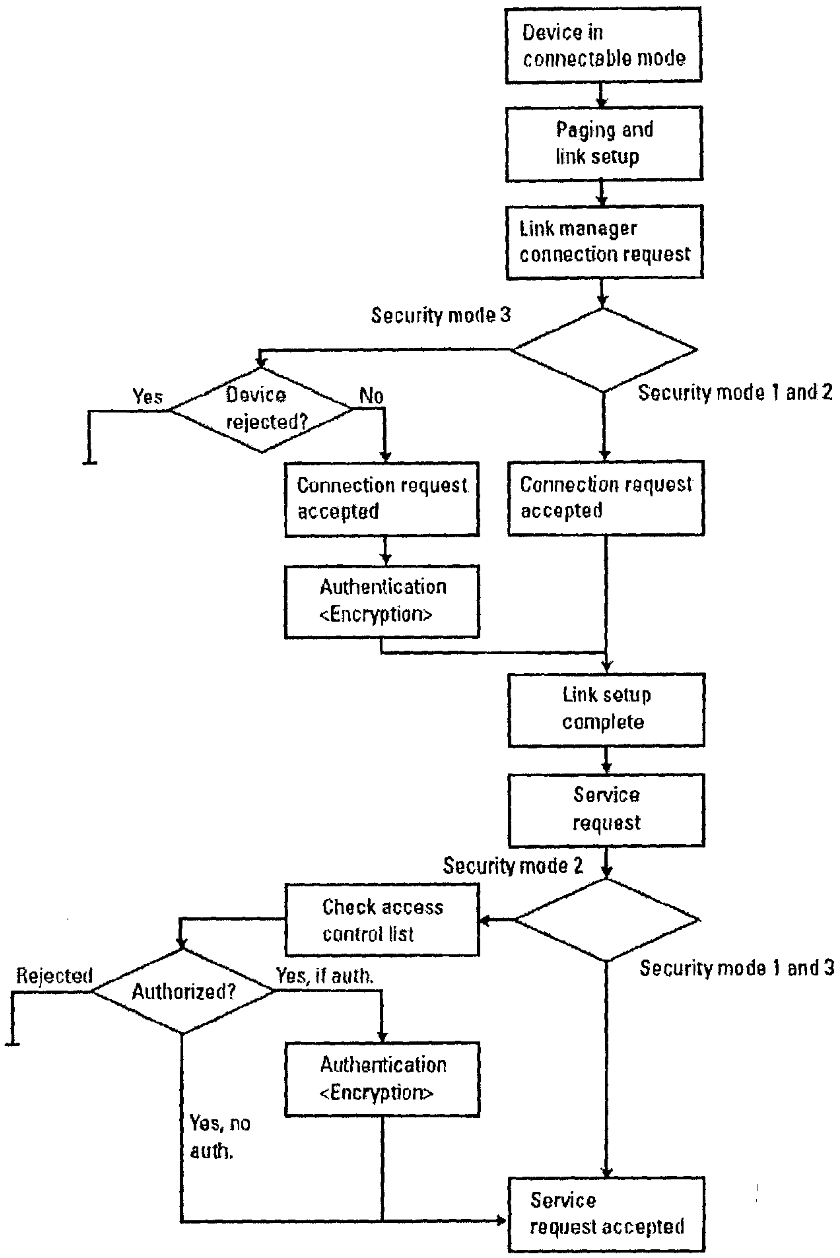


Figure 2.5 Channel establishment flow for different security modes.

2.5.2 Security policy management

If security mode 2 is required together with a high security level, an advanced security policy must be implemented. One possibility is to use a security manager that handles the security policy and enforces the security mechanism. An example of how a security manager can be implemented in Bluetooth is given in [11]. According to these recommendations, the security manager is the responsible entity for security enforcement and it interacts with several different layers in the stack (see Section 1.1.3). In this architecture, an application or set of applications (referred to as *service*) register their security demands with the security manager. The security requirements of all supported applications make up the security policy. The security manager handles the policy.

Since link-level security in Bluetooth is connected with the device address (through the link keys). The security manager needs access to a database, which contains information on different Bluetooth units, the corresponding link keys, and their level of trust. In addition to this, the manager needs access to a service database, which contains the specific security requirements of a particular service [12].

All the above techniques are of authentication and encryption are implemented at link level. The link adaptation protocol of Bluetooth protocol architecture can convert the information from upper layers to link layers [10].

Encryption and authentication is also supported by IPSec protocol one of the protocol suite from TCP/ IP protocol stack which can work at upper layers of Bluetooth protocol architecture [13][14].

IPSec is widely used in many applications specifically in Ad-hoc networks to secure communication [15] [16]. The application of IPSec in Ad-hoc networks suggests that it is the feasible choice to provide secure transmission in networks using Bluetooth as **Pico nets** and **scatter nets** [17].

Lot of vulnerabilities is present at link level security techniques in Bluetooth [17], which can be eliminated by using IPSec [13] [14].

CHAPTER 3

IPSec Protocol

3. IPSec Fundamentals

IPSec is a collection of protocols that assist in protecting communications over IP networks [19]. IPSec protocols work together in various combinations to provide protection for communications. This section will focus on the three primary components. The Encapsulating Security Payload (ESP), Authentication Header (AH), and Internet Key Exchange (IKE) protocols explaining the purpose and function of each protocol, and showing how they work together to create IPSec connections. Also, this section will discuss the value of using the IP Payload Compression Protocol (IPComp) as part of an IPSec implementation.

3.1 System Overview

This section provides a high level description of how IPSec works, the components of the system, and how they fit together to provide the security services noted above. The goal of this description is to enable the reader to "picture" the overall process/system, see how it fits into the IP environment, and to provide context for later sections of this chapter, which describe each of the components in more detail.

An IPSec implementation operates in a host, as a security gateway (SG), or as an independent device, Affording protection to IP traffic. (A security gateway is an intermediate system implementing IPSec, e.g., a firewall or router that has been IPSec-enabled.) The protection offered by IPSec is based on Requirements defined by a Security Policy Database (SPD) established and maintained by a user or system administrator, or by an application operating within constraints established by either of the above. In general, packets are selected for one of three processing actions based on IP and next layer header Information matched against entries in the SPD. Each packet is either PROTECTED using IPSec Security services, DISCARDED, or allowed to BYPASS IPSec protection, based on the applicable SPD Policies identified by the Selectors.

3.1.1 What IPSec Does

IPSec creates a boundary between unprotected and protected interfaces, for a host or a network. Traffic Traversing the boundary is subject to the access controls specified by the user or administrator responsible for the IPSec configuration. These controls indicate whether packets cross the boundary unimpeded, are afforded security services via AH or ESP, or are discarded. IPSec security services are offered at the IP Layer through selection of appropriate security protocols, cryptographic algorithms, and cryptographic Keys. IPSec can be used to protect one or more "paths":

a) Between a pair of hosts.
b) Between a pair of security gateways.
c) Between a security gateway and a host.

A compliant host implementation MUST support (a) and (c) and a compliant security Gateway must support all three of these forms of connectivity, since under certain circumstances a Security gateway acts as a host.

IPSec optionally supports negotiation of IP compression, motivated in part by the observation that when Encryption is employed within IPSec; it prevents effective compression by lower protocol layers.

3.1.2 How IPSec Works

IPSec uses two protocols to provide traffic security services Authentication Header (AH) and Encapsulating Security Payload (ESP).
IPSec implementations MUST support ESP and MAY support AH. (Support for AH has been Downgraded to MAY because experience has shown that there are very few contexts in which ESP cannot provide the requisite security services. Note that ESP can be used to provide only integrity, without confidentiality, making it comparable to AH in most contexts.) The IP Authentication Header (AH) offers integrity and data origin authentication, with optional (at the Discretion of the receiver) anti-replay features.

The Encapsulating Security Payload (ESP) protocol offers the same set of services, and also offers confidentiality. Use of ESP to provide confidentiality without integrity is NOT RECOMMENDED. When ESP is used with confidentiality enabled, there are provisions for limited traffic flow confidentiality, i.e., provisions for concealing packet length, and for facilitating efficient generation and discard of dummy packets.

This capability is likely to be effective primarily in virtual private network (VPN) and overlay network contexts.

Both AH and ESP offer access control, enforced through the distribution of cryptographic keys and the management of traffic flows as dictated by the Security Policy Database.

These protocols may be applied individually or in combination with each other to provide IPv4 and IPv6 Security services. However, most security requirements can be met through the use of ESP by itself.

Each protocol supports two modes of use: transport mode and tunnel mode. In transport mode, AH and ESP provide protection primarily for next layer protocols; in tunnel mode, AH and ESP are applied to Tunneled IP packets.

IPSec allows the user (or system administrator) to control the granularity at which a security service is Offered. For example, one can create a single encrypted tunnel to carry all the traffic between two Security gateways or a separate encrypted tunnel can be created for each TCP connection between each Pair of hosts communicating across these gateways. IPSec, through the SPD management paradigm, incorporates facilities for specifying which security protocol (AH or ESP) to employ, the mode (transport Or tunnel), security service options, what cryptographic algorithms to use, and in what combinations to Use the specified protocols and services, and the granularity at which protection should be applied.

Because most of the security services provided by IPSec require the use of cryptographic keys, IPSec relies on a separate set of mechanisms for putting these keys in place.

3.2 Authentication Header (AH)

AH [20], one of the IPSec security protocols, provides integrity protection for packet headers and data, as well as user authentication. It can optionally provide replay protection and access protection. AH cannot encrypt any portion of packets. In the initial version of IPSec, the ESP protocol could provide only encryption, not authentication, so AH and ESP were often used together to provide both confidentiality and integrity protection for communications. Because authentication capabilities were added to ESP in the second version of IPSec AH has become less significant; in fact, some IPSec software no longer supports AH. However, AH is still of value because AH can authenticate portions of packets that ESP cannot. Also, many existing IPSec implementations are using AH, so this guide includes a discussion of AH for completeness [21].

3.2.1 AH Modes

AH have two modes: transport and tunnel. In *tunnel mode*, AH creates a new IP header for each packet; in *transport mode*, AH does not create a new IP header. In IPSec architectures that use a gateway, the true source or destination IP address for packets must be altered to be the gateway's IP address. Because transport mode cannot alter the original IP header or create a new IP header, transport mode is generally used in host-to-host architectures [22]. As shown in Figures 3.1 and 3.2, AH provides integrity protection for the entire packet, regardless of which mode is used. (As explained in Section 3.2.2, IP header fields that can change unpredictably while in transit are not integrity-protected.)

New IP Header	AH Header	Original IP Header	Transport and Application Protocol Headers and Data
Authenticated (Integrity Protection)			

Figure 3.1 AH Tunnel Mode Packet

IP Header	AH Header	Transport and Application Protocol Headers and Data
Authenticated (Integrity Protection)		

Figure 3.2 AH Transport Mode Packet

3.2.2 Integrity Protection Process

The first step of integrity protection is to create a hash by using a keyed hash algorithm, also known as a message authentication code (MAC) algorithm. A standard hash algorithm generates a hash based on a message, while a *keyed hash algorithm* creates a hash based on both a message and a secret key shared by the two endpoints. The hash is added to the packet, and the packet is sent to the recipient. The recipient can then regenerate the hash using the shared key and confirm that the two hashes match, which provides integrity protection for the packet.

IPSec uses hash message authentication code (HMAC) algorithms,[23] which perform two keyed hashes. Examples of keyed hash algorithms are HMAC-MD5 and HMAC-SHA-1[24]. Another common MAC algorithm is AES Cipher Block Chaining MAC (AES-XCBC-MAC-96)[25].Technically, Figures 3.1 and 3.2 are somewhat misleading because it is not possible to protect the integrity of the entire IP header. Certain IP header fields, such as time to live (TTL) and the IP header checksum, are dynamic and may change during routine communications. If the hash is calculated on all the original IP header values, and some of those values legitimately change in transit, the recalculated hash will be different.

The destination would conclude that the packet had changed in transit and that its integrity had been violated. To avoid this problem, IP header fields that may legitimately change in transit in an unpredictable manner are excluded from the integrity protection calculations. This same principle explains why AH is often incompatible with network address translation (NAT) implementations. The IP source and destination address fields are included in the AH integrity protection calculations. If these addresses are altered by a NAT device (e.g., changing the source address from a private to a public address), the AH integrity protection calculation made by the destination will not match.

3.2.3 AH Header

AH adds a header to each packet. As shown in Figure 3.3, each AH header is composed of six fields:

1- **Next Header.** This field contains the IP protocol number for the next packet payload. In tunnel mode, the payload is an IP packet, so the Next Header value is set to 4 for IP-in-IP. In transport mode, the payload is usually a transport-layer protocol, often TCP (protocol number 6) or UDP (protocol number 17)

2- **Payload Length.** This field contains the length of the payload in 4-byte increments, minus 2.

3- **Reserved.** This value is reserved for future use, so it should be set to 0.

4- **Security Parameters Index (SPI) [26].** Each endpoint of each IPSec connection has an arbitrarily chosen SPI value, which acts as a unique identifier for the connection. The recipient uses the SPI value, along with the destination IP address and (optionally) the IPSec protocol type (in this case, AH), to determine which Security Association (SA) is being used. This tells the recipient which IPSec protocols and algorithms have been applied to the packet.

5- **Sequence Number.** Each packet is assigned a sequential sequence number, and only packets within a sliding window of sequence numbers are accepted. This provides protection against replay attacks because duplicate packets will use the same sequence number. This also helps to thwart denial of service attacks because old packets that are replayed will have sequence numbers outside the window, and will be dropped immediately without performing any more processing.

6- **Authentication Information.** This field contains the MAC output described in Section 3.2.2. The recipient of the packet can recalculate the MAC to confirm that the packet has not been altered in transit.

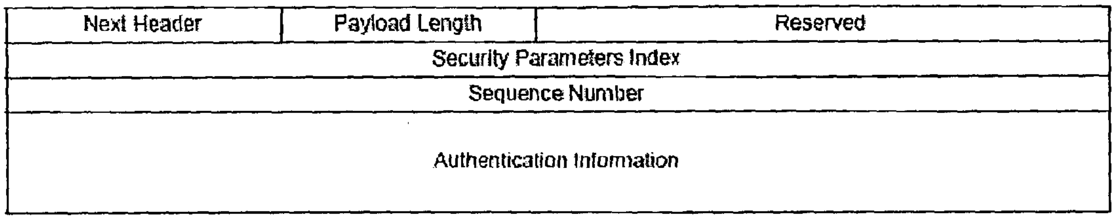
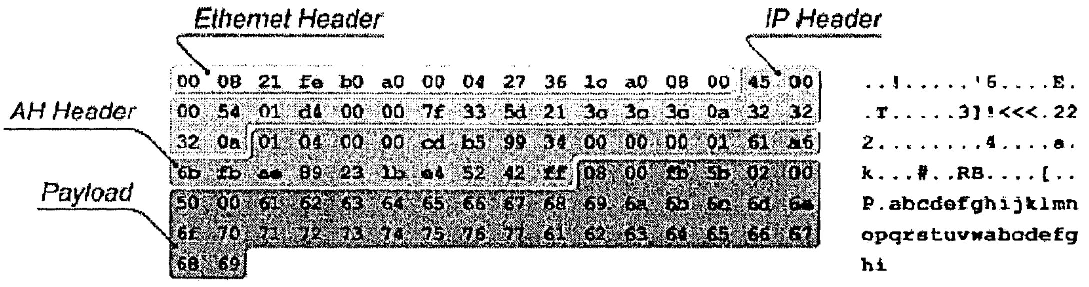


Figure 3.3 AH Header

3.2.4 How AH Works

The best way to understand how AH works is by reviewing and analyzing actual AH packets. Figure 3.4 shows the bytes that compose an actual AH packet. The values on the left side are the packet bytes in hex, and the values on the right side are attempted ASCII translations of each hex byte. (Bytes that cannot be translated into a printable ASCII character are represented by a dot.) Figure 3.4 indicates each section of the AH packet: Ethernet header, IP header, AH header, and payload [27].Based on the fields shown in Figures 3.1 and 3.2, this is a transport mode packet because it only contains a single IP header. In this case, the payload contains an ICMP echo request a ping. The original ping contained alphabetic sequences, represented in the packet by ascending hex values (e.g., 61, 62, and 63). After AH was applied, the ICMP payload is unaffected. This is because AH only provides integrity protection, not encryption.



```
00 08 21 fe b0 a0 00 04 27 36 1c a0 08 00 45 00   ..!.....'6....E.
00 54 01 d4 00 00 7f 33 5d 21 3c 3c 3c 0a 32 32   .T.....3]!<<<.22
32 0a 01 04 00 00 cd b5 99 34 00 00 00 01 61 a6   2.........4....a.
6b fb ae 89 23 1b e4 52 42 ff 08 00 fb 5b 02 00   k...#..RB....[..
50 00 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e   P.abcdefghijklmn
6f 70 71 72 73 74 75 76 77 61 62 63 64 65 66 67   opqrstuvwabcdefg
68 69                                             hi
```

Figure 3.4 Samples AH Transport Mode Packet

Figure 3.5 shows the AH header fields from the first four packets in an AH session between hosts A and B. The fields in the first header have been labeled, and they correspond to the fields identified in Figure 3.3. Items of interest are as follows:

1- **SPI:** Host A uses the hex value cdb59934 for the SPI in both its packets, while host B uses the hex value a6b32c00 for the SPI in both packets. This reflects that an AH connection is actually composed of two one-way connections, each with its own SPI.

2- **Sequence Number:** Both hosts initially set the sequence number to 1, and both incremented the number to 2 for their second packets.
3- **Authentication Information:** The authentication (integrity protection) information, which is a keyed hash based on virtually all the bytes in the packet, is different in each packet. This value should be different even if only one byte in a hashed section of the packet changes.

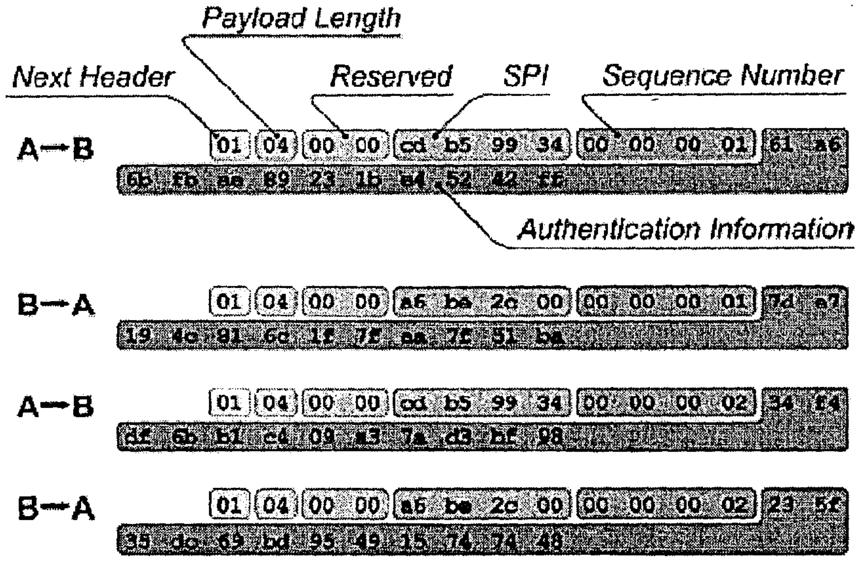


Figure 3-5. AH Header Fields from Sample Packet

3.2.5 AH Version 3

A new standard for AH, version 3, is currently in development[28].Based on the current standard draft, the functional differences between version 2 and version 3 should be relatively minor to IPSec administrators and users- some modifications to the SPI, and an optional longer sequence number. The version 3 standard draft also points to another standard draft that lists cryptographic algorithm requirements for AH[29].The draft mandates support for HMAC-SHA1-96, strongly recommends support for AES-XCBC-MAC-96, and also recommends support for HMAC-MD5-96.

3.2.6 AH Summary

- AH provides integrity protection for all packet headers and data, with the exception of a few IP header fields that routinely change in transit.
- Because AH includes source and destination IP addresses in its integrity protection calculations, AH is often incompatible with NAT.
- Currently, most IPSec implementations support the second version of IPSec, in which ESP can provide integrity protection services through authentication. The

use of AH has significantly declined. In fact, some IPSec implementations no longer support AH.

- AH still provides one benefit that ESP does not: integrity protection for the outermost IP header [30].

3.3 Encapsulating Security Payload (ESP)

ESP [31] is the second core IPSec security protocol. In the initial version of IPSec, ESP provided only encryption for packet payload data. Integrity protection was provided by the AH protocol if needed, as discussed in Section 3.2. In the second version of IPSec, ESP became more flexible. It can perform authentication to provide integrity protection, although not for the outermost IP header.

Also, ESP's encryption can be disabled through the Null ESP Encryption Algorithm. Therefore, in all but the oldest IPSec implementations, ESP can be used to provide only encryption; encryption and integrity protection; or only integrity protection [32]. This section mainly addresses the features and characteristics of the second version of ESP; the third version, currently in development, is described near the end of the section.

3.3.1 ESP Modes

ESP has two modes: transport and tunnel. In *tunnel mode*, ESP creates a new IP header for each packet. The new IP header lists the endpoints of the ESP tunnel (such as two IPSec gateways) as the source and destination of the packet. As shown in Figure 3.6, tunnel mode can encrypt and/or protect the integrity of both the data and the original IP header for each packet [33].Encrypting the data protects it from being accessed or modified by unauthorized parties; encrypting the IP header conceals the nature of the communications, such as the actual source or destination of the packet. If authentication is being used for integrity protection, each packet will have an ESP Authentication section after the ESP trailer.

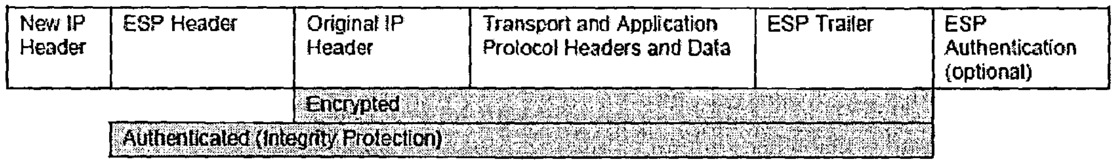

New IP Header
ESP Header
Original IP Header
Transport and Application Protocol Headers and Data
ESP Trailer
ESP Authentication (optional)
Encrypted
Authenticated (Integrity Protection)

Figure 3.6 ESP Tunnel Mode Packet

ESP tunnel mode is used far more frequently than ESP transport mode. In *transport mode*, ESP uses the original IP header instead of creating a new one. Figure 3.7 shows that in transport mode, ESP can only encrypt and/or protect the integrity of packet payloads and certain ESP components, but not IP headers. As with AH, ESP transport mode is generally only used in host-to-host architectures. Also, transport mode is incompatible with NAT. For example, in each TCP packet, the TCP checksum is calculated on both TCP and IP fields, including the source and destination addresses in the IP header. If NAT is being used, one or both of the IP addresses are altered, so NAT needs to recalculate the TCP checksum. If ESP is encrypting packets, the TCP header is encrypted; NAT cannot recalculate the checksum, so NAT fails. This is not an issue in tunnel mode; because the entire TCP packet is hidden, NAT will not attempt to recalculate the TCP checksum. However, tunnel mode and NAT have other potential compatibility issues [34]. Section 4.3.1 provides guidance on overcoming NAT-related issues.

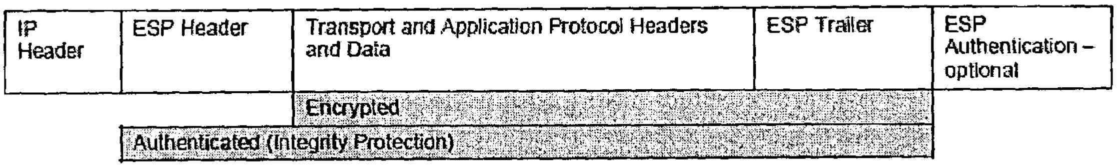

IP Header
ESP Header
Transport and Application Protocol Headers and Data
ESP Trailer
ESP Authentication – optional
Encrypted
Authenticated (Integrity Protection)

Figure 3.7 ESP Transport Mode Packet

3.3.2 Encryption Process

As described in Section 3.3, ESP uses symmetric cryptography to provide encryption for IPSec packets. Accordingly, both endpoints of an IPSec connection protected by ESP encryption must use the same key to encrypt and decrypt the packets. When an endpoint encrypts data, it divides the data into small blocks (for the AES algorithm, 128 bits each), and then performs multiple sets of cryptographic operations (known as *rounds*) using the data blocks and key. Encryption algorithms that work in this way are known as ***block cipher algorithms***.

When the other endpoint receives the encrypted data, it performs decryption using the same key and a similar process, but with the steps reversed and the cryptographic operations altered. Examples of encryption algorithms used by ESP are AES-Cipher Block Chaining (AES-CBC), AES Counter Mode (AES-CTR), and Triple DES (3DES) [35].

3.3.3 ESP Packet Fields

ESP adds a header and a trailer around each packets payload. As shown in Figure 3-8, each ESP header is composed of two fields:

- **SPI.** Each endpoint of each IPSec connection has an arbitrarily chosen SPI value, which acts as a unique identifier for the connection. The recipient uses the SPI value, along with the destination IP address and (optionally) the IPSec protocol type (in this case, ESP), to determine which SA is being used.
- **Sequence Number.** Each packet is assigned a sequential sequence number, and only packets within a sliding window of sequence numbers are accepted. This provides protection against replay attacks because duplicate packets will use the same sequence number. This also helps to thwart denial of service attacks because old packets that are replayed will have sequence numbers outside the window, and will be dropped immediately without performing any more processing.

The next part of the packet is the payload. It is composed of the payload data, which is encrypted, and the initialization vector (IV), which is not encrypted. The IV is used during encryption. Its value is different in every packet, so if two packets have the same content, the inclusion of the IV will cause the encryption of the two packets to have different results. This makes ESP less susceptible to cryptanalysis.

The third part of the packet is the ESP trailer, which contains at least two fields and may optionally include one more.

- **Padding.** An ESP packet may optionally contain *padding*, which is additional byte of data that make the packet larger and are discarded by the packet's recipient. Because ESP uses block ciphers for encryption, padding may be needed so that the encrypted data is an integral multiple of the block size. Padding may also be needed to ensure that the ESP trailer ends on a multiple of 4 bytes. Additional padding may also be used to alter the size of each packet, concealing how many bytes of actual data the packet contains. This is helpful in deterring traffic analysis.
- **Padding Length.** This number indicates how many bytes long the padding is. The Padding Length field is mandatory.
- **Next Header.** In tunnel mode, the payload is an IP packet, so the Next Header value is set to 4 for IP-in-IP. In transport mode, the payload is usually a transport-layer protocol, often TCP (protocol number 6) or UDP (protocol number 17). Every ESP trailer contains a Next Header value.

If ESP integrity protection is enabled, the ESP trailer is followed by an Authentication Information field. Like AH, the field contains the MAC output described in Section 3.2.2. Unlike AH, the MAC in ESP does not include the outermost IP header in its calculations. The recipient of the packet can recalculate the MAC to confirm that the portions of the packet other than the outermost IP header have not been altered in transit.

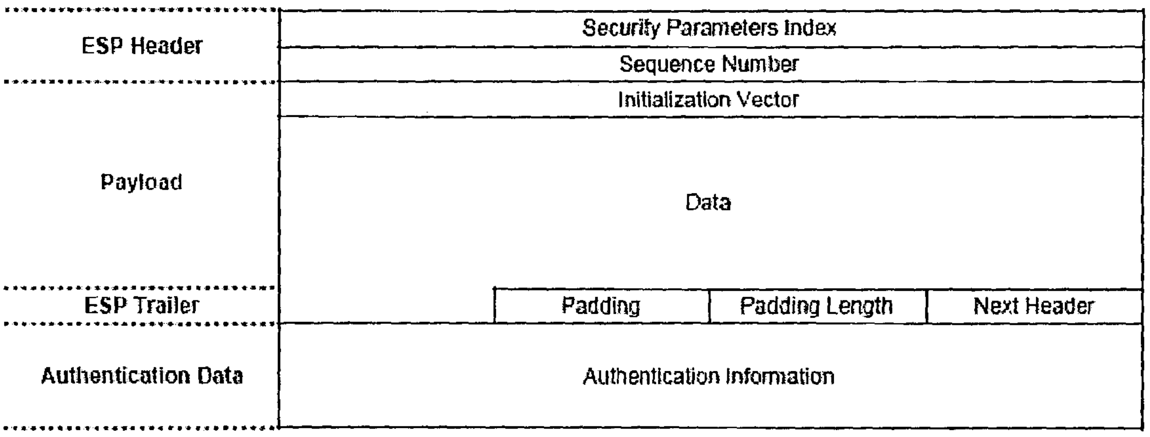
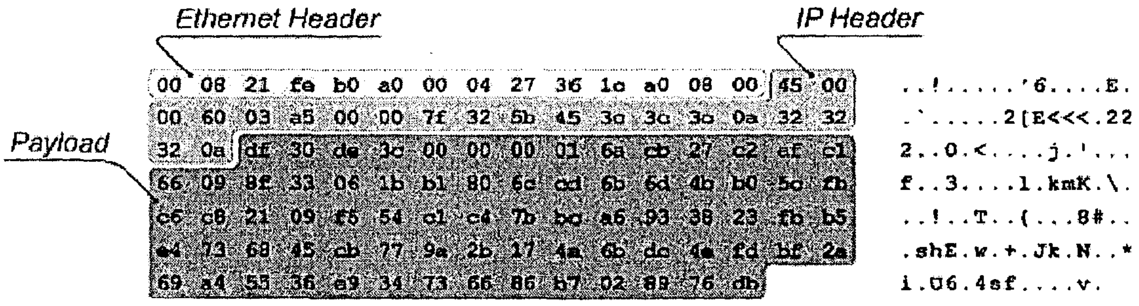


Figure 3.8 ESP Packet Fields

3.3.4 How ESP Works

Reviewing and analyzing actual ESP packets can provide a better understanding of how ESP works, particularly when compared with AH packets. Figure 3.9 shows the bytes that compose an actual ESP packet and their ASCII representations, in the same format used in Section 3.2.4. The alphabetic sequence that was visible in the AH-protected payload cannot be seen in the ESP-protected payload because it has been encrypted.

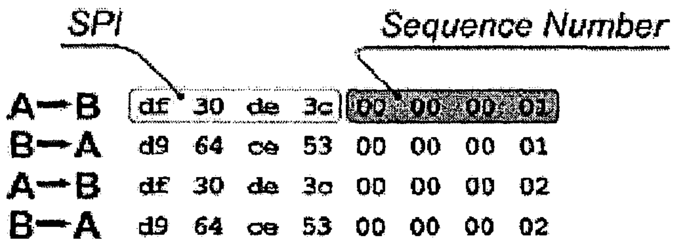
The ESP packet only contains five sections: Ethernet header, IP header, ESP header, encrypted data (payload and ESP trailer), and (optionally) authentication information. From the encrypted data, it is not possible to determine if this packet was generated in transport mode or tunnel mode. However, because the IP header is unencrypted, the IP protocol field in the header does reveal which protocol the payload uses (in this case, ESP). As shown in Figures 3.6 and 3.7, the unencrypted fields in both modes (tunnel and transport) are the same.



```
00 08 21 fe b0 a0 00 04 27 36 1c a0 08 00 45 00    ..!.....'6....E.
00 60 03 a5 00 00 7f 32 5b 45 3c 3c 3c 0a 32 32    .`.....2[E<<<.22
32 0a df 30 de 3c 00 00 00 01 6a cb 27 c2 af c1    2..0.<....j.'...
66 09 8f 33 06 1b b1 80 6c cd 6b 6d 4b b0 5c fb    f..3....l.kmK.\.
c6 c8 21 09 f5 54 c1 c4 7b bc a6 93 38 23 fb b5    ..!..T..{...8#..
e4 73 68 45 cb 77 9a 2b 17 4a 6b dc 4e fd bf 2a    .shE.w.+.Jk.N..*
69 a4 55 36 e9 34 73 66 86 b7 02 89 76 db          i.U6.4sf....v.
```

Figure 3.9 ESP Packet Capture

Although it is difficult to tell from Figure 3.9, the ESP header fields are not encrypted. Figure 3.10 shows the ESP header fields from the first four packets in an ESP session between hosts A and B. The SPI and Sequence Number fields work the same way in ESP that they do in AH. Each host uses a different static SPI value for its packets, which corresponds to an ESP connection being composed of two one-way connections, each with its own SPI. Also, both hosts initially set the sequence number to 1, and both incremented the number to 2 for their second packets.



```
A→B  df 30 de 3c 00 00 00 01
B→A  d9 64 ce 53 00 00 00 01
A→B  df 30 de 3c 00 00 00 02
B→A  d9 64 ce 53 00 00 00 02
```

Figure 3.10 ESP Header Fields from Sample Packets

3 .3.5 ESP Version 3

A new standard for ESP, version 3, is currently in development [36]. Based on the current standard draft, there should be several major functional differences between version 2 and version 3, including the following:

- The standard for ESP version 2 required ESP implementations to support using ESP encryption only (without integrity protection). The proposed ESP version 3 standards make support for this optional.
- ESP can use an optional longer sequence number, just like the proposed AH version 3 standard.
- ESP version 3 supports the use of combined mode algorithms (e.g., AES Counter with CBC-MAC [AES-CCM]) [37].Rather than using separate algorithms for encryption and integrity protection, a combined mode algorithm provides both encryption and integrity protection. The version 3 standard draft also points to another standard draft that lists encryption and integrity protection cryptographic algorithm requirements for ESP [33]. For encryption algorithms, the draft mandates support for the null encryption algorithm and 3DES-CBC, strongly recommends support for AES-CBC (with 128-bit keys), recommends support for AES-CTR, and discourages support for DES-CBC[39].For integrity protection algorithms, the draft mandates support for HMAC-SHA1-96 and the null authentication algorithm, strongly recommends support for AES-XCBC-MAC-96, and also recommends support for HMAC-MD5-96. The standard draft does not recommend any combined mode algorithms.

3.3.6 ESP Summary

- In tunnel mode, ESP can provide encryption and integrity protection for an encapsulated IP packet, as well as authentication of the ESP header. Tunnel mode can be compatible with NAT. However, protocols with embedded addresses (e.g., FTP, IRC, and SIP) can present additional complications.
- In transport mode, ESP can provide encryption and integrity protection for the payload of an IP packet, as well as integrity protection for the ESP header. Transport mode is not compatible with NAT.
- ESP tunnel mode is the most commonly used IPSec mode. Because it can encrypt the original IP header, it can conceal the true source and destination of the packet. Also, ESP can add padding to packets, further complicating attempts to perform traffic analysis.
- Although ESP can be used to provide encryption or integrity protection (or both), ESP encryption should not be used without integrity protection.

3.4 MD5 Algorithm

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

m_0 m_1 ... m_{b-1}

The following five steps are performed to compute the message digest of the message.

Step 1: Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.

Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

Step 2: Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M [0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

A four-word buffer (A, B, C, D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

Word A: 01 23 45 67
Word B: 89 ab cd ef
Word C: fe dc ba 98
Word D: 76 54 32 10

Step 3: Initialize MD Buffer

A four-word buffer (A, B, C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

Word A: 01 23 45 67
Word B: 89 ab cd ef

Word C: fe dc ba 98
Word D: 76 54 32 10

Step 4: Process Message in 16-Word Blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))

In each bit position F acts as a conditional: if X then Y else Z. The function F could have been defined using + instead of v since XY and not(X)Z will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F(X,Y,Z) will be independent and unbiased.

The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of G(X,Y,Z), H(X,Y,Z), and I(X,Y,Z) will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs.

This step uses a 64-element table T[1 ... 64] constructed from the sine function. Let T[i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians.

```
Do the following:
  /* Process each 16-word block. */
  For i = 0 to N/16-1 do
    /* Copy block i into X. */
    For j = 0 to 15 do
      Set X[j] to M [i*16+j].
    End /* of loop on j */

    /* Save A as AA, B as BB, C as CC, and D as DD. */
    AA = A
    BB = B

    CC = C
    DD = D

    /* Round 1. */
    /* Let [abcd k s i] denote the operation
       a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
    /* Do the following 16 operations. */
    [ABCD  0  7  1] [DABC  1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
    [ABCD  4  7  5] [DABC  5 12  6] [CDAB  6 17  7] [BCDA  7 22  8]
    [ABCD  8  7  9] [DABC  9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
```

```
[ABCD 12  7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

/* Round 2. */
/* Let [abcd k s i] denote the operation
   a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  1  5 17] [DABC  6  9 18] [CDAB 11 14 19] [BCDA  0 20 20]
[ABCD  5  5 21] [DABC 10  9 22] [CDAB 15 14 23] [BCDA  4 20 24]
[ABCD  9  5 25] [DABC 14  9 26] [CDAB  3 14 27] [BCDA  8 20 28]
[ABCD 13  5 29] [DABC  2  9 30] [CDAB  7 14 31] [BCDA 12 20 32]

/* Round 3. */
/* Let [abcd k s t] denote the operation
   a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  5  4 33] [DABC  8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD  1  4 37] [DABC  4 11 38] [CDAB  7 16 39] [BCDA 10 23 40]
[ABCD 13  4 41] [DABC  0 11 42] [CDAB  3 16 43] [BCDA  6 23 44]
[ABCD  9  4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA  2 23 48]

/* Round 4. */
/* Let [abcd k s t] denote the operation
   a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  0  6 49] [DABC  7 10 50] [CDAB 14 15 51] [BCDA  5 21 52]
[ABCD 12  6 53] [DABC  3 10 54] [CDAB 10 15 55] [BCDA  1 21 56]
[ABCD  8  6 57] [DABC 15 10 58] [CDAB  6 15 59] [BCDA 13 21 60]
[ABCD  4  6 61] [DABC 11 10 62] [CDAB  2 15 63] [BCDA  9 21 64]

/* Then perform the following additions. (That is increment each
   of the four registers by the value it had before this block
   was started.) */
A = A + AA
B = B + BB
C = C + CC
D = D + DD
```

Step 5: Output

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.

The MD5 message-digest algorithm is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD5 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.

3.5 SHA-1 Algorithm

Operations ion words. The following logical operators will be applied to words:

A. Bitwise logical word operations
   X AND Y = bitwise logical "and" of X and Y.
   X OR Y = bitwise logical "inclusive-or" of X and Y.
   X XOR Y = bitwise logical "exclusive-or" of X and Y.
   NOT X = bitwise logical "complement" of X.
   Example:

```
        01101100101110011101001001111011
XOR     01100101110000010110100110110111
        ----------+-----------------------
    =   00001001011110001011101111001100
```

B. The operation X + Y is defined as follows: words X and Y represent integers x and y, where 0 <= x < 2^32 and 0 <= y < 2^32. For positive integer's n and m, let n mod m be the remainder upon dividing n by m. Compute
   z = (x + y) mod 2^32.
   Then 0 <= z < 2^32. Convert z to a word, Z, and define Z = X + Y.

C. The circular left shift operation S^n(X), where X is a word and n is an integer with 0 <= n < 32, is defined by
   S^n(X) = (X << n) OR (X >> 32-n).

In the above, X << n is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). X >> n is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus S^n(X) is equivalent to a circular shift of X by n positions to the left.

3.5.1 Message Padding

SHA-1 is used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. The purpose of message padding is to make the total length of a padded message a multiple of 512. SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The following specifies how this padding shall be performed. As a summary, a "1" followed by m "0"s followed by a 64- bit integer are appended to the end of the message to produce a padded message of length 512 * n. The 64-bit integer is the length of the original message. The padded message is then processed by the SHA-1 as n 512-bit blocks.

Suppose a message has length l < 2^64. Before it is input to the SHA-1, the message is padded on the right as follows:

1. "1" is appended. Example: if the original message is "01010000", this is padded to "010100001".
2. "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length l of the original message.

Example: Suppose the original message is the bit string
01100001 01100010 01100011 01100100 01100101.

After step (1) this gives
01100001 01100010 01100011 01100100 01100101 1.

Since l = 40, the number of bits in the above is 41 and 407 "0"s are appended, making the total now 448. This gives (in hex)

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000.
```

3. Obtain the 2-word representation of l, the number of bits in the original message. If l < 2^32 then the first word is all zeroes.

Append these two words to the padded message.

Example: Suppose the original message is as in (b). Then l = 40 (note that l is computed before any padding). The two-word representation of 40 is hex 00000000 00000028. Hence the final padded message is hex

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028.
```

The padded message will contain 16 * n words for some n > 0.

The padded message is regarded as a sequence of n blocks M(1) , M(2), first characters (or bits) of the message.

3.5.2 Functions and Constants Used

A sequence of logical functions f(0), f(1),..., f(79) is used in SHA-1. Each f(t), 0 <= t <= 79, operates on three 32-bit words B, C, D and produces a 32-bit word as output. f(t;B,C,D) is defined as follows: for words B, C, D,

f(t;B,C,D) = (B AND C) OR ((NOT B) AND D) (0 <= t <= 19)
f(t;B,C,D) = B XOR C XOR D (20 <= t <= 39)
f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t <= 59)
f(t;B,C,D) = B XOR C XOR D (60 <= t <= 79).

A sequence of constant words K(0), K(1), ... , K(79) is used in the SHA-1. In hex these are given by

K(t) = 5A827999 (0 <= t <= 19)
K(t) = 6ED9EBA1 (20 <= t <= 39)
K(t) = 8F1BBCDC (40 <= t <= 59)
K(t) = CA62C1D6 (60 <= t <= 79).

3.5.3 Computing the Message Digest

The methods given in 3.5.3.1 and 3.5.3.2 below yield the same message digest. Although using method 2 saves sixty-four 32-bit words of storage, it is likely to lengthen execution time due to the increased complexity of the address computations for the { W[t] } in step (3). There are other computation methods which give identical results.

3.5.3.1 Method 1

The computation is described using two buffers, each consisting of five 32-bit words, and a sequence of eighty 32-bit words. The words of the first 5-word buffer are labeled A,B,C,D,E. The words of the second 5-word buffer are labeled H0, H1, H2, H3, H4. The words of the 80-word sequence are labeled W(0), W(1),..., W(79). A single word buffer TEMP is also employed. To generate the message digest, the 16-word blocks M(1), M(2),..., M(n) defined in section 4 are processed in order. The processing of each M(i) involves 80 steps.

Before processing any blocks, the H's are initialized as follows: in hex,

H0 = 67452301
H1 = EFCDAB89
H2 = 98BADCFE
H3 = 10325476
H4 = C3D2E1F0.

Now M(1), M(2), ... , M(n) are processed. To process M(i), we proceed as follows:

a. Divide M(i) into 16 words W(0), W(1), ... , W(15), where W(0) is the left-most word.
b. For t = 16 to 79 let W(t) = S^1(W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)).
c. Let A = H0, B = H1, C = H2, D = H3, E = H4.
d. For t = 0 to 79 do TEMP = S^5(A) + f(t;B,C,D) + E + W(t) + K(t); E = D; D = C; C = S^30(B); B = A; A = TEMP;
e. Let H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

After processing M(n), the message digest is the 160-bit string represented by the 5 words H0 H1 H2 H3 H4.

3.5.3.2 Method 2

The method above assumes that the sequence W(0), ... , W(79) is implemented as an array of eighty 32-bit words. This is efficient from the standpoint of minimization of execution time, since the addresses of W(t-3), ... ,W(t-16) in step (b) are easily computed. If space is at a premium, an alternative is to regard { W(t) } as a circular queue, which may be implemented using an array of sixteen 32-bit words W[0], ... W[38]. In this case, in hex let

MASK = 0000000F. Then processing of M(i) is as follows:

a. Divide M(i) into 16 words W[0], ... , W[38], where W[0] is the left-most word.
b. Let A = H0, B = H1, C = H2, D = H3, E = H4.
c. For t = 0 to 79 do s = t AND MASK;
If (t >= 16) W[s] = S^1(W[(s + 13) AND MASK] XOR W[(s + 8) AND MASK] XOR W[(s + 2) AND MASK] XOR W[s]);
TEMP = S^5(A) + f(t;B,C,D) + E + W[s] + K(t);
E = D; D = C; C = S^30(B); B = A; A = TEMP;

d. Let H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

3.6 DES Algorithm

DES is a ***block cipher***--meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. Thus DES results in a ***permutation*** among the 2^64 (read this as: "2 to the 64th power") possible arrangements of 64 bits, each of which may be either 0 or 1. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block **L** and a right half **R**. (This division is only used in certain operations.)

Example: Let **M** be the plain text message **M** = 0123456789ABCDEF, where **M** is in hexadecimal (base 16) format. Rewriting **M** in binary format, we get the 64-bit block of text:

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
L = 0000 0001 0010 0011 0100 0101 0110 0111
R = 1000 1001 1010 1011 1100 1101 1110 1111

The first bit of **M** is "0". The last bit is "1". We read from left to right.

DES operates on the 64-bit blocks using *key* sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used (i.e. bits numbered 8, 16, 24, 32, 40, 48, 56, and 64). However, we will nevertheless number the bits from 1 to 64, going left to right, in the following calculations. But, as you will see, the eight bits just mentioned get eliminated when we create sub keys.

Example: Let **K** be the hexadecimal key **K** = 133457799BBCDFF1. This gives us as the binary key (setting 1 = 0001, 3 = 0011, etc., and grouping together every eight bits, of which the last one in each group will be unused):

K = *00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001*

The DES algorithm uses the following steps:

Step 1: Create 16 sub keys, each of which is 48-bits long.

The 64-bit key is permuted according to the following table, **PC-1**. Since the first entry in the table is "57", this means that the 57th bit of the original key **K** becomes the first bit of the permuted key K+. The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Note only 56 bits of the original key appear in the permuted key.

PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Example: From the original 64-bit key

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

We get the 56-bit permutation

K+ = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

Next, split this key into left and right halves, C_0 and D_0, where each half has 28 bits.

Example: From the permuted key K+, we get

C_0 = 1111000 0110011 0010101 0101111

D_0 = *0101010 1011001 1001111 0001111*

With C_0 and D_0 defined, we now create sixteen blocks C_n and D_n, 1<=*n*<=16. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1}, respectively, for n = 1, 2... 16, using the following schedule of "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2

6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

This means, for example, C_3 and D_3 are obtained from C_2 and D_2, respectively, by two left shifts, and C_{16} and D_{16} are obtained from C_{15} and D_{15}, respectively, by one left shift. In all cases, by a single left shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3,..., 28, 1.

Example: From original pair pair C_0 and D_0 we obtain:

C_0 = 1111000011001100101010101111
D_0 = 0101010101100110011110001111
C_1 = 1110000110011001010101011111
D_1 = 1010101011001100111100011110
C_2 = 1100001100110010101010111111
D_2 = 0101010110011001111000111101
C_3 = 0000110011001010101011111111
D_3 = 0101011001100111100011110101
C_4 = 0011001100101010101111111100
D_4 = 0101100110011110001111010101
C_5 = 1100110010101010111111110000
D_5 = 0110011001111000111101010101
C_6 = 0011001010101011111111000011
D_6 = 1001100111100011110101010101
C_7 = 1100101010101111111100001100
D_7 = 0110011110001111010101010110
C_8 = 0010101010111111110000110011
D_8 = 1001111000111101010101011001
C_9 = 0101010101111111100001100110
D_9 = 0011110001111010101010110011
C_{10} = 0101010111111110000110011001
D_{10} = 1111000111101010101011001100
C_{11} = 0101011111111000011001100101
D_{11} = 1100011110101010101100110011
C_{12} = 0101111111100001100110010101
D_{12} = 0001111010101010110011001111
C_{13} = 0111111110000110011001010101
D_{13} = 0111101010101011001100111100

C_{14} = 1111111000011001100101010101
D_{14} = 1110101010101100110011110001
C_{15} = 1111100001100110010101010111
D_{15} = 1010101010110011001111000111
C_{16} = 1111000011001100101010101111
D_{16} = 0101010101100110011110001111
We now form the keys K_n, for $1<=n<=16$, by applying the following permutation table to each of the concatenated pairs C_nD_n. Each pair has 56 bits, but **PC-2** only uses 48 of these.

PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Therefore, the first bit of K_n is the 14th bit of C_nD_n, the second bit the 17th, and so on, ending with the 48th bit of K_n being the 32th bit of C_nD_n.

Example: For the first key we have C_1D_1 = 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110 which, after we apply the permutation **PC-2**, becomes
K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
For the other keys we have

K_2	=	011110	011010	111011	011001	110110	111100	100111	100101
K_3	=	010101	011111	110010	001010	010000	101100	111110	011001
K_4	=	011100	101010	110111	010110	110110	110011	010100	011101
K_5	=	011111	001110	110000	000111	111010	110101	001110	101000
K_6	=	011000	111010	010100	111110	010100	000111	101100	101111
K_7	=	111011	001000	010010	110111	111101	100001	100010	111100
K_8	=	111101	111000	101000	111010	110000	010011	101111	111011
K_9	=	111000	001101	101111	101011	111011	011110	011110	000001
K_{10}	=	101100	011111	001101	000111	101110	100100	011001	001111
K_{11}	=	001000	010101	111111	010011	110111	101101	001110	000110
K_{12}	=	011101	010111	000111	110101	100101	000110	011111	101001
K_{13}	=	100101	111100	010111	010001	111110	101011	101001	000001
K_{14}	=	010111	110100	001110	110111	111100	101110	011100	111010
K_{15}	=	101111	111001	000110	001101	001111	010011	111100	001010

K_{16} = 110010 110011 110110 001011 000011 100001 011111 110101

So, much for the sub keys. Now we look at the message itself.

Step 2: Encode each 64-bit block of data.

There is an *initial permutation* **IP** of the 64 bits of the message data **M**. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order. The 58th bit of **M** becomes the first bit of **IP**. The 50th bit of **M** becomes the second bit of **IP**. The 7th bit of **M** is the last bit of **IP**.

IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Example: Applying the initial permutation to the block of text **M**, given previously, we get
M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
IP = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010
Here the 58th bit of **M** is "1", which becomes the first bit of **IP**. The 50th bit of **M** is "1", which becomes the second bit of **IP**. The 7th bit of **M** is "0", which becomes the last bit of **IP**.
Next divide the permuted block **IP** into a left half L_0 of 32 bits, and a right half R_0 of 32 bits.
Example: From **IP**, we get L_0 and R_0
L_0 = 1100 1100 0000 0000 1100 1100 1111 1111
R_0 = 1111 0000 1010 1010 1111 0000 1010 1010
We now proceed through 16 iterations, for $1<=n<=16$, using a function f which operates on two blocks--a data block of 32 bits and a key K_n of 48 bits--to produce a block of 32 bits. **Let + denote XOR addition, (bit-by-bit addition modulo 2).** Then for **n** going from 1 to 16 we calculate
$L_n = R_{n-1}$
$R_n = L_{n-1} + f(R_{n-1},K_n)$
This results in a final block, for $n = 16$, of $L_{16}R_{16}$. That is, in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For the right 32 bits in the current step, we XOR the left 32 bits of the previous step with the calculation f.
Example: For $n = 1$, we have
K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
$L_1 = R_0$ = 1111 0000 1010 1010 1111 0000 1010 1010
$R_1 = L_0 + f(R_0,K_1)$

It remains to explain how the function *f* works. To calculate *f*, we first expand each block R_{n-1} from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in R_{n-1} . We'll call the use of this selection table the function E. Thus E (R_{n-1}) has a 32 bit input block, and a 48 bit output block.
Let E be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to the following table:

E BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Thus the first three bits of E(R_{n-1}) are the bits in positions 32, 1 and 2 of R_{n-1} while the last 2 bits of E(R_{n-1}) are the bits in positions 32 and 1.
Example: We calculate E(R_0) from R_0 as follows:
R_0 = 1111 0000 1010 1010 1111 0000 1010 1010
E (R_0) = 011110 100001 010101 010101 011110 100001 010101 010101
(Note that each block of 4 original bits has been expanded to a block of 6 output bits.)
Next in the *f* calculation, we XOR the output E(R_{n-1}) with the key K_n:
K_n + E (R_{n-1}).
Example: For K_1 , E(R_0), we have
K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
E(R_0) = 011110 100001 010101 010101 011110 100001 010101 010101
K_1+E(R_0) = 011000 010001 011110 111010 100001 100110 010100 100111.
We have not yet finished calculating the function *f*. To this point we have expanded R_{n-1} from 32 bits to 48 bits, using the selection table, and XORed the result with the key K_n . We now have 48 bits, or eight groups of six bits. We now do something strange with each group of six bits: we use them as addresses in tables called "**S boxes**". Each group of six bits will give us an address in a different S box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the S boxes) for 32 bits total.
Write the previous result, which is 48 bits, in the form:
K_n + E(R_{n-1}) $=B_1B_2B_3B_4B_5B_6B_7B_8$,
Where each B_i is a group of six bits. We now calculate
$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$
Where $S_i(B_i)$ referres to the output of the *i*-th S box.
To repeat, each of the functions *S1, S2,..., S8*, takes a 6-bit block as input and yields a 4-bit block as output. The table to determine S_1 is shown and explained below:

S1

Column Number

Row No.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	**14**	**4**	**13**	**1**	**2**	**15**	**11**	**8**	**3**	**10**	**6**	**12**	**5**	**9**	**0**	**7**
1	**0**	**15**	**7**	**4**	**14**	**2**	**13**	**1**	**10**	**6**	**12**	**11**	**9**	**5**	**3**	**8**
2	**4**	**1**	**14**	**8**	**13**	**6**	**2**	**11**	**15**	**12**	**9**	**7**	**3**	**10**	**5**	**0**
3	**15**	**12**	**8**	**2**	**4**	**9**	**1**	**7**	**5**	**11**	**3**	**14**	**10**	**0**	**6**	**13**

If S_1 is the function defined in this table and B is a block of 6 bits, then $S_1(B)$ is determined as follows: The first and last bits of B represent in base 2 a number in the decimal range 0 to 3 (or binary 00 to 11). Let that number be i. The middle 4 bits of B represent in base 2 a number in the decimal range 0 to 15 (binary 0000 to 1111). Let that number be j. Look up in the table the number in the i-th row and j-th column. It is a number in the range 0 to 15 and is uniquely represented by a 4 bit block. That block is the output $S_1(B)$ of S_1 for the input B. For example, for input block B = 011011 the first bit is "0" and the last bit "1" giving 01 as the row. This is row 1. The middle four bits are "1101". This is the binary equivalent of decimal 13, so the column is column number 13. In row 1, column 13 appears 5. This determines the output; 5 is binary 0101, so that the output is 0101. Hence S_1(011011) = 0101.

The tables defining the functions $S_1,...,S_8$ are the following:

S1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Example: For the first round, we obtain as the output of the eight S boxes:
K_1 + E(R_0) = 011000 010001 011110 111010 100001 100110 010100 100111.
$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

The final stage in the calculation of f is to do a permutation **P** of the S-box output to obtain the final value of f:
f = P($S_1(B_1)S_2(B_2)...S_8(B_8)$)

The permutation **P** is defined in the following table. **P** yields a 32-bit output from a 32-bit input by permuting the bits of the input block.

P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Example: From the output of the eight S boxes:
$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111
We get
f= 0010 0011 0100 1010 1010 1001 1011 1011

$R_1 = L_0 + f(R_0, K_1)$

=	1100	1100	0000	0000	1100	1100	1111	1111
+	0010	0011	0100	1010	1010	1001	1011	1011

= 1110 1111 0100 1010 0110 0101 0100 0100

In the next round, we will have $L_2 = R_1$, which is the block we just calculated, and then we must calculate $R_2 = L_1 + f(R_1, K_2)$, and so on for 16 rounds. At the end of the sixteenth round we have the blocks L_{16} and R_{16}. We then ***reverse*** the order of the two blocks into the 64-bit block

$R_{16}L_{16}$ and apply a final permutation $\mathbf{IP^{-1}}$ as defined by the following table:

$\mathbf{IP^{-1}}$

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

Example: If we process all 16 blocks using the method defined previously, we get, on the 16th round,

L_{16} = 0100 0011 0100 0010 0011 0010 0011 0100

R_{16} = 0000 1010 0100 1100 1101 1001 1001 0101

We reverse the order of these two blocks and apply the final permutation to

$R_{16}L_{16}$ = 00001010 01001100 11011001 10010101 01000011 01000010 00110010 00110100

IP^{-1} = 10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101 which in hexadecimal format is 85E813540F0AB405.

This is the encrypted form of **M** = 0123456789ABCDEF: namely, **C** = 85E813540F0AB405.

Decryption is simply the inverse of encryption, following the same steps as above, but reversing the order in which the sub keys are applied.

3.7 Triple DES Encryption

The Data Encryption Standard (DES) was developed by an IBM team around 1974 and adopted as a national standard in 1977. Triple DES is a minor variation of this standard. It is three times slower than regular DES but can be billions of times more secure if used properly. Triple DES enjoys much wider use than DES because DES is so easy to break with today's rapidly advancing technology. In 1998 the Electronic Frontier Foundation, using a specially developed computer called the DES Cracker, managed to break DES in less than 3 days. And this was done for under $250,000. The encryption chip that powered the DES Cracker was capable of processing 88 billion keys per second. In addition, it has been shown that for a cost of one million dollars a dedicated hardware device can be built that can search all possible DES keys in about 3.5 hours. This just serves to illustrate that any organization with moderate resources can break through DES with very little effort these days. No sane security expert would consider using DES to protect data.

Triple DES was the answer to many of the shortcomings of DES. Since it is based on the DES algorithm, it is very easy to modify existing software to use Triple DES. It also has the advantage of proven reliability and a longer key length that eliminates many of the shortcut attacks that can be used to reduce the amount of time it takes to break DES. However, even this more powerful version of DES may not be strong enough to protect data for very much longer. The DES algorithm itself has become obsolete and is in need of replacement. To this end the National Institute of Standards and Technology (NIST) is holding a competition to develop the Advanced Encryption Standard (AES) as a replacement for DES. Triple DES has been endorsed by NIST as a temporary standard to be used until the AES is finished sometime in 2001.

The AES will be at least as strong as Triple DES and probably much faster. Many security systems will probably use both Triple DES and AES for at least the next five years. After that, AES may supplant Triple DES as the default algorithm on most systems if it lives up to its expectations. But Triple DES will be kept around for compatibility reasons for many years after that. So the useful lifetime of Triple DES is far from over, even with the AES near completion. For the foreseeable future Triple DES is an excellent and reliable choice for the security needs of highly sensitive information.

3.7.1 In Depth

Triple DES is simply another mode of DES operation. It takes three 64-bit keys, for an overall key length of 192 bits. In Private Encryptor, you simply type in the entire 192-bit (24 character) key rather than entering each of the three keys individually. The Triple DES DLL then breaks the user provided key into three sub keys, padding the keys if necessary so they are each 64 bits long. The procedure for encryption is exactly the same as regular DES, but it is repeated three times. Hence the name Triple DES. The data is encrypted with the first key, decrypted with the second key, and finally encrypted again with the third key.

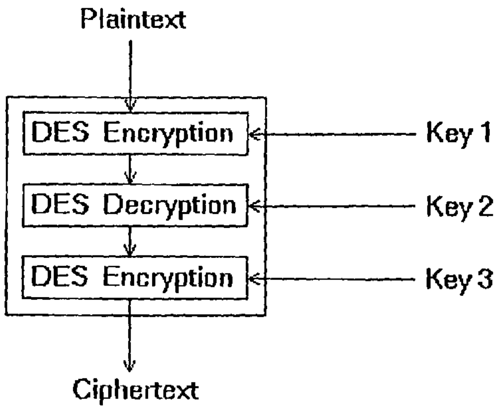


Figure 3.11 3DES

Consequently, Triple DES runs three times slower than standard DES, but is much more secure if used properly. The procedure for decrypting something is the same as the procedure for encryption, except it is executed in reverse. Like DES, data is encrypted and decrypted in 64-bit chunks. Unfortunately, there are some weak keys that one should be aware of: if all three keys, the first and second keys, or the second and third keys are the same, then the encryption procedure is essentially the same as standard DES. This situation is to be avoided because it is the same as using a really slow version of regular DES .

Note that although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length. The least significant (right-most) bit in each byte is a parity bit, and should be set so that there are always an odd number of 1s in every byte. These parity bits are ignored, so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits. This means that the effective key strength for Triple DES is actually 168 bits because each of the three keys contains 8 parity bits that are not used during the encryption process.

3.7.2 Modes of Operation

3.7.2.1 Triple ECB (Electronic Code Book)

This variant of Triple DES works exactly the same way as the ECB mode of DES. Triple ECB is the type of encryption used by Private Encrypted. This is the most commonly used mode of operation.

3.7.2.2 Triple CBC (Cipher Block Chaining)

This method is very similar to the standard DES CBC mode. As with Triple ECB, the effective key length is 168 bits and keys are used in the same manner, as described above, but the chaining features of CBC mode are also employed. The first 64-bit key acts as the Initialization Vector to DES. Triple ECB is then executed for a single 64-bit block of plaintext. The resulting cipher text is then XORed with the next plaintext block to be encrypted, and the procedure is repeated. This method adds an extra layer of security to Triple DES and is therefore more secure than Triple ECB, although it is not used as widely as Triple ECB.

CHAPTER 4

Methodology

4. Methodology

In this chapter first we discuss the problem area of our research and cover the security weaknesses of Bluetooth security. Then we suggest the solution and methodology.

4.1 Security weaknesses

There are many security weaknesses in the Bluetooth standard. Some of these problems can very easily be exploited by an attacker; other security weaknesses are rather theoretical. An extensive overview of the most important problems will now be given

4.1.1 Security depends on security of PIN

The initialization key is a function of a random number IN RAND, a shared PIN and the length L of the PIN. The random number is sent in clear and hence known by an attacker that is present during the initialization phase. Note that it is not so difficult for an attacker to obtain this random number.

He can place some (small) devices near the two Bluetooth devices that are going to be paired or even place a small sensor on one of the devices. This means that only the PIN is a secret value, all the rest is public. If an attacker obtains the PIN, he knows the initialization key. It even gets worse! Since all the other keys are derived from the initialization key, they also will be known by the attacker. The security of the keys depends on the security of the PIN. If it is too short or weak (e.g., 0000), it is very easy for an attacker to guess the PIN.

Note that it is always possible to guess the PIN. The reason is that a mutual authentication protocol is executed after the generation of the initialization key. If an attacker observes this protocol, he obtains a challenge and the corresponding response. It is now very easy to perform a brute force attack. The attacker tries every PIN and calculates for every PIN the corresponding response. When the calculated response is equal to the observed response, the correct PIN is used. The shorter the PIN, the faster this brute force attack can be executed.

Sometimes a fixed PIN is used (the default value is 0000) or the PIN is sent in clear to the other device. In those cases, the PIN is publicly known and the keys only depend on public values. It is then trivial for an attacker to obtain the secret keys. This certainly has to be avoided in security-sensitive applications.

4.1.2 Unit key

The unit key is used if one of the Bluetooth devices does not have enough memory to store session keys. This key is stored in non-volatile memory and almost never changed. The unit key is sent encrypted (with the initialization key) to the other device. This is not very secure! Suppose A has sent its unit key to device B. The result is that B now knows the key of A and can use this key itself. B can send this unit key to C and impersonate itself as A. It is impossible for C to detect this impersonation attack. This is why the use of unit keys should be avoided.

4.1.3 Encryption algorithms

Bluetooth uses the encryption algorithm E0 and E1. This stream cipher has some security flaws.

The attacks with the lowest complexity are the algebraic attacks. E0 is vulnerable to algebraic attacks because of the possibility to recover the initial value by solving a system of non-linear equations of degree 4 over the finite field GF (2). This system can be transformed by linearization into a system of linear independent equations with at most 223 unknowns. Fortunately, this attack does not work in Bluetooth because it needs a long key stream during the initialization and E0 in Bluetooth only uses small packets (the payload ranges from zero to a maximum of 2745 bits.

There is however an attack which can be implemented on the E0 algorithm in Bluetooth. Golic has found an attack on the Bluetooth stream cipher that can reconstruct the 128-bit secret key with complexity about 270 from about 45 initializations. In the pre computation stage, a database of about 280 103-bit words has to be sorted out. The attack uses a general linear iterative cryptanalysis method for solving binary systems of approximate linear equations.

Irrespective of the security mode used, encryption of data during transmission is only optional and has to be explicitly requested by the applications.

Problems with E0 encryption algorithm are:

- Output (KCIPHER) = combination of 4 LFSRs
- Key (KC) = 128 bits
- Best attack: guess some registers 2 to the power of 266 (memory and complexity)

Problems with E1 encryption algorithm are:

- E1 = SAFER+
- Some security weaknesses (although not applicable to Bluetooth)
- slow

4.1.4 Denial of service attacks

Mobile networks are vulnerable to denial of service attacks. They consist of mobile devices and these devices are often battery fed. Bluetooth is no exception. An attacker can send dummy messages to a mobile device.

When this device receives a message (a real of a fake one), it consumes some computation (and battery) power. After some time, all battery power will be consumed and the device won't be available anymore. This exhaustion of the battery power is called the sleep deprivation attack. There are a lot more denial of service attacks. The attacker can try to interfere with the radio propagation. Bluetooth uses the 2.4 GHz ISM band, which is also used by some other mobile networks (e.g., WIFI). To avoid the interference caused by other mobile networks or an attacker, frequency hopping and spread spectrums are used in Bluetooth.

There is also some denial of service attacks caused by implementation decisions. An example is the black list which is used during the mutual authentication procedure. To avoid that a device would start a mutual authentication procedure over and over again, each device has a black list with the Bluetooth addresses of the devices which failed to authenticate themselves correctly. These devices can not start an authentication procedure during some period. This period will be increased exponentially (until a certain upper limit is reached) if the authentication fails again.

The black list is used to avoid a denial of service attack (successive wrong authentication procedures), but in fact opens the door for other DoS attacks. An attacker can try to authenticate to device A, but change every time its address. All these authentication attempts will fail and the black list of A will become quite large. If there is no upper limit on this black list, the entire memory of A will be filled with the entries of the black list and device A will crash.

This is not the only DoS attack. Suppose device B wants to authenticate to A. After A has sent a random number (the challenge) to B (this is the first step in the authentication procedure), the attacker sends a wrong response to A using the Bluetooth address of B. The authentication will fail, B will be put on the black list of A and hence the (correct) response of B will be ignored by A. The attacker keeps repeating this attack and B will never be able to authenticate successful to A.

4.1.5 Location Privacy

When two or more Bluetooth devices are communicating, the transmitted packets always contain the Bluetooth address of the sender and the receiver.

When an attacker eavesdrops on the transmitted data, he knows the Bluetooth addresses of the devices which were communicating (the attacker can do this by placing a small device near the two Bluetooth devices). This way, the attacker can keep track of the place and the time these two devices were communicating. It is also quite probable that the two devices are from the same user (most of the communication takes place between devices of the same owner).

This is a violation of the privacy of the user. The location information can be sold to other persons and used for location dependent commercial advertisements (e.g., a shop can send advertisements to all the users which are near the shop). It should be possible for the user to decide him self when his location is revealed and when not.

4.1.6 Other security problems

There are also some security problems in the challenge-response protocol. Another security flaw is that:

1) No integrity check on the Bluetooth packets.
2) An attacker can always modify or replace a transmitted Bluetooth packet.
3) Man-in-the-middle attacks are also not prevented in Bluetooth.
4) A user can switch off security. Often, the default configuration is no security at all.

4.2 Bluetooth Network Encapsulation Protocol

The functional requirement for Bluetooth networking encapsulation protocol (BNEP) includes the following [47]:

- Support for common networking protocols such as IPv4, IPv6, IPX, and other existing or emerging networking protocols.
- Low Overhead -- The encapsulation format SHALL be bandwidth efficient.

BNEP is used for transporting both control and data packet over Bluetooth to provide networking capabilities for Bluetooth devices. BNEP provides capabilities that are similar to capabilities provided by Ethernet (Ethernet/DIX Framing /IEEE 802.3). The following diagram illustrates stack overview [40] [47].

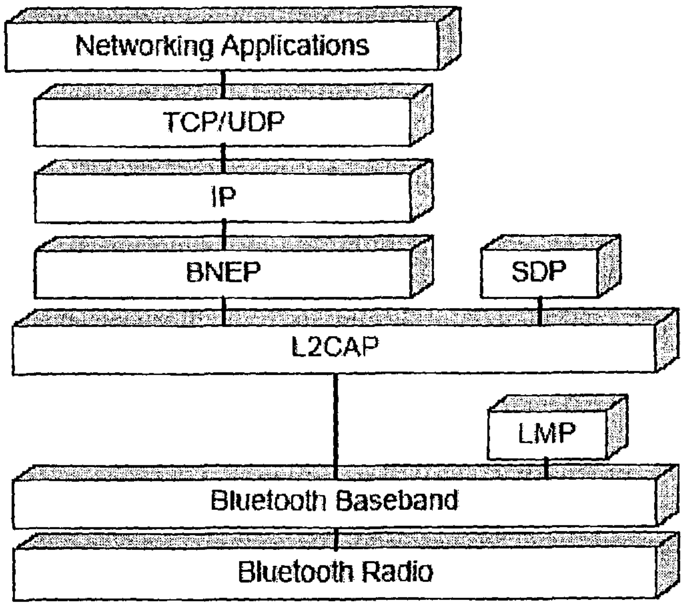


Figure 4.1: BNEP Stack

4.2.1 BNEP Header Format

The following diagram illustrates the BNEP header format.

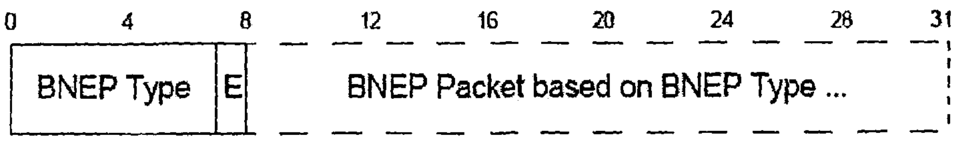


Figure 4.2: BNEP Format

BNEP Type: Seven bit Bluetooth Network Encapsulation Protocol
Type value identifies the type of BNEP header contained in this packet [47].
Extension Flag (E): One bit extension flag that indicates if one or more extension headers follow the BNEP Header before the data payload if the data payload exists. If the extension flag is equal to 0x1 then one or more extension headers follows the BNEP header. If the extension flag is equal to 0x0 then the BNEP payload follows the BNEP header [47].

BNEP Packet: Based on the BNEP Type [47].
The following table defines various BENP packet formats.

Value	BNEP Packet Type
0x00	BNEP_GENERAL_ETHERNET
0x01	BNEP_CONTROL
0x02	BNEP_COMPRESSED_ETHERNET
0x03	BNEP_COMPRESSED_ETHERNET_SOURCE_ONLY
0x04	BNEP_COMPRESSED_ETHERNET_DEST_ONLY
0x05 – 0x7E	Reserved for future use
0x7F	Reserved for 802.2 LLC Packets for IEEE 802.15.1 WG

Figure 4.3: BNEP Packet Format

4.2.2 Packet encapsulation

The following diagram illustrates the use of the BNEP for transporting an Ethernet packet. BNEP removes and replaces the Ethernet Header with the BNEP Header. Finally, both the BNEP Header and the Ethernet Payload is encapsulated by L2CAP and is sent over the Bluetooth media.

The maximum payload that BNEP SHALL accept from the higher layer is equal to the negotiated L2CAP MTU (minimum value: 1691), minus 191 bytes (or 187 bytes if an IEEE 802.1Q tag header is present) reserved for BNEP headers. This way it can be assured that enough frame buffer space is reserved to transmit all BNEP.

The minimum payload that BNEP SHALL accepts from the higher layer is zero; BNEP is not required to pad payloads to the Ethernet minimum size (46 bytes) [40] [47].

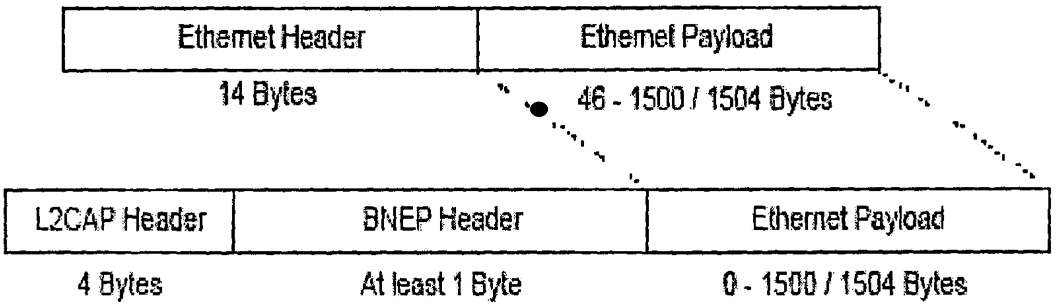


Figure 4.4 Packet Encapsulation

The following is a simple example in which an IP packet is sent using BNEP. The example illustrates an IPv4 packet sent from a device with 48 bit IEEE address of 00: AA: 00:55:44:33 to a 48 bit Bluetooth address of 00:30:B7:45:67:89.

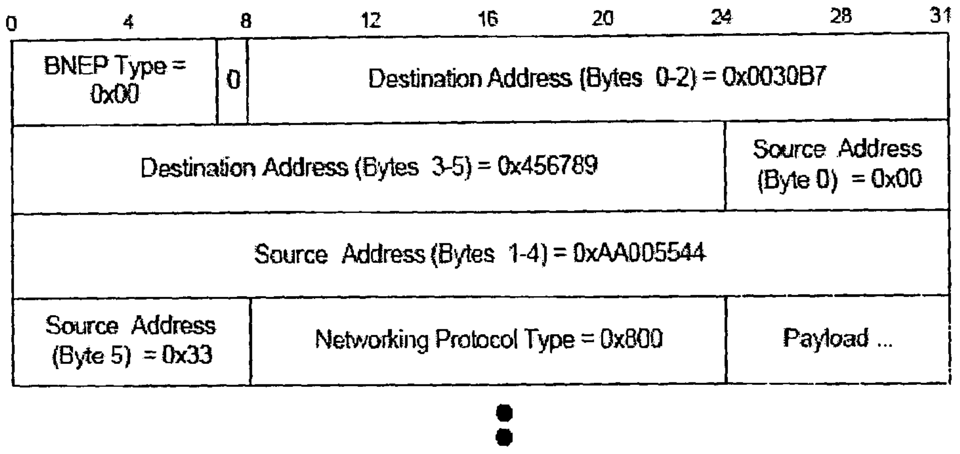
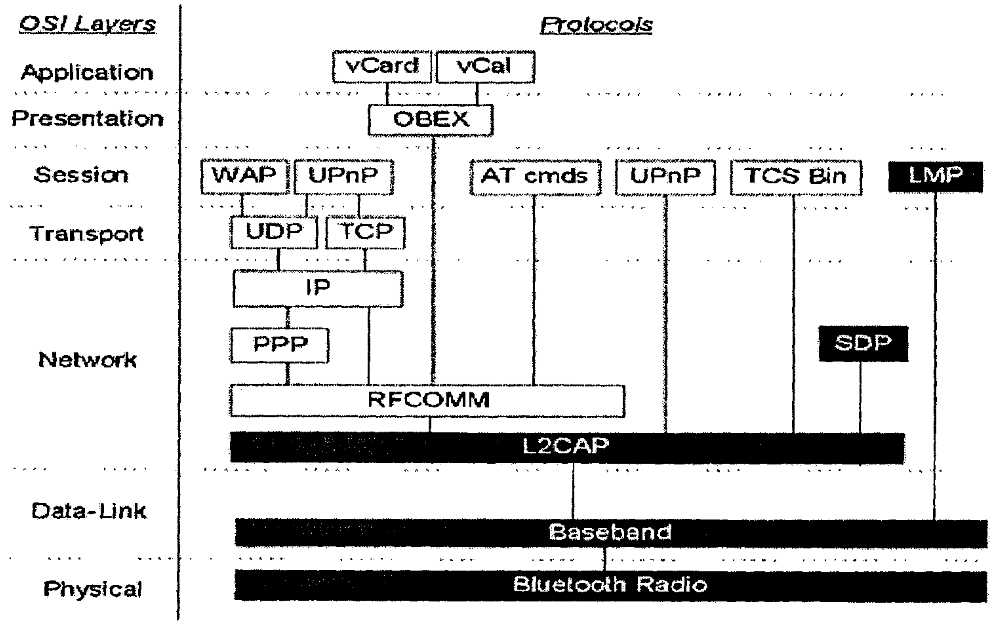

0
4
8
12
16
20
24
28
31
BNEP Type = 0x00
0
Destination Address (Bytes 0-2) = 0x0030B7
Destination Address (Bytes 3-5) = 0x456789
Source Address (Byte 0) = 0x00
Source Address (Bytes 1-4) = 0xAA005544
Source Address (Byte 5) = 0x33
Networking Protocol Type = 0x800
Payload ...

Figure 4.5 IP Packet in BNEP

4.3 Suggested Solution

The suggested idea is that authentication and encryption in Bluetooth to be provided on IP or application level by using IPSec [46] at the IP level. A protocol like IPSec is most suitable to secure end-to-end IP services like Virtual Private Network (VPN) services. IPSec can be used for any IP connection independent of the particular access method. Here only LAN access using the Bluetooth wireless technology is considered. It is important to notice that the use of link level security and VPN solutions does not exclude each other but rather complement each other.

IPSec, however, can protect any protocol running above IP and any medium which IP runs over. More to the point, it can protect a mixture of application protocols running over a complex combination of media. This is the normal situation for Internet communication; IPSec is the only general solution. The following diagram illustrates layers architecture which makes an understanding between Bluetooth and IPSec in Network layer.



Bluetooth Protocol Stack

The problems raises is that Bluetooth enabled devices will have the ability to form networks and exchange information. For these devices to interoperate and exchange information, a common packet format needs to be defined to encapsulate layer 3 network protocols.

Due to that, a specific packet format used to transport common networking protocols over the Bluetooth media [41] [42] [43]. The packet format is based on Ethernet/DIX Framing as defined by IEEE 802.3[44] [45].

BNEP accommodates IP communication by transporting IP packets between two Ethernet-based link layer end-points on an IP segment. It encapsulates the IP packets in BNEP headers, letting the source and destination addresses reflect the Bluetooth end-points and setting the 6-bit Networking Protocol Type field to code for an IP packet in the payload. BNEP finally encapsulates the BNEP packet in an L2CAP header and sends it over the L2CAP connection.

4.3.1 AH and BNEP

This following diagrams illustrates BNEP with an IPv4 packet payload sent using L2CAP before and after positioning AH header for transport and tunnel modes.

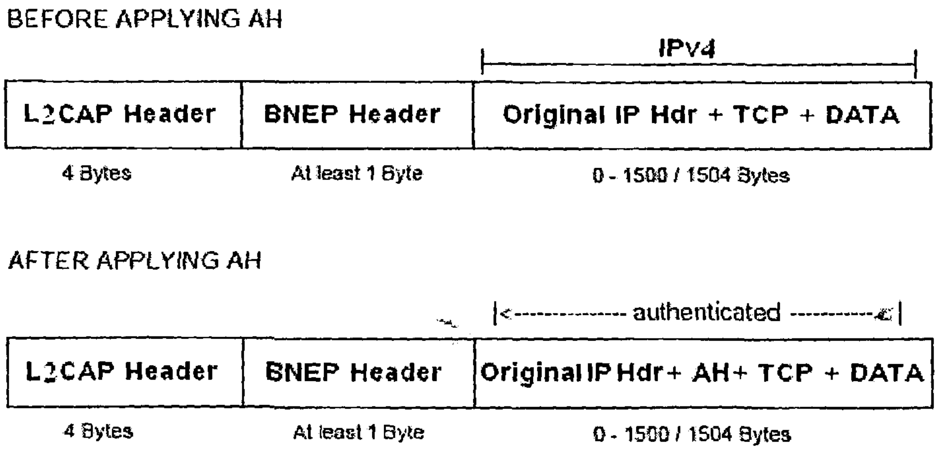


Figure 4.6 BNEP before and after applying AH (Transport Mode)

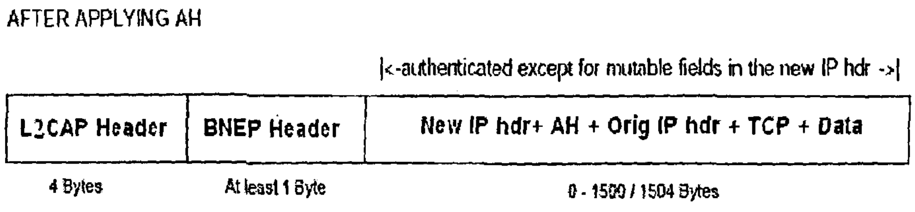

AFTER APPLYING AH
|<-authenticated except for mutable fields in the new IP hdr ->|
L2CAP Header
BNEP Header
New IP hdr+ AH + Orig IP hdr + TCP + Data
4 Bytes
At least 1 Byte
0 - 1500 / 1504 Bytes

Figure 4.7 BNEP after applying AH (Tunnel Mode)

4.3.2 ESP and BNEP

This following diagrams illustrates BNEP with an IPv4 packet payload sent using L2CAP before and after positioning ESP header for transport and tunnel modes.

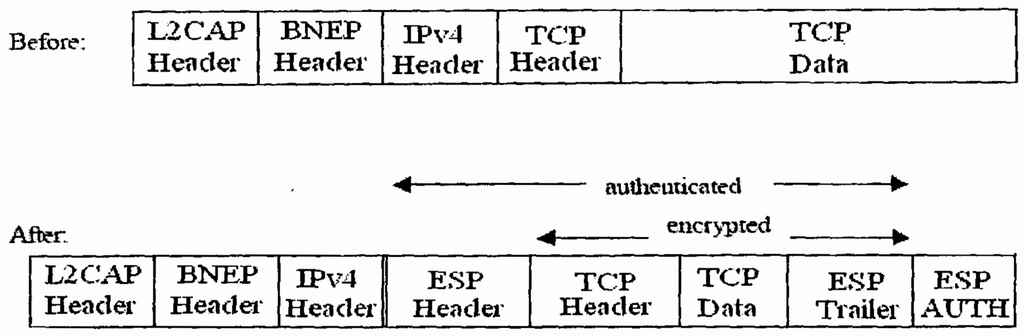


Figure 4.8 BNEP before and after applying ESP (Transport Mode)

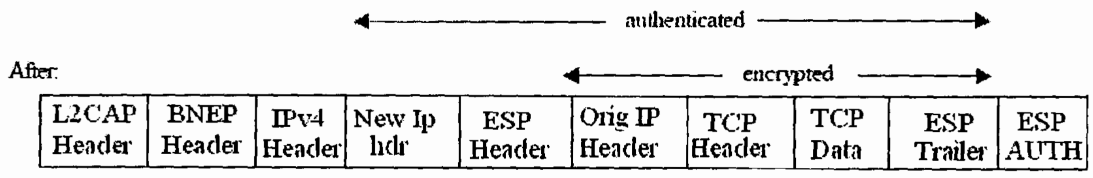


Figure 4.9 BNEP after applying ESP (Tunnel Mode)

4.4 System Design

The simulation of the IPSec protocols in NS2 was based on the existing implementation of wireless network NS-2 [48] version 2 and UCBT (Bluetooth extension for NS2). UCBT implements a full Bluetooth stack, including Baseband, LMP, L2CAP, BNEP layers [49].

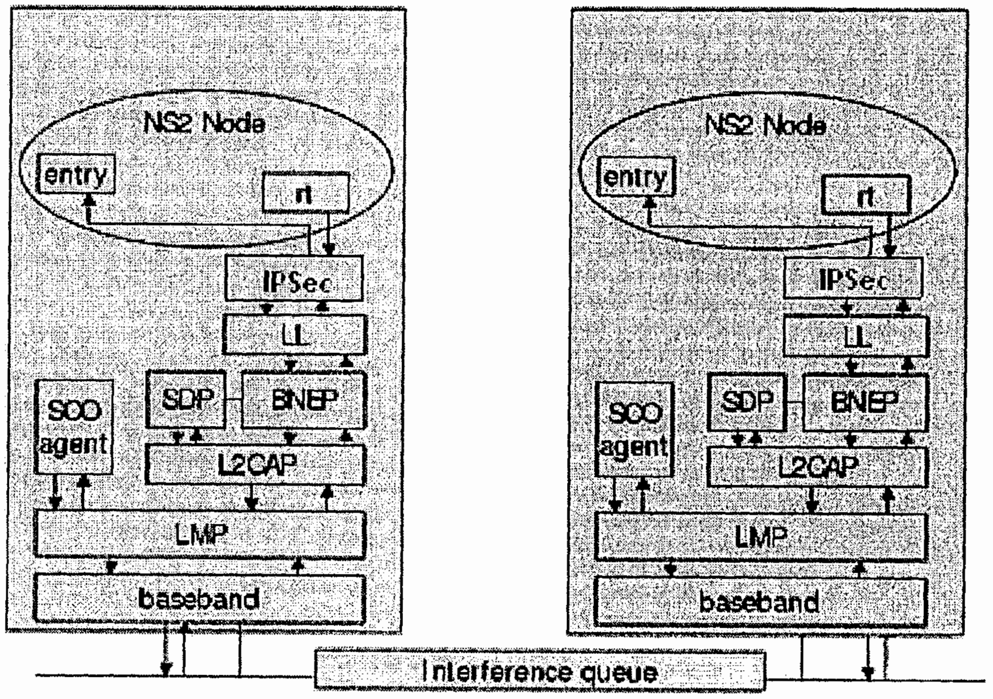


Figure 4.10 IPSec Functionality in NS-2 and over UCBT

Since there are many security protocols in terms of algorithms in IPSec, we had to choose appropriate algorithms. For this purpose we took into account several aspects:

a) Existing documentation of simulations that expressed the time to compute the cryptographic functions involved in the algorithm.
b) Low computational time of cryptographic functions.
c) Algorithms that had proven enough reliability.
d) Algorithms that comply with the basic requirements of the protocols.

For the reasons stated above, we decided on:

- HMAC - MD5 and HMAC - SHA1 to provide origin authentication and integrity for IP packets. MD5 should be preferred because its performance is much better than that of SHA1.
- In ESP implementation we support both encryption and authentication. Encryption is done by the widely used 3DES algorithm, which is applied in CBC mode. Pure DES is also implemented. For authentication we use HASH-MAC MD5.

The IPSec Module is the central part, which does the whole standard conform processing of the incoming and outgoing IP traffic. It uses a set of data bases (SPD and SAD) to determine the flow of the IP packets. The main processing is then done in the AH and ESP module. A small cryptographic library contains all the functionality used to encrypt, decrypt or to authenticate the packets.

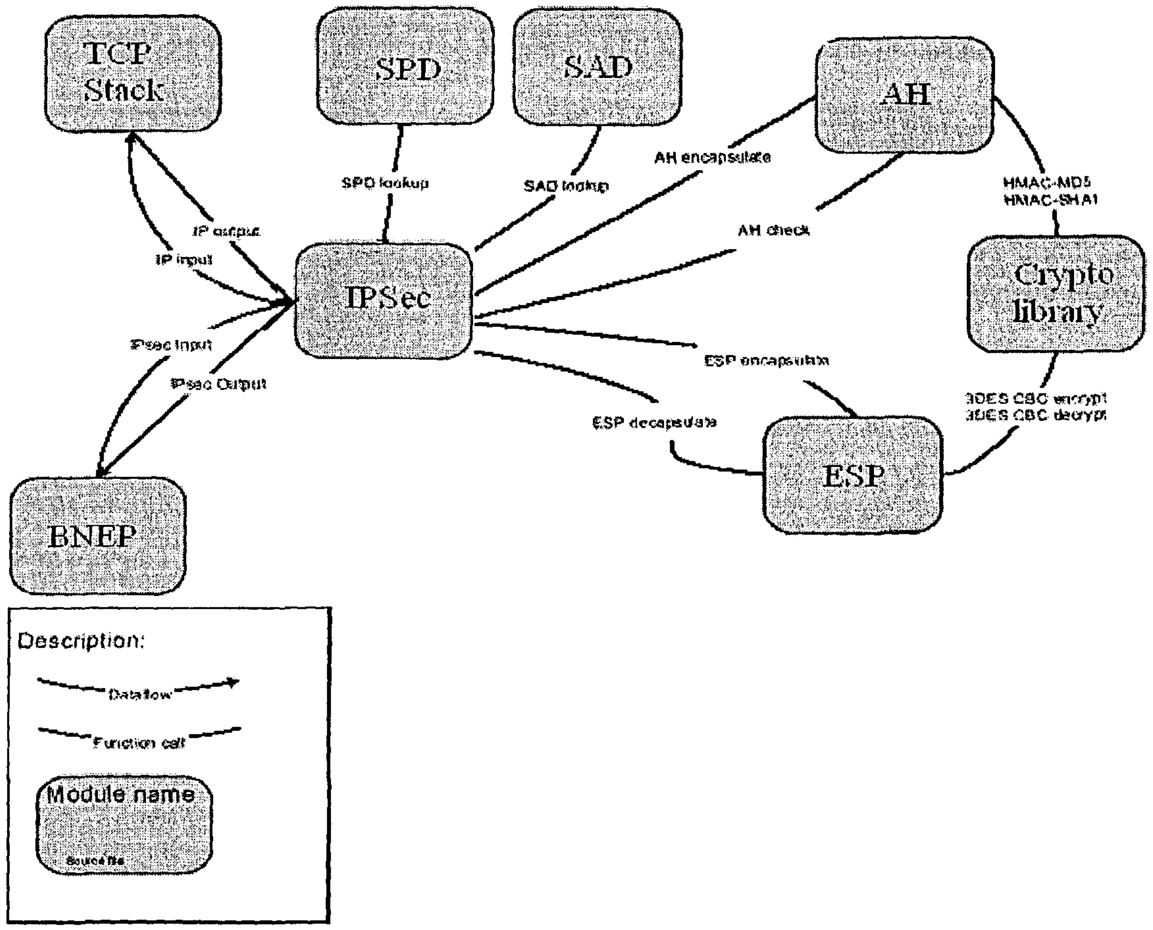


Figure 4.11 The whole IPSec system with dependencies

It was necessary to get familiar with IPSec. This implied reading through various RFCs since there exists no single document covering the whole standard. The "Big Book of IPSec RFCs" *[LOSH02]* is a collection of the relevant documents. The key aspects of a basic IPSec implementation include *[RFC]*:

1. RFC 2401: Security Architecture for the Internet Protocol (IPSec)
2. RFC 2402: IP Authentication Header (AH)
3. RFC 2406: IP Encapsulating Security Payload (ESP)
4. Various RFCs: Algorithm descriptions for MD5, SHA1 and DES

More sophisticated parts of IPSec such as the Internet Key Exchange (IKE) Protocol, the OAKLEY Key Determination Protocol or the Internet Security Association and Key Management Protocol (ISAKMP) where not discussed in this research work.

After having become familiar with the IPSec standard, a breakdown of the whole system was necessary. We needed to identify the different modules out of the IPSec architecture so that we were able to characterize the following attributes of the modules:

- Priority
- Dependencies
- Performance sensibility

An important part of our semester work was to find a suitable IP-stack and Bluetooth stack that is able to carry our IPSec implementation. The Network Simulator NS-2 TCP/IP Stack has all the desired features: modular design, active community and free BSD-style license. As well as UCBT which has all the desired features needed for Bluetooth stack.

Any inbound data is forwarded to ipsecdev_input() function. Depending on the protocol field in the packet header, the entire packet is forwarded to the IP protocol stack. If the packet could be identified as belonging to the suit of IPSec protocols, it is transferred to the IPSec library. Pure IPSec specific processing, such as applying ESP de-/encapsulation or AH de-/encapsulation is done within the IPSec library.

After these steps, the original IP packet is rebuilt by applying new offsets and packet length to the pbuf structure. Then the clear-text packet is passed up to the ip_input() function.

For outbound packets, all IP based protocols forward their data to ipsecdev_output(). Here the decision is made whether the packet needs IPSec processing or not. Depending on the appropriate Security Association, AH or ESP functionality will encapsulate the packet. After these steps, the packet is forwarded to the BNEP Class of Bluetooth Stack and sent over to the receiver.

The Security Policy Database (SPD) can be accessed from the IPSec module. This database contains all rules required to decide how to handle packets, which have security associations but also how to handle non-IP traffic. There are several possibilities: any non-IPSec packet can be forwarded to the default protocol handler (in order for connections from non-IPSec clients are accepted) or any non-IPSec packet can be dropped immediately without wasting CPU time on further analysis.

4.4.1 Security Policy & Association Management

IPSec needs one database to control the flow of the IP packets. This database is called Security Policy Database. It simply describes which traffic requires IPSec processing and which traffic does not.

The other database, the Security Association Database, holds data about each configured connection and also defines how the traffic must be processed if the policy in the SPD defines the APPLY rule for a certain packet.

The SPD can be seen as a persistent database while the SAD is only temporary for each connection. In our simplified environment the SAD could also be static because a dynamic standard conform way to add SA's is not implemented.

4.4.1.1 Basic Concept of Security Association

IPSec needs the Security Policy Database and the Security Association Database to process packets correctly.

The SPD defines the packets, to which IPSec needs to be applied. To guarantee that each packet is processed the right way, each IP packet leaving or entering the system must be checked against the SPD. We call this action the SPD lookup. This lookup does nothing except compare the selectors from the database with the ones from the packet. The SPD lookup delivers back the following results:

- BYPASS: this packet is forwarded directly to the next protocol layer without applying IPSec.
- DISCARD: this packet is discarded, it will be dropped.
- APPLY: this packet requires IPSec processing

If the result of a SPD lookup is BYPASS, the unmodified packet is forwarded to the next protocol layer. This is particularly useful if certain protocols such as ICMP should not be protected by IPSec or communication with non-IPSec hosts must be concurrently possible.

The DISCARD rule is returned when the intention is not to process this packet. If this is the case, the packet will be dropped. This means that we simply delete the packet instead of passing it to the next protocol layer. It is possible to use this feature to build a primitive firewall.

IPSec processing is only needed if the result of the SPD lookup is APPLY. Whenever a packet matches an SPD entry whose policy says APPLY, then there must also be an SA that describes exactly how the packet has to be processed.
A successful SPD lookup provides us with a pointer to the SP over which we can access the SA using a pointer stored in the SP structure.

In a dynamic environment this SA can be created using IKE. As soon as the SPD finds out that there is no current SA available, it will trigger an IKE function which is responsible for the negotiation of the required parameters. The packet can be processed only after Security Association parameters are successfully negotiated.

In a more static environment, where IKE functionality is missing, an SA cannot be set-up on the fly. In such a case, the SA needs to be created at system start-up so that IPSec is ready to process traffic.

4.4.1.2 SPD Outbound Processing

1. When a packet leaves TCP/IP stack, the very first step is an SPD lookup, a determination of how the packet must be processed. When the policy says APPLY, the IPSec process continues. Otherwise the function passes the packet to the Bluetooth stack or returns to the TCP/IP stack without doing anything.
2. Now (in case of an APPLY policy) the packet must be processed according to the SA that was given back by the SPD lookup. When no SA is available, IKE functionality would be invoked. If no IKE is available or the IKE negotiation fails, the packet must be discarded.
3. In case of a valid SA being available the packet is encapsulated either in an AH- or ESP-header.
4. After the new IPSec packet has been built, it must be sent out on the Bluetooth stack.

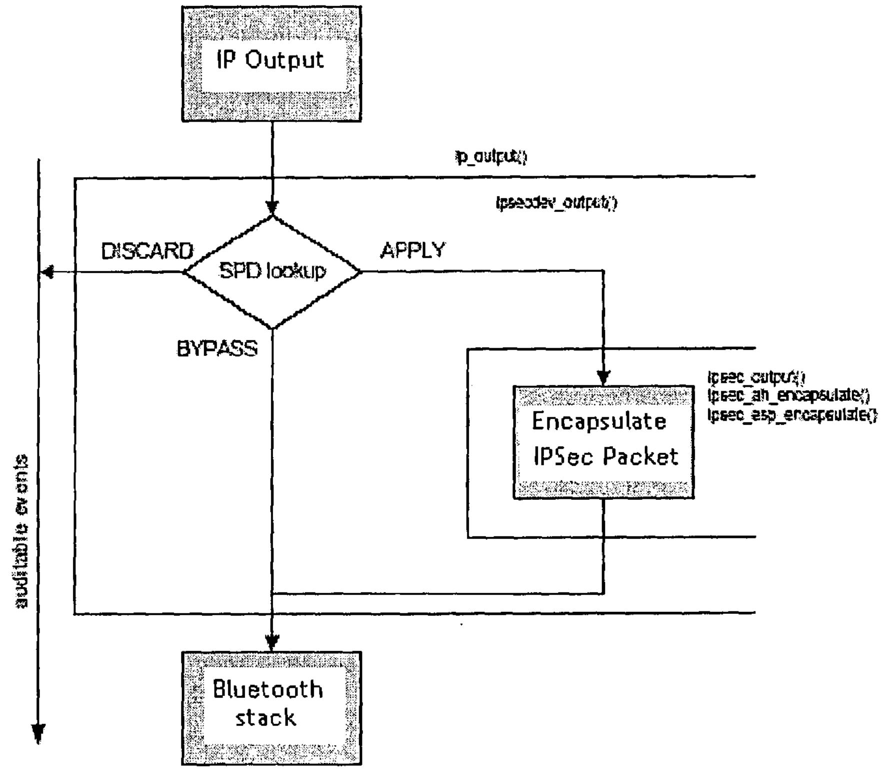


Figure 4.12 Outbound Processing

4.4.1.3 SPD/SAD Inbound Processing

1. Inbound processing is somewhat different because an incoming IPSec packet already has an SPI, which allows a direct lookup in the SAD table. The reason for using the SPI is straightforward. The incoming IPSec packet may be encrypted and so the SPD lookup, which must be performed on the inner packet data, cannot be performed. The SAD lookup would directly give back an SA if one was found. If no SA is found, then the packet must be discarded.
2. With the valid SA we are now able to process the packet properly. In inbound processing this corresponds to decapsulation in ESP or integrity checking in AH.

3. After decapsulation, we have a clear-text or authenticated packet. To be sure that the right SA was applied to the packet, an SPD lookup has to be performed now on the clear text packet. This check will confirm that there was a valid SPD entry for the SA, which was used. This must be done because a packet could have been sent with a fake SPI to force the proper processing of the packet. If the SPD lookup fails or points to a different SA, the packet must be dropped.
4. After the IPSec packet has been decapsulated, it can be passed on to the TCP/IP.

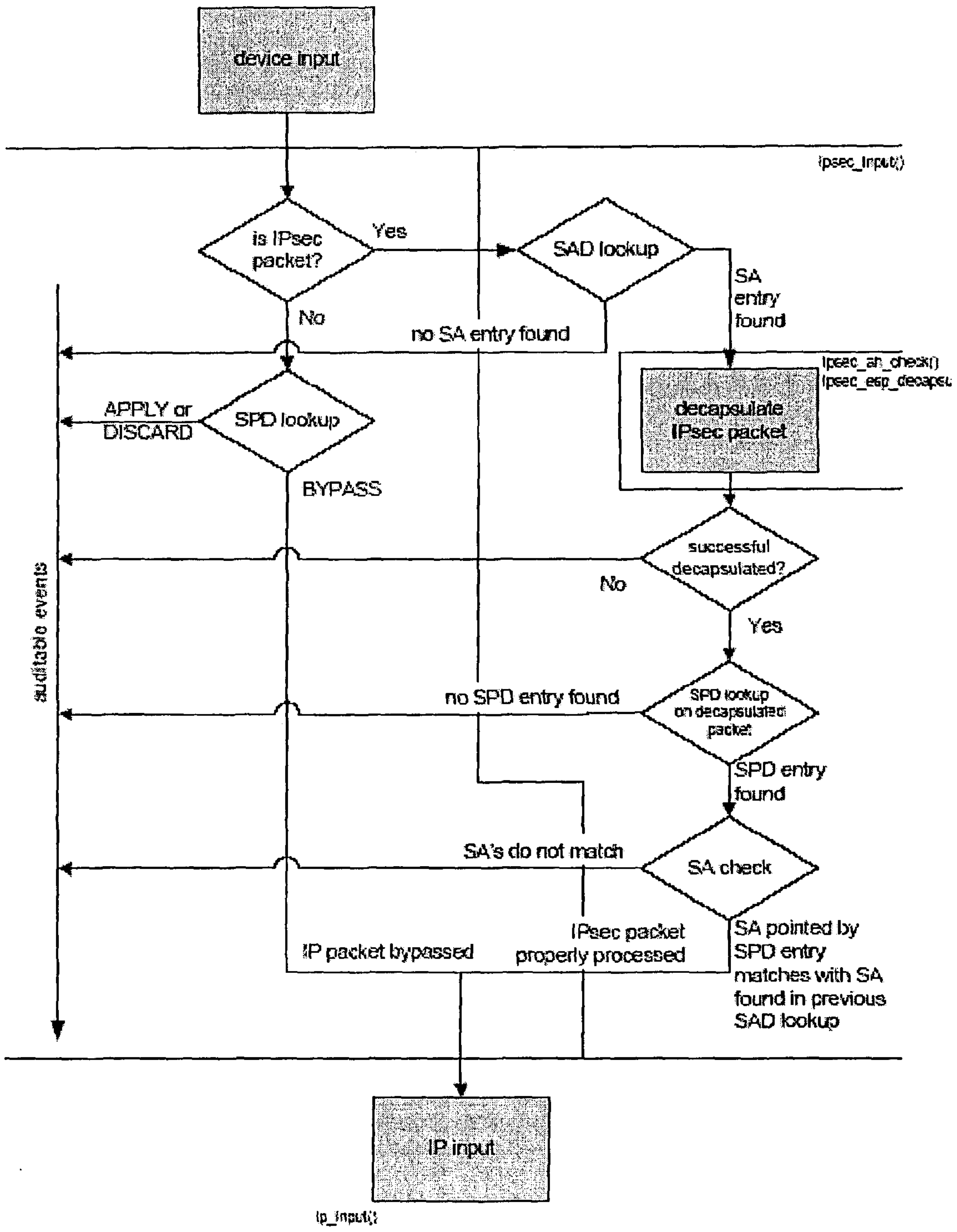


Figure 4.13 Inbound Processing

4.4.2 AH Processing

The IP Authentication Header (AH) provides data origin authentication and integrity for IP packets. Authentication is done by the well-known HASH-MAC4 MD5 and HASH-MAC SHA1algorithms. These are the algorithms requested by the standard. MD5 should be preferred because its performance is much better than that of SHA1.

In the SA configuration, the user has the possibility to choose the algorithm that shall be selected for authentication. Because the integrity of an AH packet can always be verified, the anti-replay check is performed on each packet. If one considers that ESP also supports integrity and authentication, one may think that there is no need for AH. This is not true because the authentication and integrity check of AH is a bit more sophisticated. Authentication in AH covers more fields of the packet than ESP does.

AH processing can be split up into inbound and outbound processing. These two parts are implemented in:

- **ipsec_ah_check():** verifies the integrity of the AH packet by applying a HMAC with given key and performs an anti-replay check.
- **ispec_ah_encapsulate()**: sets up a new AH and IP header in front of the inner packet and calculates its integrity check value. The next two paragraphs describe more detailed how AH processing was implemented.

4.4.2.1 AH Inbound Processing

AH inbound processing was implemented with the function ipsec_ah_check(). This function gets the following input parameters:

- Pointer to the IP packet that must be verified
- Pointer to the SA that describes how the packet must be processed

After AH processing is done, two variables are passed back:

- Offset to the decapsulated IP packet (relative to the address of the input IP packet)
- Length of the inner IP packet

The processing itself described step-by-step

1. In order to check the integrity and the authentication of the packet, the ICV must be calculated. The ICV calculation in AH also covers the outer IP header. In this header there are so-called mutable fields, which change their value while they are sent across the network. Those fields (Type of Service, Offset, TTL and checksum) must first be set to zero. The ICV fields in the AH header must be backed up and zeroed, so that later comparing remains possible. It becomes clear that AH authentication also covers the source and destination address of the outer IP packet.
2. The packet is now ready to be verified and the integrity check value can be calculated over the whole packet. The SA determines the appropriate algorithm and key.
3. The calculated ICV can be compared with the one saved in the first step. Processing continues only if the calculated ICV matches the original one.

4. The authentication of the packet is now verified and the anti-replay check can be performed. If it is successful, the sequence number (stored in the SA) is incremented. Finally, the offset and packet length are passed back.

4.4.2.2 AH Outbound Processing

AH outbound processing was implemented with the ipsec_ah_encapsulate() function. This function gets the following parameters:

- Pointer to the IP packet, which must be encapsulated.
- Pointer to the SA, which defines how the packet must be encapsulated.
- Source IP address, describing the tunnels source address
- Destination IP address, describing the tunnels destination address

After AH processing has been completed, two variables are passed back:

- Offset to the encapsulated IP packet (relative to the address of the input packet)
- Length of the encapsulated IP packet

The processing itself described step-by-step:

1. First of all a new AH header is placed in front of the IP packet, leaving a gap between the inner IP header and the AH header. This gap is later used to place the ICV. The AH header fields: next header, length, SPI and sequence number are added.
2. After the outer IP header has been constructed, only the source and destination address, version, header length and total length are set. The other fields are set to zero as a preparation for the ICV calculation. Padding is not required because the packet is already aligned.
3. The integrity check value can now be calculated and placed into the gab between AH header and inner IP header.
4. After the ICV has been calculated, the zeroed fields are now filled with the appropriate values.
5. Finally, the offset and the packet length are passed back.

4.4.3 ESP Processing

An Encapsulating Security Payload (ESP) header is designed to provide a mix of security services for IP packets. In our ESP implementation we support both encryption and authentication. Encryption is done by the widely used 3DES algorithm, which is applied in CBC mode. Pure DES is also implemented. For authentication we use HASH-MAC MD5 and HASH-MAC SHA1.

With the SA configuration the user has the possibility to configure the security features that are to be applied to ESP processing.

When the user also selects authentication, the anti-replay service can guarantee that resent IP packets or packets entering the system out of the replay-window are discard.

ESP processing can be split up into inbound and outbound processing. These two parts are implemented with

- **ipsec_esp_decapsulate():** checks the content of the ESP header, and optionally verifies the authentication and anti-replay and decrypts the packet with the given key and initialization vector.

- **ipsec_esp_encapsulate():** sets up a new ESP header, encrypts the packet and optionally calculates the integrity check value (for authentication).

Both functions also setup the outer IP-header, which is needed for tunnel mode.

4.4.3.1 ESP Inbound Processing

ESP inbound processing is implemented with the function ipsec_esp_decapsulate().

This function receives the following input parameters:

- Pointer to the IP packet which must be decapsulated
- Pointer to the SA which describes how the packet must be processed.

After ESP processing has been done, two variables are passed back:

- Offset to the decapsulated IP packet (relative to the address of the input IP packet).
- Length of the decapsulated IP packet.

The processing it self described step-by-step:

1. A check in the SA structure indicates whether authentication needs to be checked or not. If an authentication algorithm is specified within the SA, the ICV must be calculated and compared with the one stored at the end of the ESP packet. The ICV is calculated over the whole ESP header, IV and encrypted payload. Processing continues only when the packets ICV matches our recalculated one.
2. In the next step we have to decrypt the packet. The decryption algorithm and the secret key can be accessed over the SA. Because the packet was encrypted in CBC-mode, the IV must be copied out of the ESP packet. The IV is stored between ESP header and encrypted payload. The decryption happens in-place, so no copying must be done.
3. Since the IP packet has now been extracted out of the ESP packet, we can perform some sanity checks before terminating ESP processing. In our implementation we verify that the total length field in the extracted IP packet is within our valid range (20-1500 bytes).
4. Before everything is done the sequence number counter in the SA is incremented, and optionally the same is done with the anti-replay window. To let the caller of the ESP function know about the location and the size of the extracted IP packet, the offset and the packet length are giving back.

4.4.3.2 ESP Outbound Processing

ESP outbound processing is implemented with the function ipsec_esp_encapsulate().

This function receives the following input parameters:

- Pointer to the IP packet which must be encapsulated.
- Pointer to the SA, which describes how the packet must be processed.
- Source IP address describing the tunnels source address (from outer IP header).

- Destination IP address, describing the tunnels destination address (from outer IP header).

After ESP processing has been completed, two variables are passed back:

- Offset to the decapsulated IP packet (relative to the address of the input IP packet).
- Length of the decapsulated IP packet.

The processing itself described step-by-step:

1. The first step of encapsulation is to test whether the decremented TTL field of the IP header reaches zero. If this is the case, the packet must be discarded in order to prevent endless straying of packets.
2. Then we have to calculate how much padding must be added to fulfill the requirements of the encryption algorithm. In our case (DES/3DES) we must have the payload aligned to 8 bytes because the block size of DES/3DES is 64-bit (8 bytes). The right amount of padding bytes is added at the end of the payload. The fields: padding length and next header are appended right after the padding.
3. Encryption is now performed according to the settings in the SA. After encryption, the used IV is copied in front of the encrypted payload.
4. ESP header is now added in front of the IV. Inserted are a incremented sequence number and the SPI taken out of the SA.
5. As was done in inbound processing, the SA must be checked to see if authentication is enabled. If this is the case, then the ICV must be calculated according the SA's settings. The ICV, which is calculated the ESP header, the IV and the encrypted payload, is copied at the end of the payload.
6. Now the outer IP header can be constructed using the tunnels source and destination address given as input arguments to the function. The TOS field is copied from the inner IP header.
7. Finally, the offset and the length are passed back, so that the caller can update its data structure (in our case the pbuf), where the packet is stored.

CHAPTER 5

Implementation

5. Implementation

In this chapter we shall discuss the implementation of the system. We shall discuss the technologies used to develop our system and the benefits these technologies. These technologies have greater scope on other related technologies.

5.1 Network Simualtor NS-2

5.1.1 Introduction

Ns began as a variant of the REAL network simulator in 1989 and has evolved substantially over the past few years. In 1995 ns development was supported by DARPA through the VINT project at LBL, Xerox PARC, UCB, and USC/ISI. Currently ns development is supported through DARPA with SAMAN and through NSF with CONSER, both in collaboration with other researchers including ACIRI. Ns has always included substantal contributions from other researchers, including wireless code from the UCB Daedelus and CMU Monarch projects and Sun Microsystems.

Ns is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. **Nam** is a Tcl/TK based animation tool for viewing network simulation traces and real world packet traces. It supports topology layout, packet level animation, and various data inspection tools.

ns-2 is the second major iteration of a discrete-event network simulation platform programmed in C++ and Object Tcl (OTcl).. ns-2 is a major architectural change from ns-1-- the simulator became entirely based on the blend of OTcl and C++.

5.1.2 Functionality

Ns-allinone is a package which contains equired components and some optional components used in running ns. The package contains an "install" script to automatically configure, compile and install these components. After downloading, run the install script. If you haven't installed ns before and want to quickly try ns out, ns-allinone may be easier than getting all the pieces by hand.
Currently the package contains:

- Tcl release 8.4.11 (required component).
- Tk release 8.4.11 (required component).
- Otcl release 1.11 (required component).
- TclCL release 1.17 (required component).
- Ns release 2.29 (required component).
- Nam release 1.11 (optional component).
- Xgraph version 12 (optional component).
- CWeb version 3.4g (optional component).
- SGB version 1.0 (optional component, builds sgblib for all UNIX type platforms).
- Gt-itm gt-itm and sgb2ns 1.1 (optional component).
- Zlib version 1.2.3 (optional, but required should Nam be used).

Currently, ns-allinone works on UNIX systems and under Cygwin for Windows 9x/2000/XP .The current release 2.29; released Oct 22, 2005 can be downloaded from [50]:
http://prdownloads.sourceforge.net/nsnam/ns-allinone-2.29.2.tar.gz?download

5.2 UCBT

5.2.1. Introduction

UCBT (stands for University of Cincinnati - Bluetooth) is an ns-2 based Bluetooth network module which simulates the Bluetooth network operations in great details. Most specifications at Baseband and above like LMP, L2CAP, BNEP have been simulated in UCBT, including frequency hopping scheme, device discovery, connection set up, Hold, Sniff and Park modes management, role switch and multi-slot packet type negotiation, SCO voice connection, etc. There is a provision to constitute a cluster of Bluetooth devices and such formation with up to 8 Bluetooth devices is known as a piconet. It also allows a number of piconets to be connected together using "bridge nodes" and such a large network is usually referred to as a scatternet.

UCBT is not the first ns-2 based Bluetooth simulator. BlueHoc developed at IBM and its scatternet extension, Blueware at MIT, both pre-date UCBT. However, with 28,000+ lines of C++ code:

1. UCBT is the most accurate, complete and up-to-date open-source Bluetooth simulator.
2. It adapts to the PAN profile with Bluetooth Network Encapsulation Protocol (BNEP).
3. It takes clock drift into account, which is very important for simulating synchronization or scheduling protocols accurately, as difference devices will drift apart in long period.
4. It also includes the newly adopted Enhanced Data Rate (EDR) specification to simulate new devices with 2 or 3 Mbps data rate.

One of its main contributions is that UCBT provides a flexible framework to conduct Bluetooth scatternet research. A scatternet requires time sharing of some common devices (bridges) among piconets. Coordination of the presence schedule of bridge nodes in a large mesh scatternet is very challenging. UCBT provides multiple bridge scheduling algorithms to enable scatternets to operate smoothly. Prototype self-organized scatternets are being designed and simulated.

5.2.2. Functionality

As a Bluetooth module for ns-2, UCBT implements a full Bluetooth stack, including Baseband, LMP, L2CAP, BNEP layers. It integrates with ns-2 well and works out of box for recent ns-2 release, ns-2.28. UCBT closely follows spec 1.1 and is partially updated to spec 2.0.

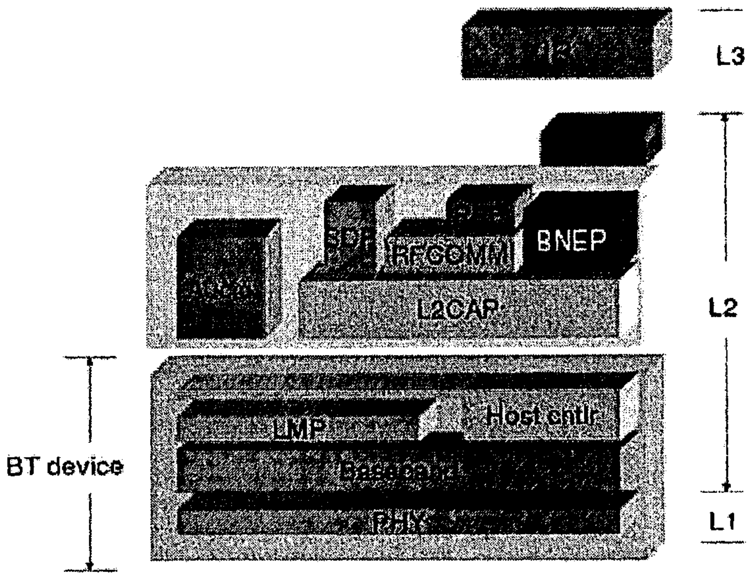


Figure 5.1 Bluetooth stack

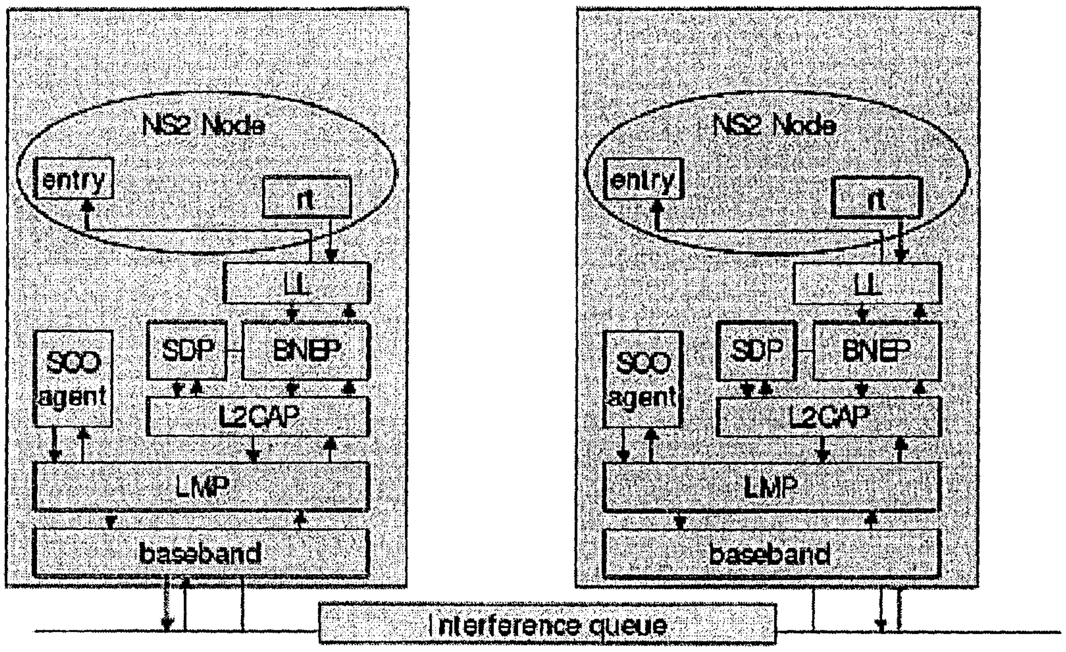


Figure 5.2 UCBT Bluetooth Node

- Radio Channel/PHY: It is not modeled explicitly. It is modeled using a configurable Loss model and Interpiconet interference detection model Baseband.
- Baseband modeling: It has a correct frequency hopping kernel which generates the same sequence as illustrated on the Specs 1.1 (pp963-968). It handles multiple slots packets. It handles SCO. It understands clock drift.
- LMP: Handles Link setup, Role Switch, Link Suspension (Hold, Sniff, and Park) and Piconet Switch.
- L2CAP: Handles SAR and Protocol Multiplexing.

- BNEP: Provides MAC interface to higher layer such as LL.
- SCO: SCO connections are handled by SCO-Agents, which bridge Baseband/LMP and Applications.
- Mobility: Mobility is specified as the wlan node style. That is, let node 'setdest' at specific time to change the direction and speed.

5.2.3 Interface & Implementation:

Bluetooth node is a normal ns node with different MAC/PHY. Currently I view the entire Bluetooth as a new MAC in the ns system. Create a Bluetooth Node using the following interface:

```
Set ns_      [new Simulator]
$ns_ node-config -macType Mac/Bluetooth; # or Mac/BNEP

Set addr 1
Set node0 [$ns_ node $addr]

Set statetime 0.1
$ns_ at $statetime "$node0 on"
```

Most the control interfaces are located in BTNode::command () (bt-node.cc). You should read that method to be familiar with all controls. You can also get a hint about which variables are settable at runtime in file ns-btnode.tcl.

Bluetooth Node has the following exclusive components besides those common to Node:

BNEP:

- MAC interface to upper protocol.

L2CAP:

- Maintains Channels
- mapping connection to L2cap channels
- Mapping Channels to Connection Handle. M: 1 mapping
- SAR

LMP:

- Maintains Bluetooth device information database -- should move up!!
- Maintains piconets
- Mapping Connection Handle to LMP Link (ACL/SCO). 1:1 mapping
- mapping Link to TxBuffer at Baseband

Base Band:

- Page/inquiry/scan, etc.
- Link Scheduler.
- Frequency hopping kernel.
- TxBuffers
- ARQ.

Different from the specs requirement of TDD scheme, UCBT controls the TDD by a schedule word at the Baseband. Say, at current clk, if looking up this word return M, UCBT picks up a package at TxBuffer [M], and transmits it. Obviously, this word can be used to implement easily what the specs says: we let the word return valid TxBuffer slot at Master transmitting slot for the master and at slave transmitting slot for the slave. This sched word is also useful for QoS mapping. Link scheduler is implemented on the top of schedule word, because only tight link (like SCO) need to schedule as specified by the word.

A non-qos ACL link is usually not specified by the word therefore can be scheduled by the Link Scheduler. Another difference is, the master also have a CLK controller, though normally set to CLKN.

Trace format is not fixed by now. It's interfaced by two set of commands: trace-all-xxx and trace-me-xxx. The former has an effect on all BT devices, while the latter only has an effect on the specific node.

Physical layer is not explicitly implemented; the bottom protocol stack is baseband, which are interconnected so each baseband packet will be forwarded to all other basebands.

Different LossMod to model packet loss can be specified, though there is only the table driven module used by BlueHoc project is available. Package collision between different piconets can be detected.

Currently, the baseband is fairly complete. Some minor things like clock wrapping around is not modeled.Park/Unpark is not completed because it's quite complex and less useful. Clock drifting is being implemented and should complete soon.

Spec1.2 features like interlaced scan is implemented. AFH is also in place in baseband but work in LMP is needed. ESCO is not implemented at this moment.

Baseband (BB) can support 2 piconet parameters concurrently: an active piconet and a possible SCO link of other piconet. This doesn't mean that BB has two sets of transmitter. BB just switches between the 2 piconet parameter before and after the SCO slots. This happens at the slot granularity. An lmp scheduled piconet switch would be much less frequent.

LMP can handle as many sniffing, held, and parked piconets as you need, by taking suitable scheduling scheme. Two HV3 SCO links from different piconets can be supported at the same time. This is also the maximum capacity regarding 2 SCO piconets, since CLK is not aligned.

SCO traffic is handled separated by SCO agents, which connect Application/ Traffic generator and LMP/Baseband. L2CAP doesn't handle SCO, as the specs specified.

CHAPTER 6

Testing

6. Testing

System testing is an essential step for the development of a reliable and error-free system. Testing is the process of executing a program with the explicit intention of finding errors i.e., making the program fail and test cases are devised with the purpose in mind. A test case is a set of data items that the system processes as normal input. A successful test is the one that finds an error.

This chapter explains our test framework that enabled us to test early and often. We continuously controlled our work in order to be able to provide good quality of coding. Our test framework consists of three main parts. They are all described in the next paragraphs.

The basic strategies that were used for testing were following

6.1 Structural (White Box) Testing

6.1.1 Why Structural Testing?

This type of testing can be done early in the project phase. It involves testing of a single unit of software. In our case these units are our modules. The tests should check all the important elements of the module. We were able to add all structural tests to one unit so all the structural tests can be performed in a batch process (all tests at the same time). This enabled us to run the batch often and regularly. The tests can also be run after each small change in the code, so we always knew that our software was still working in a consistent state. Structural testing is also called white box testing because the exact kind of functions and code tested is known.

6.1.2 How We Implemented Structural Testing

Structural testing means running a test for each implemented function. We decided to apply the extreme Programming paradigm "test driven development". This forced us to write down testing procedures first. After the test procedures were implemented, we were able to start coding the actual problem. Test-driven development helped us to: - First think about what the tested function really needs to do. - Define the input and output data types and value ranges. - Be able to run a test after having finished coding the function.

We wanted to go even a bit further. We first wrote down all the test functions needed to test the whole IPSec implementation. This helped us thinking about the whole programming structure of our end product. The test functions were added to our main test routine. At first, all these test functions were just stubs, without any implemented code. They were then implemented with a predefined interface, so that they could easily be called from the main test routine. The test routine of a certain module was responsible for the testing of the whole structure of the module.

The module test function needed to implement the following features:

- Implementation with a predefined interface and naming convention.
    - Interface: int function(void)
    - Name: modulename_test()
- When the test function is returned without an error, we can be sure that the module is implemented properly and without bugs.

- It is able to do many tests of the same function if required. Input data may be random (i.e. for value range testing) or statically programmed.
- It is able to call many different functions of the module.
- The tests need to be reproducible.
- As long the test function is not implemented, it must print out that the module / module test function has not yet been implemented.

When the module test functions were implemented according to these rules we were able to run the module test function out of one main (executable) function. The first time we ran the test function we got a list of printed messages. Each line corresponded to one module test function and it showed that the module had not yet been implemented. Our goal for the end of the project was for each module test function to print a line indicating that all tests were successful.

6.2 Functional (Black Box) Testing

6.2.1 Why Functional Testing?

The second step in our testing concept was functional testing. In this phase we tested the functionality of our IPSec implementation. The reason why it is also called black box testing is because we didn't care how the tested functionality was implemented and what functions are needed to implement this functionality. It was a kind of abstraction: we just wanted to know whether certain functionality did its job properly. This kind of test is not based on modules but on functionality. As an example we wanted to know if an AH packet can be verified properly. In order to be able to run such a test we needed functions of many modules (HASH-MAC, AH, etc...).

Functional testing can always be started with small tests. The more modules were implemented, the more functional tests could be done. The objective of the test was to check whether the modules work properly with each other. The further the project preceded the more complex the tests became. The last functional test was a test that checked the whole function of our IPSec implementation. After having run the functional test properly we expected to be able to run the implementation on the desired hardware in the real world.

For functional testing we used similar rules that were needed for structural testing. Failing functions need to print out the input, output and expected output data in order for the error to be reproducible.

As wanted side-effect, each functional test performs many different tests on the various involved modules.

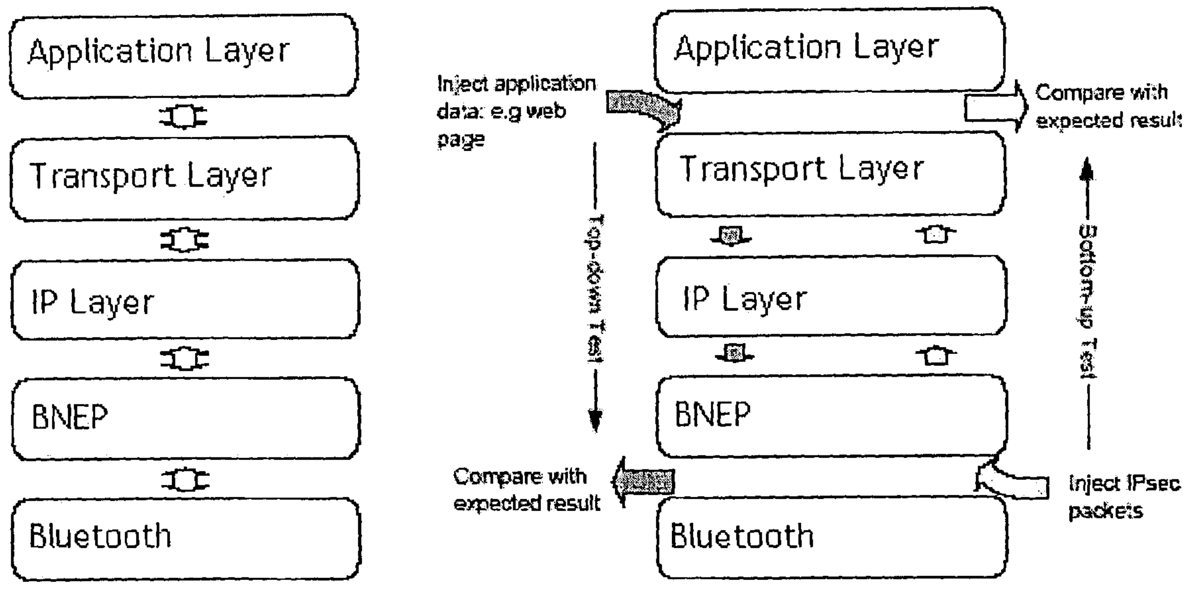


Figure 6.1 Normal dataflow and dataflow of functional tests.

6.2.2 How We Implemented Structural Testing

Since functional testing was implemented as the project proceeded, we did not specify the structure of functional tests. Because functional testing demanded a lot of memory for static test data, we were not able to make one executable. We split up the functional tests into functional units.

There were two possible approaches: Top-Down and Bottom-Up tests (Figure 6.1).Both could be implemented. They were independent to each other and they tested the functionality in different ways.

In our IPSec implementation there are two directions of data flow. One direction is from the network device upwards the stack, until data reaches the application. The other direction is downwards the stack, starting at a normal TCP or UDP packet which gets encapsulated and injected into the Ethernet.

Top-Down Tests

Top-down starts at the top of the stack and goes down the whole stack. The following is an example of what a top-down test could look like:

- Lets assume that there is a function called esp_encapsulate (char *data, char *esp_packet). This function creates an ESP packet out of some user data (e.g. TCP packet).
- We capture an IPSec stream from two communicating IPSec peers. They use an already working IPSec implementation like FreeS/WAN or IPSec_tunnel. We know the keys and algorithms from the configuration and we know the plain text data of the packet, because a well-known sequence of packets has already been captured.
- We can now feed the esp_encapsulate () function with the known plain text data. This could for example be the content of a small web page packed into a TCP packet.

- The function must now return an ESP packet looking equal or very similar (maybe there are fields which may be different i.e. sequence numbers, etc) to the original packet we captured from the IPSec stream. With such a test we were able to ensure that the whole functionality of generating an ESP packet was implemented properly. Of course we still didn't know whether it was possible to check if we were able to verify and decapsulate an ESP packet. This was done in the Bottom-Up tests. With such a test we can check a part of the functionality of our implementation. Another test could have done the same for AH packets.

Bottom-Up Tests

This test starts at the bottom of the stack. The last piece of software right before the hardware is the network interface driver. A bottom-up test could look as follows:

- From an IPSec stream we capture one or multiple IPSec packets.
- From the configuration we know all the parameters.
- We now feeds the appropriate IPSec function with a captured IPSec packet through BNEP protocol.
- We can now see if our implementation is able to find the appropriate SAD (Security Association Database) entry for this packet. We are able to check if the implementation is able to find out if the packet needs to be processed by IPSec or not.

6.3 Interoperability Testing

A useful implementation must be able to talk with other IPSec products. Interoperability means that our implementation should be able to establish IPSec tunnels with other peers in the network. First of all we analyzed different IPSec products, which we could use for the interoperability tests. We had to choose a product that is easily available and suits as a good reference. We had to take a product that was already widely used and thus well tested. We soon focused on the following IPSec implementations:

- IPSec_tunnel: a simple and free implementation for Linux
- FreeS/WAN: a free Linux IPSec implementation.
- Windows IPSec: on a Windows 2000 operating system.
- PGPNet: another free IPSec implementation for Windows and Mac
  The next paragraph describes the products we looked at more closely.

6.3.1 Testing Environments

6.3.1.1 IPSec_tunnel for Linux

The IPSec_tunnel developed by Tobias Rainstorm [TRS] is an elegant, minimal implementation of IPSec tunnel functionality for the Linux 2.4 kernel series. It is designed as kernel module and the IPSec_tunnel module itself does not require any kernel patching. The only requirement is the presence of the Crypto API code in the kernel source (some kernels are shipped with it already; others require the addition of strong cryptography by applying the International Kernel Patch). After successfully having loaded the IPSec_tunnel module, an unconfigured network device "IPSec0" becomes available. The "IPSecadm" utility is used to create, list and modify SA records for IPSec_tunnel. Selecting a previously defined Security Association to create a tunnel is the final step in configuring IPSec_tunnel. The standard Linux network tools "ifconfig" and "route" can be used to establish a route through the IPSec0 device. Since IPSec_tunnel relies on the encryption and digest functionality provided by Crypto API (Figure 6.2), all common algorithms are supported:

Ciphers	Digests
3DES, DES, AES, Blowfish, Twofish, Cast5, DFC, IDEA, Mars, RC5, RC6, Serpent, null	MD5, RIPEDEM160, SHA1, SHA256, SHA384, SHA512

Figure 6.2 Common Algorithms

The supported protocols of IPSec_tunnel are ESP and ESP with authentication. Support for dynamic configuration is NOT available.

As long as manual keying is supported by the other party, IPSec_tunnel looks promising regarding to interoperability. There are already successful interactions with FreeS/WAN and Open BSD documented.

6.3.1.2 FreeS/WAN

FreeS/WAN was interesting for us from the beginning because it is widely used in the Linux community. There is a lot of documentation available and in case of problems or additional interest we would be able to have a look at the source code, which of course is freely available. FreeS/WAN also supports many features, which we wanted to, implement in our IPSec library.

With FreeS/WAN we would be able to perform the following tests:

- AH tunnels.
- ESP tunnels.
- Encryption with 3DES.
- Authentication with MD5.
- Authentication with SHA1.
- Manual Keying.

Manual keying was a very important feature for us because we were not yet sure whether we would be able to implement automatic keying. Besides this, manual keying simplifies debugging because it's independent from complicated key generation and negotiation.

By default, FreeS/WAN is a very restrictive implementation and so the developers do not like to implement features that decrease the security. The following features can only be added using patches:

- Single DES encryption
- Null encryption

We did not want to lose time with patching the FreeS/WAN source and recompilation of kernels. The supported features were rich enough to provide a good test environment. FreeS/WAN also provides good debugging facilities. This helped us to analyze problems and observe what happened in the IPSec kernel. Unfortunately, documentation is poor and bad structured so that getting starting with FreeS/WAN turned out to be a time consuming task.

6.3.1.3 Windows 2000

We also had a look at the Windows IPSec implementation. Of course Windows would be a spectacular testing environment because of its wide spreading. Windows 2000 and Windows XP have built-in IPSec support. Soon we found out that manual keying is not a possible configuration. The advantages of Windows IPSec would be: - Support of Null encryption (good for debugging) - Support of single DES encryption (good for performance) Because we were not able to find out whether and how manual keying can be done with windows, we had to omit this implementation for our tests.

6.3.1.4 PGPnet

PGPnet is an easy to install and configure VPN software for Windows and Macintosh computers. It implements the IPSec and IKE protocols and supports OpenPGP keys for authentication in addition to X.509. The tested version 7.0.3 cannot handle manual keying and therefore cannot be used to test the simple IPSec configuration.

6.3.2 Security Tests

This section describes how the security of our IPSec implementation can be tested. Implementing according to the standard guaranties certain security features, but nothing assures that these features really work and that coding was done properly. With the intention of proving good security in our implementation, certain attacks to our implementation and its results are discussed and explained below.

Not all security features can be tested easily. For example, we were not able to prove the security of ESP encryption. The user of our IPSec system has no other choice but to trust the DES standard and carefully inspect its implementation.

6.3.2.1 Packet was altered during transmission

Scenario: A bad guy may want to modify the content of some network packets and somehow manages to alter the content, let's say of a HTTP transmission. If the packet was authenticated, he will not be able to recalculate the proper ICV because he doesn't know the required secret key to update the ICV in the packet. **Proper IPSec processing:** After the packet has entered the IPSec system it is processed by either the AH or ESP module. When authentication is enabled (AH or ESP with HMAC), IPSec recalculates the ICV using the secret authentication key. The recalculated value will not match the one stored in the packet and therefore the packet will be discarded. Verification of this threat: If a packet with some changed bits either in the packet itself or in the ICV value is injected into an IPSec stream our implementation will discard the packet with the following message:

```
        ERR ipsec_ah_check:              -2 AH ICV does not match
or
        ERR ipsec_ah_check:              -2 ESP ICV does not match
```

6.3.2.2 Non-IPSec packet, which should be one

Scenario: A bad guy may try to send non-IPSec packets to our IPSec enabled host. He may hope that a non-IPSec packet (which should be one according to the SPD) will reach the IP stack without any intervention.

Proper IPSec processing: When a clear text packet enters the system, it is checked against the SPD. If the policy says APPLY, the packet will be discarded (because the packet should be encrypted and IPSec decapsulation can't be applied to non-IPSec traffic). The goal of this test is to prevent that non-IPSec traffic bypasses the IPSec engine as requested by the RFC. Verification of this threat: The incoming packet will fail on the above-mentioned SPD lookup. The packet will be discarded with the following message:

```
AUD ipsecdev_input:   3: POLICY_APPLY: got non-IPsec packet which should be one
```

6.3.2.3 Packets are resent

Scenario: A bad guy may want to resend a certain IPSec packet.

Proper IPSec processing: This threat can only be caught if authentication is activated. This is the case when AH or ESP with authentication is used. Otherwise, the packet passes IPSec processing without any problems. In case of activated authentication, the anti-replay check will find out that the packet has already been sent, because the packet's bit in the bit-mask is already set to 1. Verification of this threat: It is necessary to resend a packet that has already been processed and thereby verify that authentication is turned on. Our IPSec implementation will discard this packet with the following message:

```
        AUD ipsec_ah_check          :  7 : packet rejected by anti-replay check
or
        AUD ipsec_esp_decapsulate  :  7 : packet rejected by anti-replay check
```

6.3.2.4 Packets that are out of the window

Scenario: A bad guy may want to disturb IPSec processing by sending IPSec packets with a high sequence number. This could lead the anti-replay mechanism to shift the anti-replay window in such a way, that the normal IPSec packets seem to arrive out of the window (their sequence number would be too low to be accepted).

Proper IPSec processing: The anti-replay check is only performed when authentication is turned on. This is the case when AH or ESP with an authentication algorithm is used. Before the packet is authenticated, a preliminary check of the sequence number is done to avoid wasting CPU time for authenticating packets that are obviously out of sequence. Only if the sequence number is valid (not yet obvious since it is within the window), the packet is passed to the authentication function. When authentication has passed, it will again be checked for validity of the sequence number before the packet is marked as seen and anti-replay window is shifted.

If the sequence number was altered, the integrity check would fail and the packet would be discarded.

Verification of this threat: If a forged sequence number is injected into an IPSec stream, our IPSec implementation will discard this packet with the following message:

```
AUD ipsec_ah_check        : 7 : packet rejected by anti-replay update
```

or

```
AUD ipsec_esp_decapsulate : 7 : packet rejected by anti-replay update
```

6.3.2.5 Packets with a bad SPI

Scenario: A bad guy may want to send IPSec packets with a forged SPI. If he monitors the IPSec traffic of a certain host, he is able to set a packet with a valid sequence number and a valid SPI, which could lead the IPSec system to properly process the packet. The correct sequence number is only required when authentication is activated. For example, the sequence number is not tested in the case of a ESP without authentication.

Proper IPSec processing: Such an attack would be possible if only an SA lookup was performed on incoming IPSec packets. The SA lookup uses only the outer destination address, the IPSec protocol and the SPI to determine the appropriate SA. However, the standard requires that after an inbound IPSec packet has been processed properly, an SPD lookup must be performed. A successful SPD lookup gives back a security policy with a pointer to the SA describing how the packet must be processed. To prevent such attacks, the SA pointer from the policy must now point to the same SA that was used to process the packet.

Verification of this threat: Inject a forged IPSec packet into an "ESP 3DES only" packet stream. The inner IP packet's fields may be modified in such a way, that there is no matching entry in the SPD. Our IPSec implementation discards such a packet with the following message:

```
AUD ipsecdev_input:   3: POLICY_APPLY: got non-IPsec packet which should be one
```

6.4 Specification Testing

Even if the code testing is performed exclusively, it doesn't provide grantee against the program failure. Code testing doesn't answer whether the code meets the agreed specification document. It doesn't also determine whether all aspects of the design are implemented.

Therefore, examining specifications stating what program should do and how it should behave under various conditions performs specification testing. Test cases are developed to test the range of values expected including both valid and invalid data. It helps in finding discrepancies between the system and its original objective. During these testing phases, all efforts were made to remove programming bugs and minor design faults.

6.5 Regression Testing

In regression testing the software was testing against the boundary condition. Various input fields were tested against abnormal values and it was tested that the software does not behave abnormally at any time.

6.6 Acceptance Testing

In acceptance testing the software was tested for its completeness that it is ready. Normally the quality assurance department performs the acceptance testing that the software is ready and can be exported.

6.7 Assertion Testing

In assertion testing the software is tested against the possible assertions. Assertions are used to check the program and various locations that whether the state of the program at a particular point is the same as expected or not.

6.8 Unit Testing

In unit testing we checked that all the individual components were working properly. Before integration of the entire components unit testing is essential because it gives a confidence that all the components individually are working fine and ready to be integrated with other ones.

6.9 System Testing

When all the units were working properly and unit testing was performed then comes the time for system testing where we checked all the integrated components as a whole and looked for possible discrepancies, which could have arisen after the integration.

6.10 System Evaluation

The objectives of the system evaluation are to determine whether the desired objectives have been accomplished or not. Determining the merits and demerits of the proposed system over the existing system is also covered in the system evaluation. This is concerned with the detailed study of the developed system, from implementation point of view. At the end, some suggestions for the improvements of the system are coded.

CHAPTER 7

Performance

7. Performance

Since there are many security protocols in terms of algorithms in IPSec, we had to choose appropriate algorithms. For this purpose we took into account several aspects:

- Existing documentation of simulations that expressed the time to compute the cryptographic functions involved in the algorithm.
- Low computational time of cryptographic functions.
- Algorithms that had proven enough reliability.
- Algorithms that comply with the basic requirements of the protocols.

For the reasons stated above, we decided on:

- HMAC – MD5 and HMAC – SHA1 to provide origin authentication and integrity for IP packets. MD5 should be preferred because its performance is much better than that of SHA1.
- In ESP implementation we support both encryption and authentication. Encryption is done by the widely used 3DES algorithm, which is applied in CBC mode. Pure DES is also implemented. For authentication we use HASH-MAC MD5.

7.1 Features

Our implementation is still a prototype but all the features that were requested for our work are quite well tested. We created test cases for almost each feature. Several functional tests were run over night to detect memory leaks. The nightly tests usually processed between thirty to sixty thousand packets without crashing and failing.

Such tests showed with reasonable certainty that features listed below are implemented in a quite stable manner. Utilization in a busy real-life network environment would probably show some not yet known shortcomings that were not apparent up to now due to the clean lab environment.

Our implementation has the following features:

- Dynamic Security Policy management.
- Dynamic Security Association management.
- AH protocol.
- ESP protocol.
- Support for AH with HMAC-MD5 and HMAC-SHA1
- Support for ESP with DES-HMAC-MD5 and 3DES-HMAC-MD5
- Support for tunnel mode
- Support for manual keying

7.2 Analysis & Results

Simulations of the implementation described previously were performed, using hand-off rate of 60 seconds. The scenario was simulated with a constant FTP source on top of TCP with a packet size of 2000 bytes between two nodes (node0 and node1).

	No IPSec	HMAC MD5	HMAC SHA1	DES HMAC-MD5	3DES HMAC-MD5
Simulation Start Time	1.3	1.4	1.4	1.5	1.6
Simulation End Time	60.08	60.07	59.66	59.91	60.04
Simulation length(Sec)	58.78	58.62	58.20	58.37	58.37
No of Generated Packets	4500	3501	2443	658	237
No of Sent Packets	4500	3501	2443	658	237
No of Lost Packets	2260	1753	1222	330	119
No of Received Packets	2240	1648	1217	328	118
Avg. Packet Size	1051	1066	1066	1074	1069
No of sent Bytes	4698000	3700136	2578664	701480	251668

Table 7.1 Retrieved Initial results

We measured the cumulative sum of packets in each of the cases, as well as the throughput, and end 2 end delay imposed by the security protocols: IPSec.

Not much comment is needed since the table can easily be interpreted. In some aspects, the results were as expected. The relative difference between the different cryptographic algorithms, correspond to the performance test, which were done.

The results from this chapter show, that depending on the chosen cryptography, various types of security can be added with various amounts of cost.

If only authentication and integrity is required, IPSec packets can be processed in quite a short time. Using strong encryption, a noticeable performance loss must be accepted.

Figure 7.1, depicts the increment of the TCP packets sequence in different scenarios (No IPSec), MD5 , SHA1 , DES-MD5 and 3DES-MD5), The illustrated scenarios showed that number of packets are decreasing in authentication and encryption compared to Bluetooth with no IPSec. As it is seen HMAC – MD5 perform better than HMAC – SHA1 while sending and receiving packets. As well as DES-MD5 performs better than 3DES-MD5.

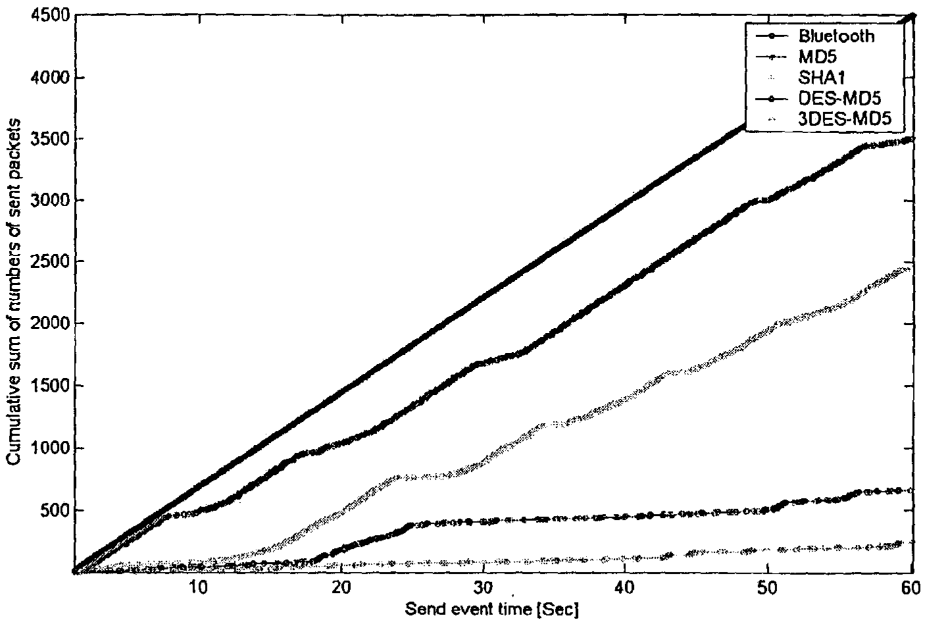


Figure 7.1(a) Performance of Cumulative sum of numbers of sent packets.

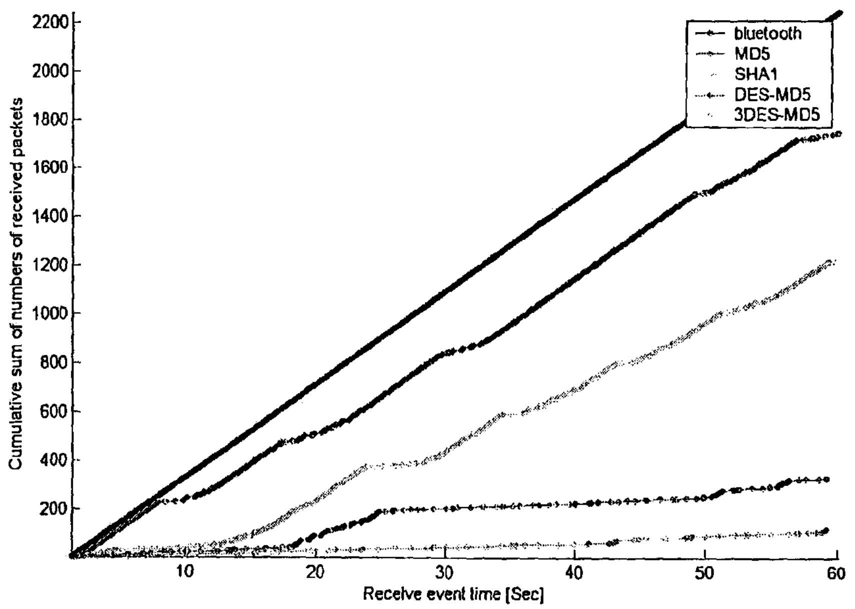


Figure 7.1(b) Performance of Cumulative sum of numbers of received packets.

The throughput results are shown in Fig 7.2. There is a significant difference between the simulated scenarios. Contrary to our expectations, the throughput in Bluetooth environment is driven by the effect of the erratic behavior of delay imposed by the encryption and decryption of the data and the erratic behavior of the Bluetooth wireless link.

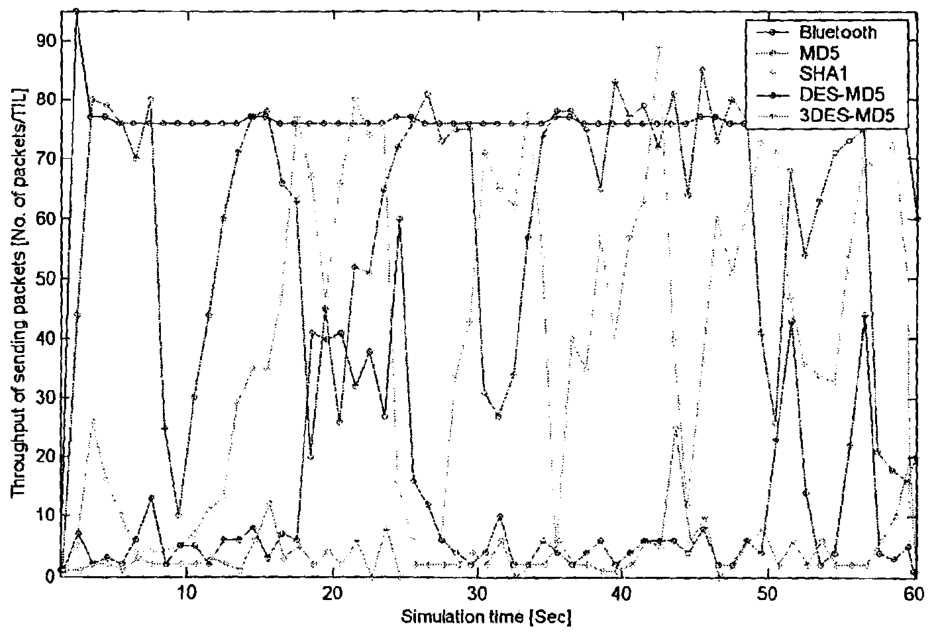


Figure 7.2(a) Throughput of sending packets.

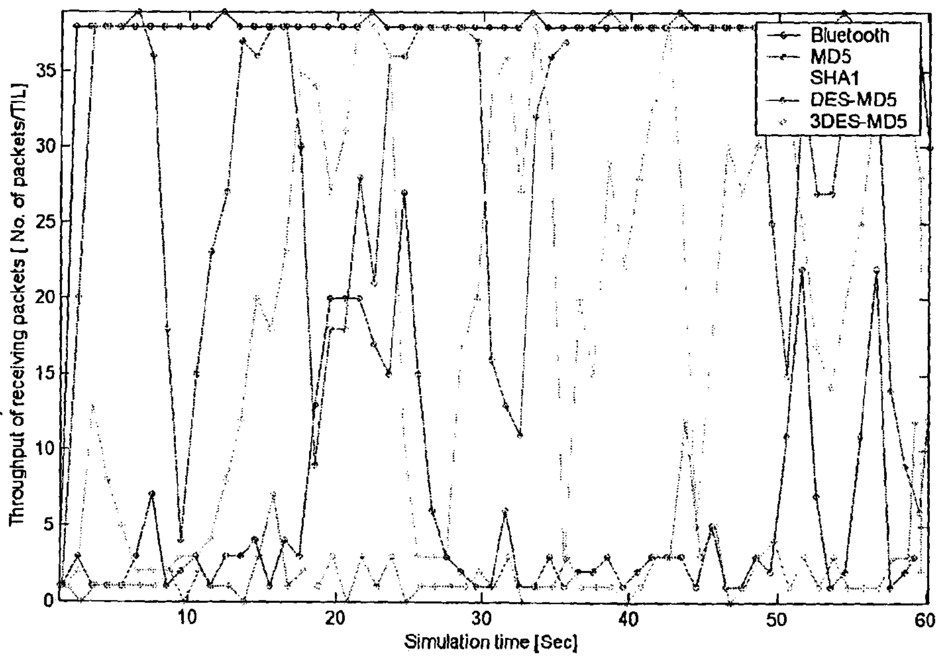


Figure 7.2(b) Throughput of receiving packets.

As we can see in Fig 7.2, the throughput in HMAC – MD5 and HMAC – SHA1 is not the same due to better performance of HMAC –MD5. For the encryption, DES-MD5 has better throughput than 3DES-MD5.

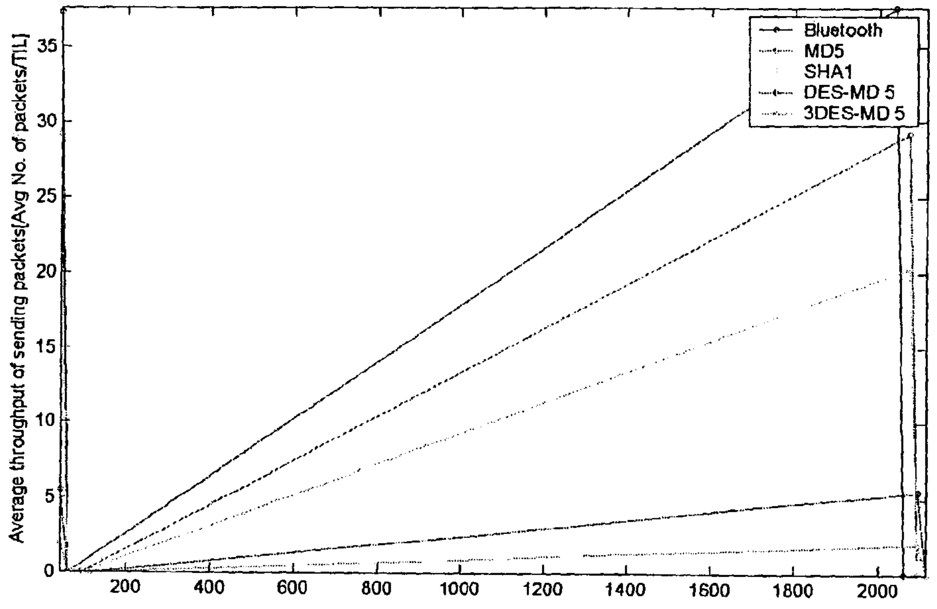


Figure 7.3(a) Packet size vs. average throughput of sending packets.

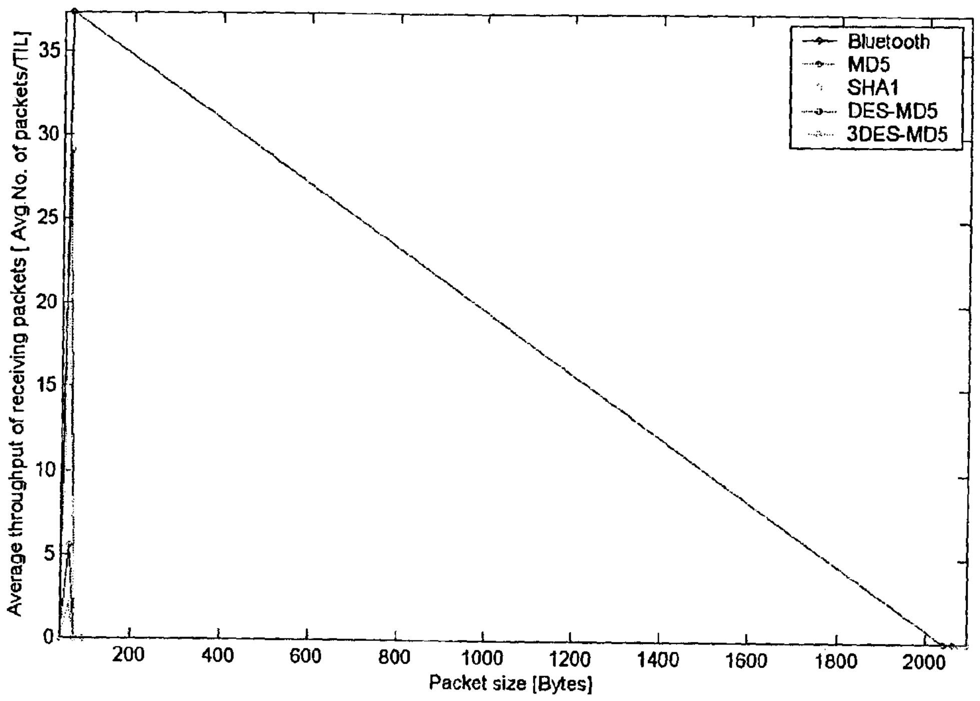


Figure 7.3(b) Packet size vs. average throughput receiving packets.

The following Figure 7.4 illustrates a comparison of Throughputs vs. average simulation End2End delay. There is a big and noticed difference of end 2 end delays while sending the bits, and rise in the throughput. While in receiving bits, both factors affect the bits transmission.

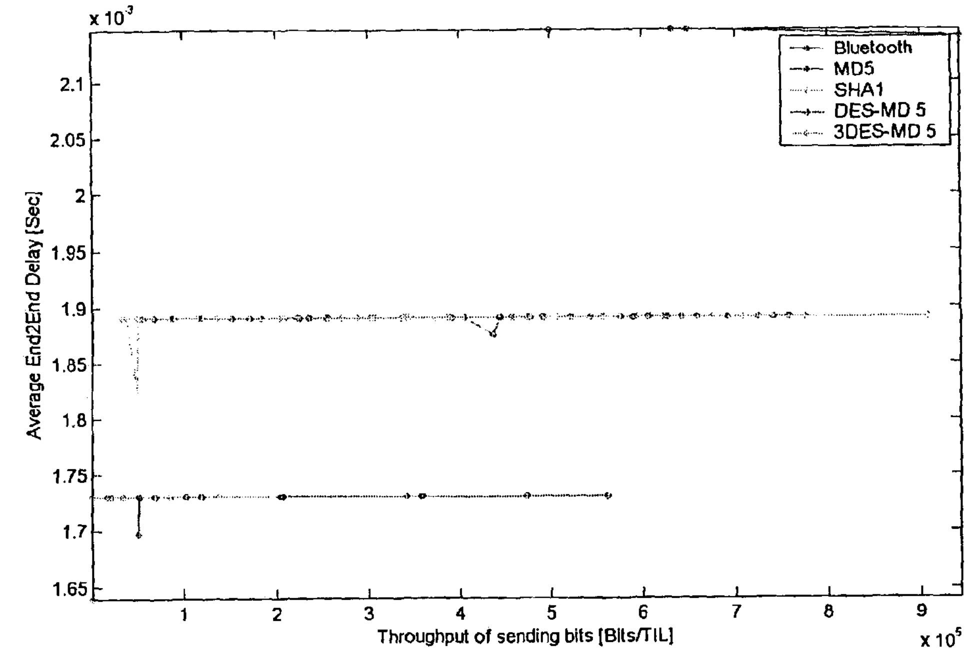


Figure 7.4(a) Throughputs of senidng bits vs. average simulation End2End delay

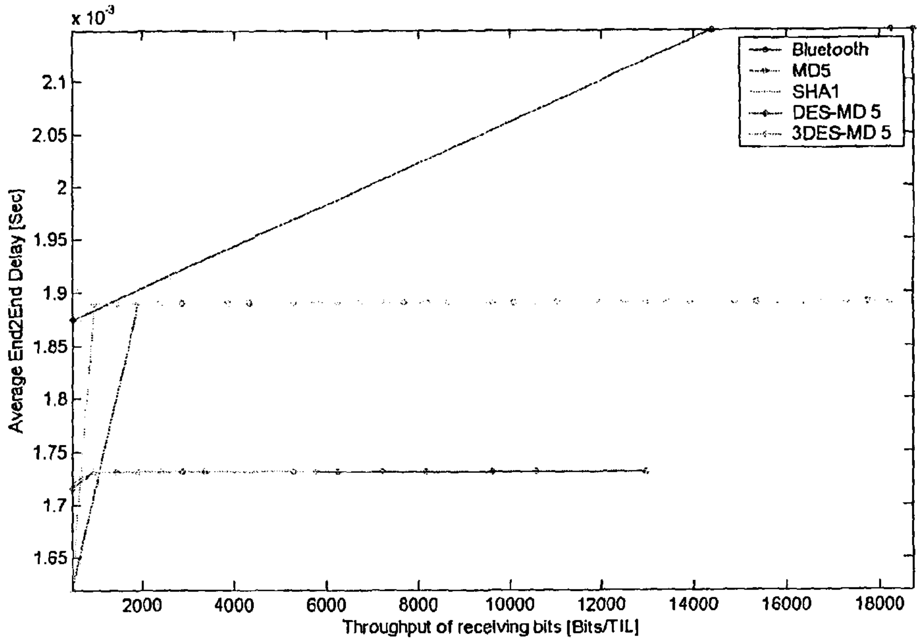
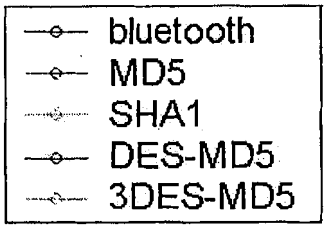


Figure 7.4(b) Throughputs of receiving bits vs. average simulation End2End delay



7.3 Conclusion

This research proposed a new Bluetooth security scheme, which allows ad-hoc (PAN) based on Bluetooth technology to communicate with other devices in full secure channel includes authentication and encryption, unlike for the present schemes with weak security.

IPSec protocols over Bluetooth do not impose a significant penalty. This is because the main factor in the reduction of the performance is due to the delay imposed by the encryption and decryption of the data and the erratic behavior of the wireless link. In addition, the throughput is reduced by almost 25% for authentication and 65% for encryption compared with the cases where IPSec was not used.

Bibliography

&

References

References & Bibliography

[1] Bluetooth Special Interest Group, *Specification of the Bluetooth System, Version 1.2, Core System Package*, November 1999.
[2] Bluetooth Special Interest Group, *Specification of the Bluetooth System, Version 1.2, Core System Package, Part B, Baseband Specification, Link Controller Operation*, November 2003.
[3] Lin, S., and D. J. Costello Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.
[4] Bluetooth Special Interest Group, *Specification of the Bluetooth System, Version 1.2, Core System Package, Part E, Host Controller Interface Functional Specification*, November 2003.
[5] CCITT: International Telegraph and Telephone Consultative Committee, *X.800: Data Communication Networks: Open Systems Interconnection (OSI); Security, Structure and Applications*, International Telecommunication Union, Geneva, 1991.
[6] van Oorschot, P.C., A. J. Menezes, and S. A. Vanstone, *Handbook of Applied Cryptography*, Boca Raton, FL: CRC Press, 1997.
[7] Dierks, T., and C. Allen, *The TLS Protocol, Version 1.0, RFC 2246*, January 1999.
[8] Harkins, D., and D. Carrel, RFC 2409, "*The Internet Key Exchange (IKE),* available at <http://www.ietf.org/rfc/rfc2409.txt>.
[9] Massey, J. L., and R. A. Rueppel, "Method of, and Apparatus for, Transforming a Digital Sequence into an Encoded Form," U.S. Patent No. 4,797,922, 1989.
[10] Bluetooth Special Interest Group, *Specification of the Bluetooth System, Version 1.1, Profiles, Part K: 1 Generic Access Profile*, February 2001.
< http://www.bluetooth.com/developer/specification/specification.asp >
[11] Müller, T., ed., "Bluetooth Security Architecture," White Paper Revision 1.0, Bluetooth Special Interest Group, July 1999.
[12] Chritian, G., Joakim, P., and Ben, S. *"Bluetooth Security"*, Artech House, June 2004.
[13] S. Kent and R. Atkinson, RFC 2402, *"IP Authentication Header"*, Network Working Group, available at <http://www.ietf.org/rfc/rfc2402.txt>.
[14] S. Kent and R. Atkinson, RFC 2406, *"IP Encapsulating Security Payload (ESP)"*, Network Working Group, available at <http://www.ietf.org/rfc/rfc2406.txt>.
[15] Janne Lundberg, "Routing Security in Ad Hoc Networks,
<http://citeseer.nj.nec.com/cache/papers/cs/19440/http:zSzzSzwww.tml.hut.fizSzjluz SznetseczSznetsec-lundberg.pdf/routing-security-in-ad.pdf>
[16] Secure data transmission in mobile ad hoc networks. P Papadimitratos, ZJ Haas - Proceedings of WiSe, 2003
<http://portal.acm.org/citation.cfm?id=941318&dl=ACM&coll=ACM>
[17] Bluetooth security JuhaT. Vainio. Department of Computer Science and Engineering, Helsinki University of Technology, 2000.
<http://www.iki.fi/jiitv/bluesec.html>
[18] Schneier B., Applied Cryptography, 2nd Ed., John Wiley & Sons Inc., 1996, 758p.
[19] RFC 2401, *Security Architecture for the Internet Protocol*, available at

<http://www.ietf.org/rfc/rfc2401.txt>

[20] RFC 2402, *IP Authentication Header*, available at <http://www.ietf.org/rfc/rfc2402.txt>.

[21] RFC 3316, *Internet Protocol Version 6 (IPv6) for Some Second and Third Generation Cellular Hosts,* available at http://www.ietf.org/rfc/rfc3316.txt.

[22] RFC 3884, *Use of IPSec Transport Mode for Dynamic Routing,* available at <http://www.ietf.org/rfc/rfc3884.txt>.

More information on IP-in-IP is available from RFC 2003, *IP Encapsulation within IP*, available at <http://www.ietf.org/rfc/rfc2003.txt>

[23] RFC 2104, *HMAC: Keyed-Hashing for Message Authentication, available at* <http://www.ietf.org/rfc/rfc2104.txt>.

[24] Federal agencies are required to use FIPS-approved algorithms and FIPS-validated cryptographic modules. HMAC-SHA-1 is a FIPS-approved algorithm, but HMAC-MD5 is not.

[25] RFC 3566, *The AES-XCBC-MAC-96 Algorithm and Its Use with IPSec*, available at <http://www.ietf.org/rfc/rfc3566.txt>.

[26] SPI is sometimes known as Security Parameter Index instead of Security Parameters Index. RFC 2402, *IP Authentication Header*, and RFC 2406, *IP Encapsulating Security Payload (ESP)*, use the word Parameters; RFC 2401, *Security Architecture for the Internet Protocol*, uses Parameter,

[27]<http://www.ethereal.com/>.

[28] The current draft of the proposed standard for AH version 3 is available at <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-rfc2402bis-xx.txt>. There is also a new proposed standard to replace RFC 2401, which provides an overview of IPsec version 2 (which includes AH version 2 and ESP version 2). The current version of the replacement for RFC 2401 is available at <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-rfc2401bis-xx.txt>.

[29] <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-esp-ah-algorithms-xx.txt >.

[30] Using IKE to negotiate IPSec protections can indirectly provide authentication for the source and destination IP addresses of ESP-protected packets as well.

[31] RFC 2406, IP Encapsulating Security Payload (ESP), available at <http://www.ietf.org/rfc/rfc2406.txt>.

[32] As specified in RFC 2406, ESP version 2 is only required to support DES for encryption, but most implementations support stronger encryption algorithms. For more information on such attacks, see the paper titled Problem Areas for the IP Security Protocols by Steven Bellovin, available at <http://www.research.att.com/~smb/papers/badesp.pdf>.

[33] Either ESP encryption or ESP authentication (but not both) can be set to null, disabling that capability.

[34] One possible issue is the inability to perform incoming source address validation to confirm that the source address is the same as that under which the IKE SA was negotiated.

[35] FIPS 197, Advanced Encryption Standard (AES), available at <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[36] <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-esp-v3-xx.txt>.

[37] <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ciph-aes-ccm-xx.txt>.

[38] <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-esp-ah-algorithms-xx.txt>.
[39] RFC 3686, Using Advanced Encryption Standard (AES) Counter Mode with IPSec Encapsulating Security Payload (ESP), available at <http://www.ietf.org/rfc/rfc3686.txt>.
[40] Bluetooth, The Bluetooth Specification, v.1.1B [referred 2001-02-11]. < http://www.bluetooth.com/developer/specification/specification.asp>
[41] Internet Engineering Task Force, "A Standard for the Transmission of IP Datagrams over Ethernet Networks", RFC 894.
[42] Internet Engineering Task Force, "Classical IP and ARP over ATM", RFC 2225, April 1998
[43] Internet Engineering Task Force, "IPv4 over IEEE 1394", RFC2734, December 1999.
[44] http://www.iana.org/assignments/ethernet-numbers
[45] "The Ethernet – A Local Area Network", Version 1.0 Digital Equipment Corporation, Intel Corporation, Xerox Corporation. September 1980.
[46] S. Kent and A. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, 1998.
[47] BNEP_specification_rev10RC3 [referred 2002-12-17]. < www.grc.upv.es/localdocs/blue/BNEP_specification_rev10RC3.pdf >
[48] The network simulator NS-2. < http://nsnam.isi.edu/nsnam/index.php/User_Information>
[49] UCBT - Bluetooth extension for NS2 at the University of Cincinnati < http://www.ececs.uc.edu/~cdmc/ucbt>
[50] http://www.isi.edu/nsnam/ns/ns-build.html

Appendix-A

User Manual

A User Manual

IPSec Based Bluetooth Security Architecture has been designed keeping in view user's interaction and ease in use. All interfaces are simple and easy to use. This user manual facilitates the user to understand different forms and interfaces. The use of forms and different options are described in details in this manual.

Steps to IPSec over Bluetooth

The following document is a step by step guide to using the IPSec over Bluetooth system. This is a step by step guide for a user assumed to have no technical experience other that an assumed basic familiarity with desktop computing.

The steps begin on bare PC's and cumulate to a full IPSec over Bluetooth between 2+ users. If you have a NS-2 and UCBT installed, skip to step A-4, otherwise proceed as follows.

A-1 Check out hardware

As far as hardware goes the minimum system requirements are recommended at the end of the performance testing and are as follows...

Minimum recommended system requirements:

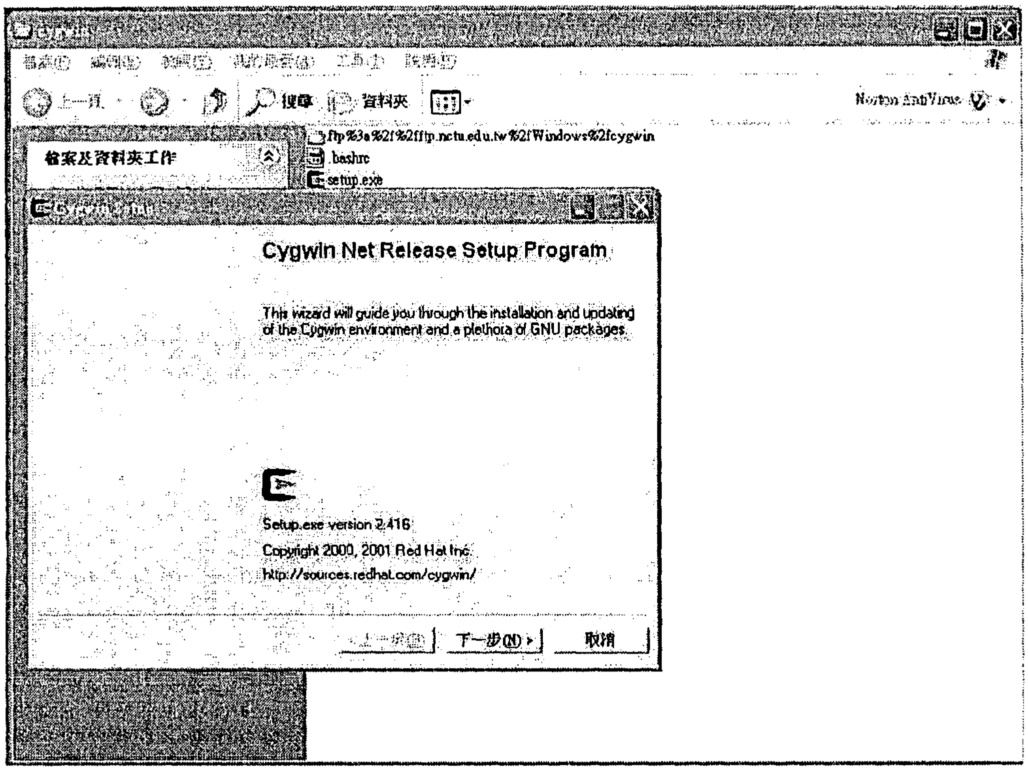
- 500 MHz processor.
- 128mb ram.
- Bluetooth Device.
- Monitor.
- Keyboard and Mouse.

A-2 Download and install NS-2

This part will show you how to install NS2 on Windows platform (windows 2000 or windows XP). The NS2 version for this document is ns-2.29.

Cygwin

1. Download the cygwin.rar from http://140.116.72.80/~smallko/ns2/cygwin.rar
2. Decompress the cygwin.rar
3. Click the setup.exe to install cygwin.

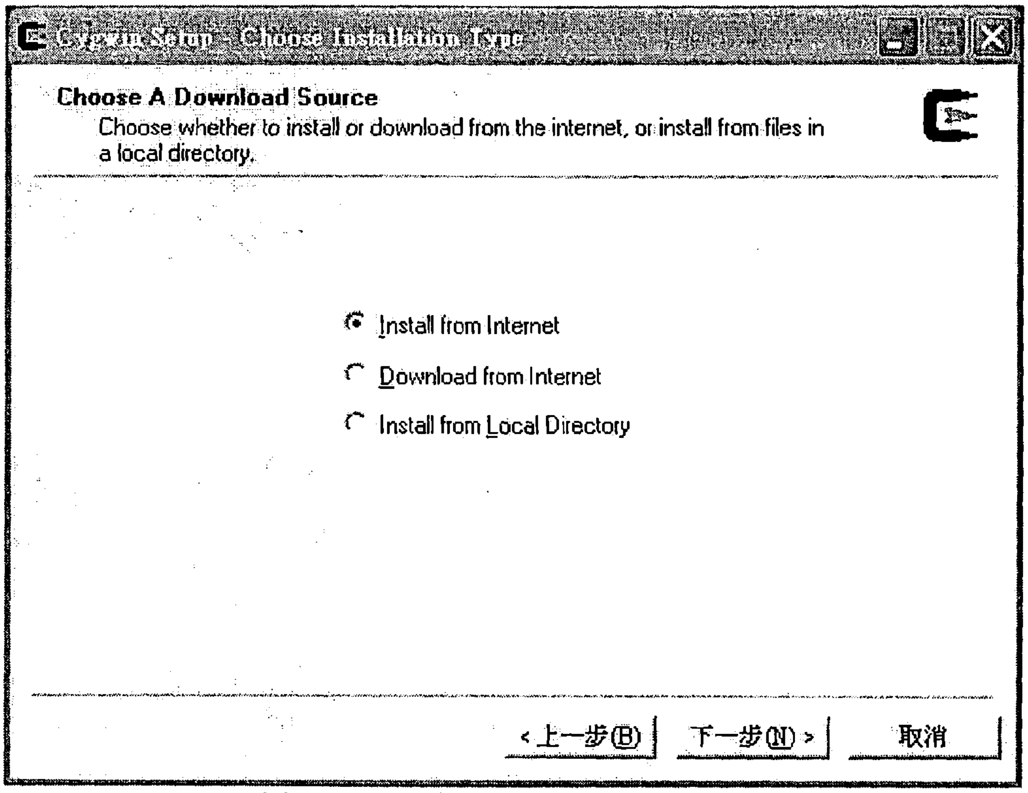

Cygwin Net Release Setup Program

This wizard will guide you through the installation and updating of the Cygwin environment and a plethora of GNU packages.

Setup.exe version 2.416
Copyright 2000, 2001 Red Hat Inc.
http://sources.redhat.com/cygwin/

Figure A.1 Install cygwin

4. Because the language of operating system is Traditional Chinese. The text of button is shown in Chinese. But you don‘¦t need to worry about this. I think it is shown in English in your computer. Just click "Next".

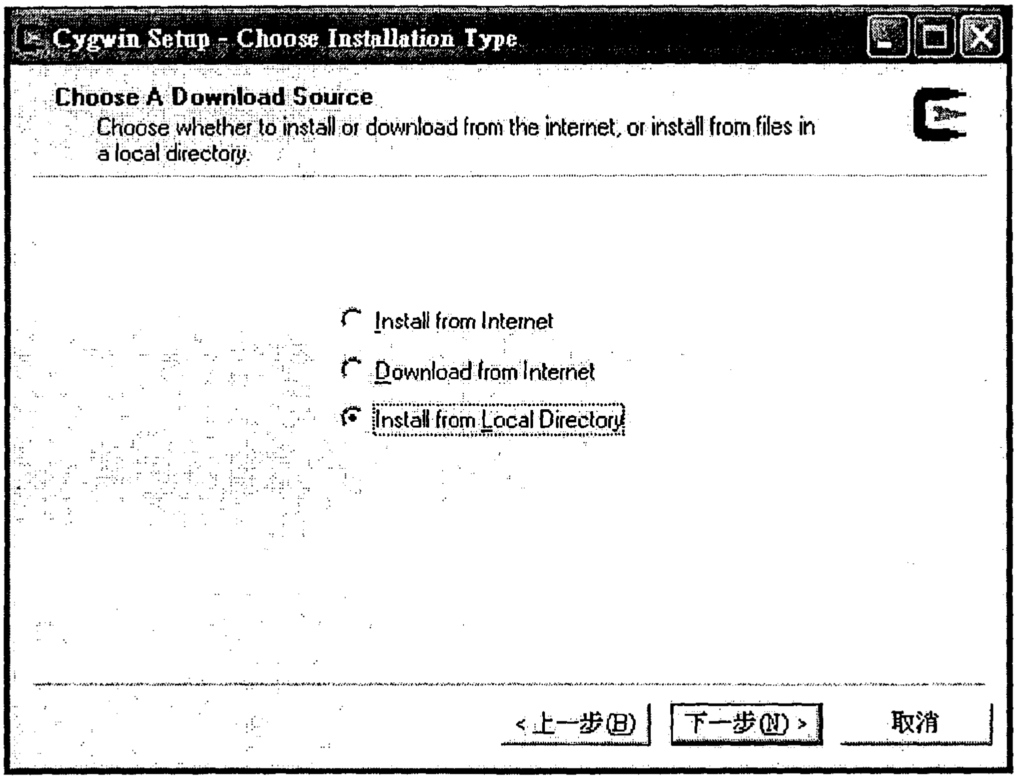

Cygwin Setup - Choose Installation Type

Choose A Download Source
Choose whether to install or download from the internet, or install from files in a local directory.

[✓ Install from Internet]
Download from Internet
Install from Local Directory

Figure A.2 Choose a download source

5. Then choose "Install from Local Directory".



Cygwin Setup - Choose Installation Type

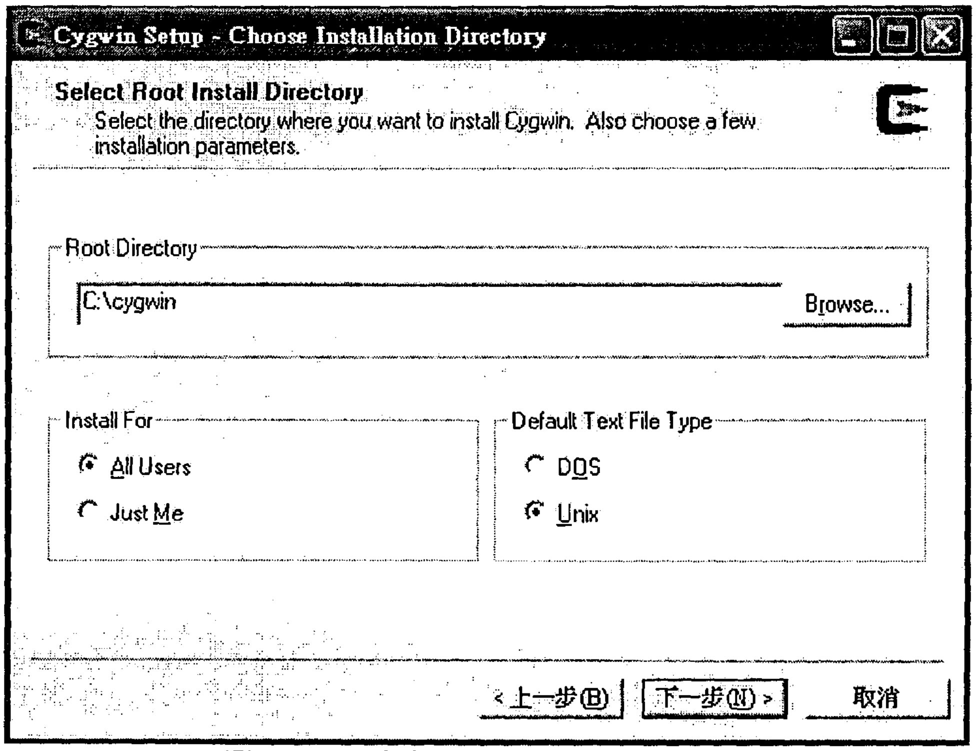
Choose A Download Source

Choose whether to install or download from the internet, or install from files in a local directory.

- Install from Internet
- Download from Internet
- [✓ Install from Local Directory]

Figure A.3 Install from Local Directory

6. Click "Next" and keep the settings as they are.



Cygwin Setup - Choose Installation Directory

Select Root Install Directory

Select the directory where you want to install Cygwin. Also choose a few installation parameters.

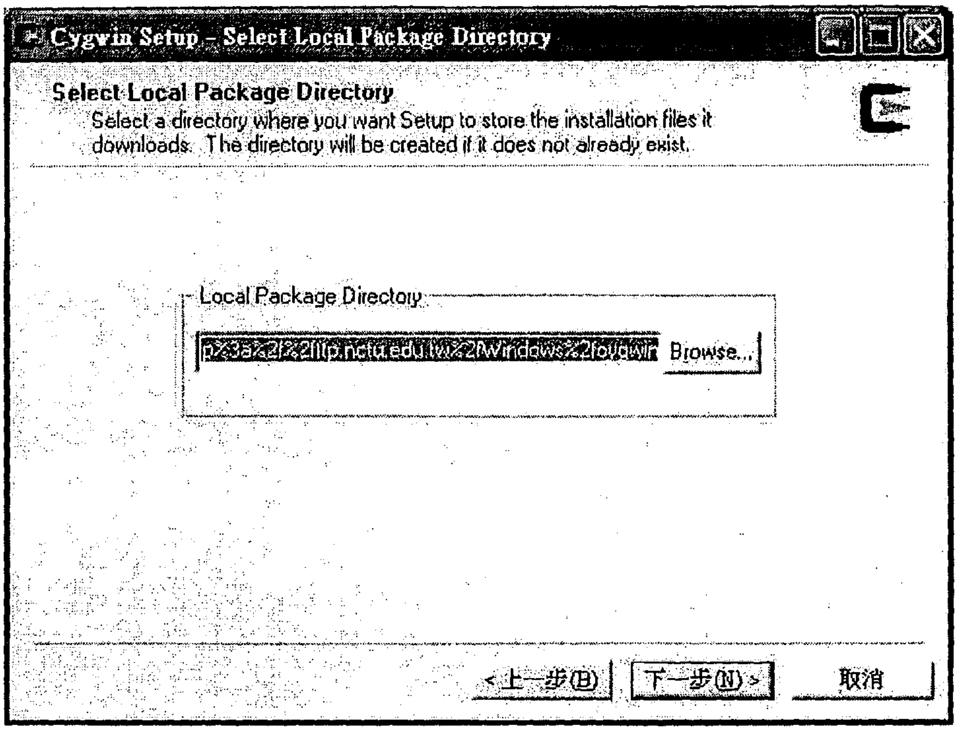
Root Directory: C:\cygwin

Install For: [✓ All Users] Just Me

Default Text File Type: DOS [✓ Unix]

Figure A.4 Select Root Install Directory

7. Click "Next".


Cygwin Setup - Select Local Package Directory

Select Local Package Directory

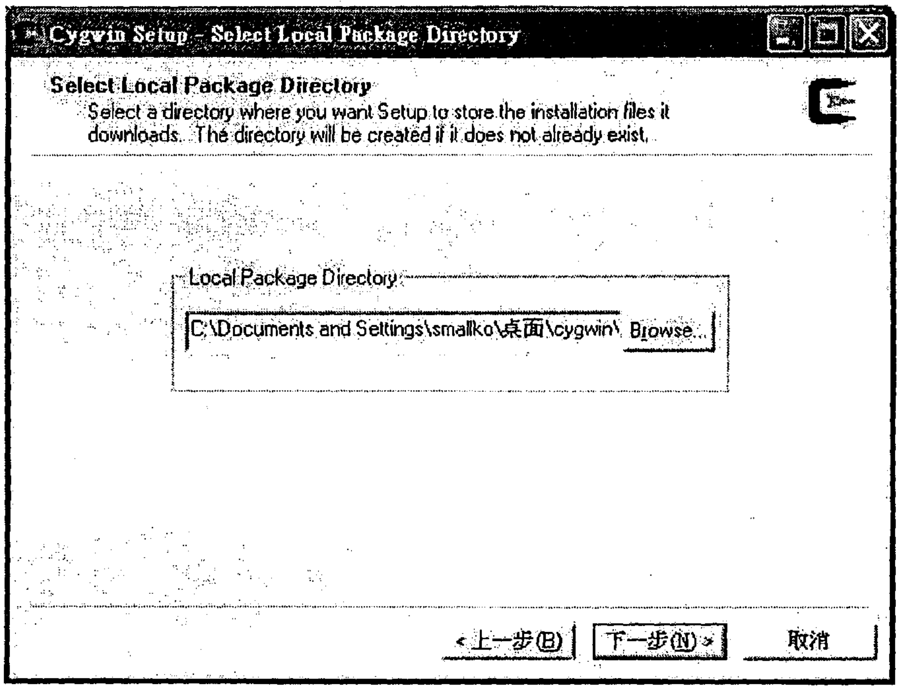
Select a directory where you want Setup to store the installation files it downloads. The directory will be created if it does not already exist.

Local Package Directory

p%3a%2f%2fftp.nctu.edu.tw%2fWindows%2fcygwin

Figure A.5 Select local Package Directory

8. Click "Browse" to choose where the software is. (choose '§ftp%3a%2f%2fftp.nctu.edu.tw%2fWindows%2fcygwin'")
9. Click "OK".


Cygwin Setup - Select Local Package Directory

Select Local Package Directory

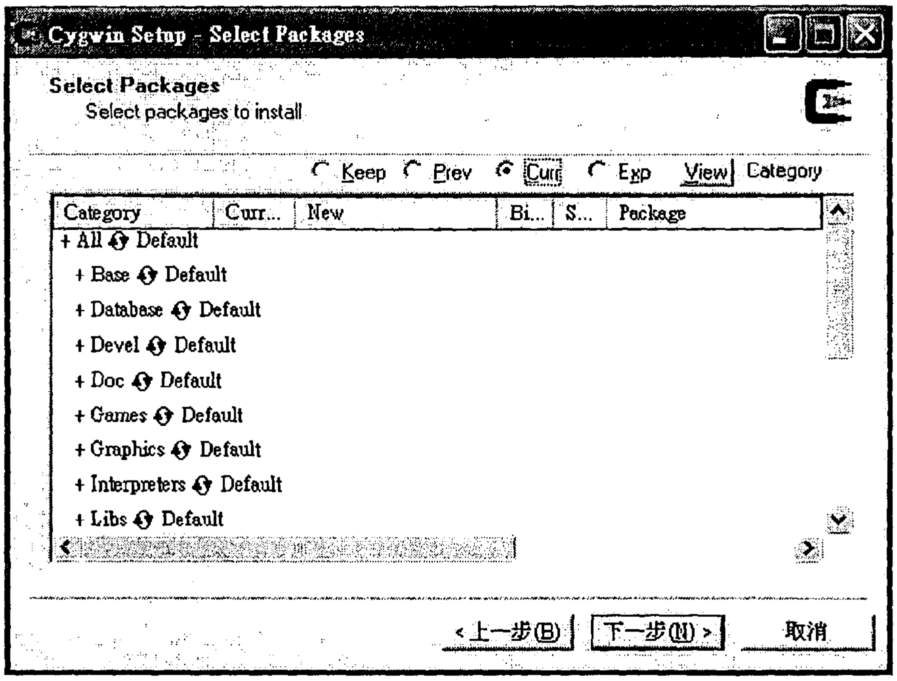
Select a directory where you want Setup to store the installation files it downloads. The directory will be created if it does not already exist.

Local Package Directory

C:\Documents and Settings\smallko\桌面\cygwin\

Figure A.6 Select local Package Directory

10. Click "Next" and you will see the figure shown as follows. In this window, the cygwin setup program let you choose what software you want to install.



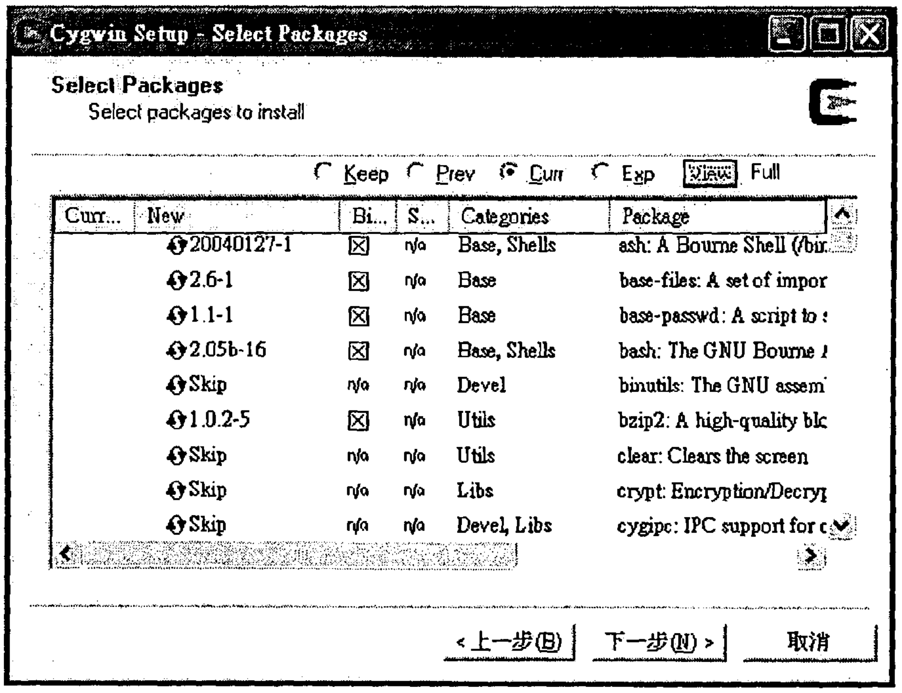
Select Packages
Select packages to install

Keep Prev [✓ Curr] Exp View Category

Category	Curr...	New	Bi...	S...	Package
+ All ↻ Default					
+ Base ↻ Default					
+ Database ↻ Default					
+ Devel ↻ Default					
+ Doc ↻ Default					
+ Games ↻ Default					
+ Graphics ↻ Default					
+ Interpreters ↻ Default					
+ Libs ↻ Default					

Figure A.7 Select Packages

11. Click "View" first to make the word ‹§Category ‹§change to "Full".



Select Packages
Select packages to install

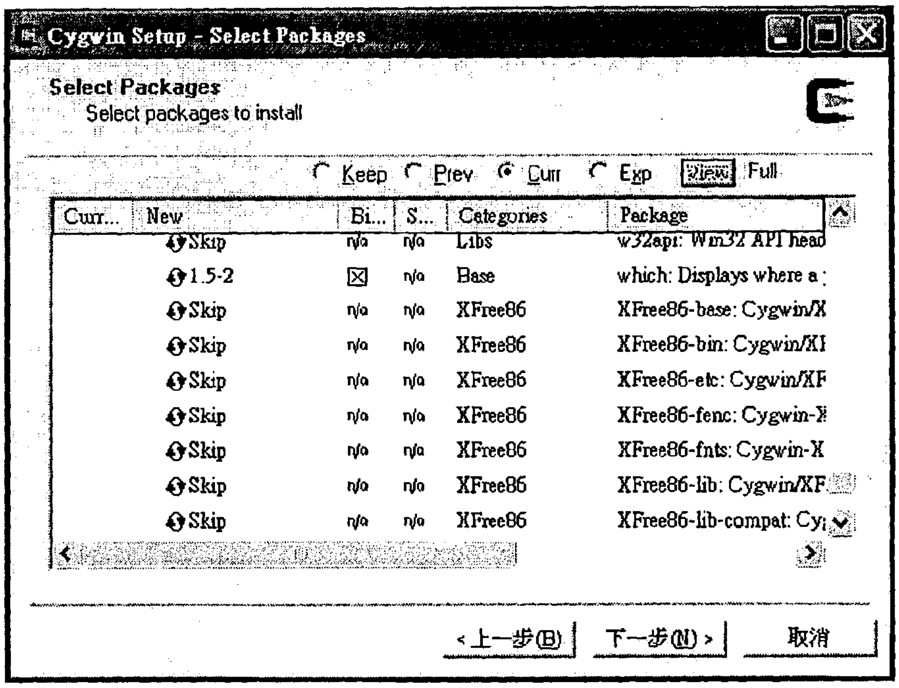
Keep Prev [✓ Curr] Exp View Full

Curr...	New	Bi...	S...	Categories	Package
	↻20040127-1	☒	n/a	Base, Shells	ash: A Bourne Shell (/bi
	↻2.6-1	☒	n/a	Base	base-files: A set of impor
	↻1.1-1	☒	n/a	Base	base-passwd: A script to
	↻2.05b-16	☒	n/a	Base, Shells	bash: The GNU Bourne
	↻Skip	n/a	n/a	Devel	binutils: The GNU assem
	↻1.0.2-5	☒	n/a	Utils	bzip2: A high-quality blc
	↻Skip	n/a	n/a	Utils	clear: Clears the screen
	↻Skip	n/a	n/a	Libs	crypt: Encryption/Decry
	↻Skip	n/a	n/a	Devel, Libs	cygipc: IPC support for c

Figure A.8 Changing Category

12. XFree86-base, XFree86-bin, XFree86-prog, XFree86-lib, XFree86-etc, make, patch, perl, gcc, gcc-g++, gawk, gnuplot, tar and gzip must be chosen. For example, if

I want to install XFree-86 base (upper figure), click the "Skip" of "New" column. The "Skip" will be changed to "4.3.0-1".


Cygwin Setup - Select Packages

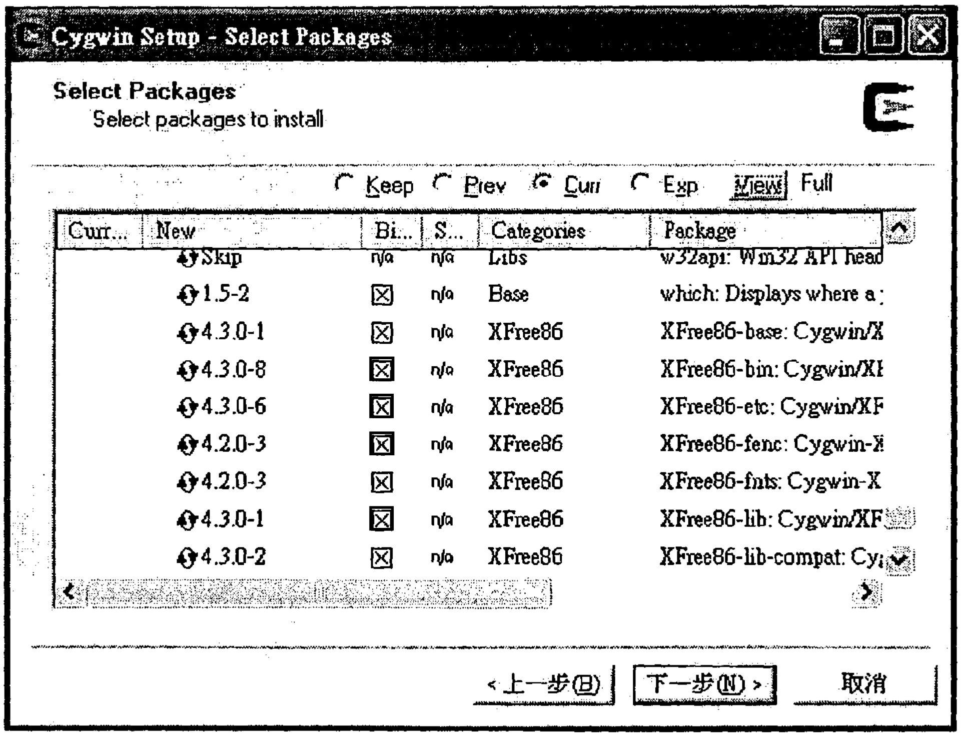
Select Packages
Select packages to install

Keep Prev [✓ Curr] Exp View Full

Curr...	New	Bi...	S...	Categories	Package
	Skip	n/a	n/a	Libs	w32api: Win32 API head
	1.5-2	☒	n/a	Base	which: Displays where a
	Skip	n/a	n/a	XFree86	XFree86-base: Cygwin/X
	Skip	n/a	n/a	XFree86	XFree86-bin: Cygwin/XI
	Skip	n/a	n/a	XFree86	XFree86-etc: Cygwin/XF
	Skip	n/a	n/a	XFree86	XFree86-fenc: Cygwin-X
	Skip	n/a	n/a	XFree86	XFree86-fnts: Cygwin-X
	Skip	n/a	n/a	XFree86	XFree86-lib: Cygwin/XF
	Skip	n/a	n/a	XFree86	XFree86-lib-compat: Cy

< 上一步(B) 下一步(N) > 取消

Figure A.9 Select Packages to Install


Cygwin Setup - Select Packages

Select Packages
Select packages to install

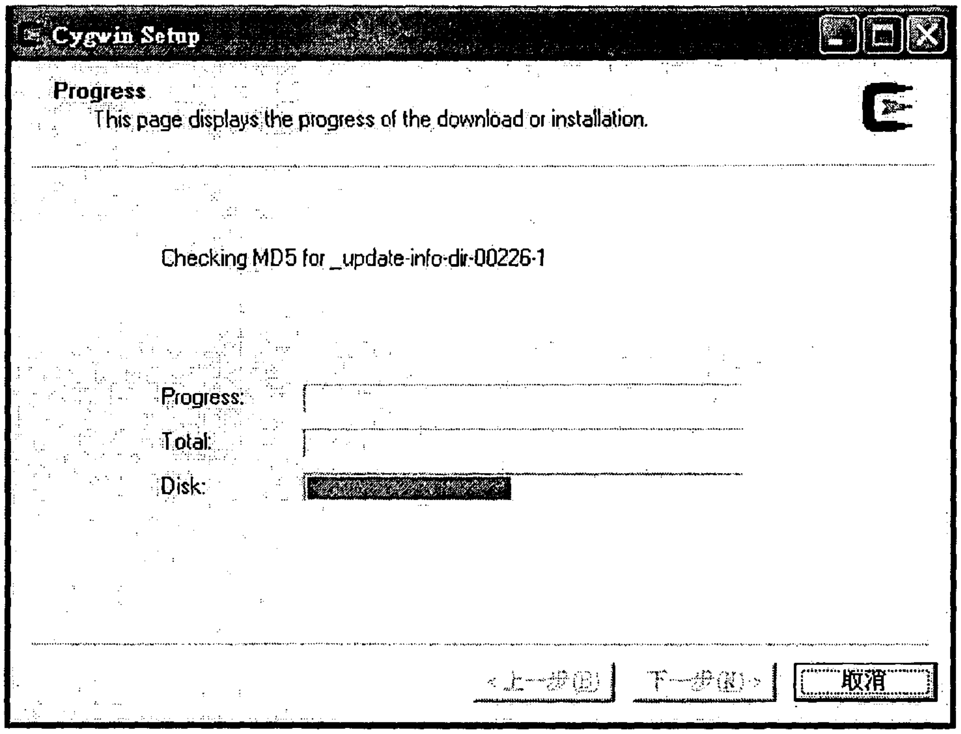
Keep Prev [✓ Curr] Exp View Full

Curr...	New	Bi...	S...	Categories	Package
	Skip	n/a	n/a	Libs	w32api: Win32 API head
	1.5-2	☒	n/a	Base	which: Displays where a
	4.3.0-1	☒	n/a	XFree86	XFree86-base: Cygwin/X
	4.3.0-8	☒	n/a	XFree86	XFree86-bin: Cygwin/XI
	4.3.0-6	☒	n/a	XFree86	XFree86-etc: Cygwin/XF
	4.2.0-3	☒	n/a	XFree86	XFree86-fenc: Cygwin-X
	4.2.0-3	☒	n/a	XFree86	XFree86-fnts: Cygwin-X
	4.3.0-1	☒	n/a	XFree86	XFree86-lib: Cygwin/XF
	4.3.0-2	☒	n/a	XFree86	XFree86-lib-compat: Cy

< 上一步(B) 下一步(N) > 取消

Figure A.10 Select Packages to Install

13. Click "Next". Please be patient. It may take a long time to finish the installation.



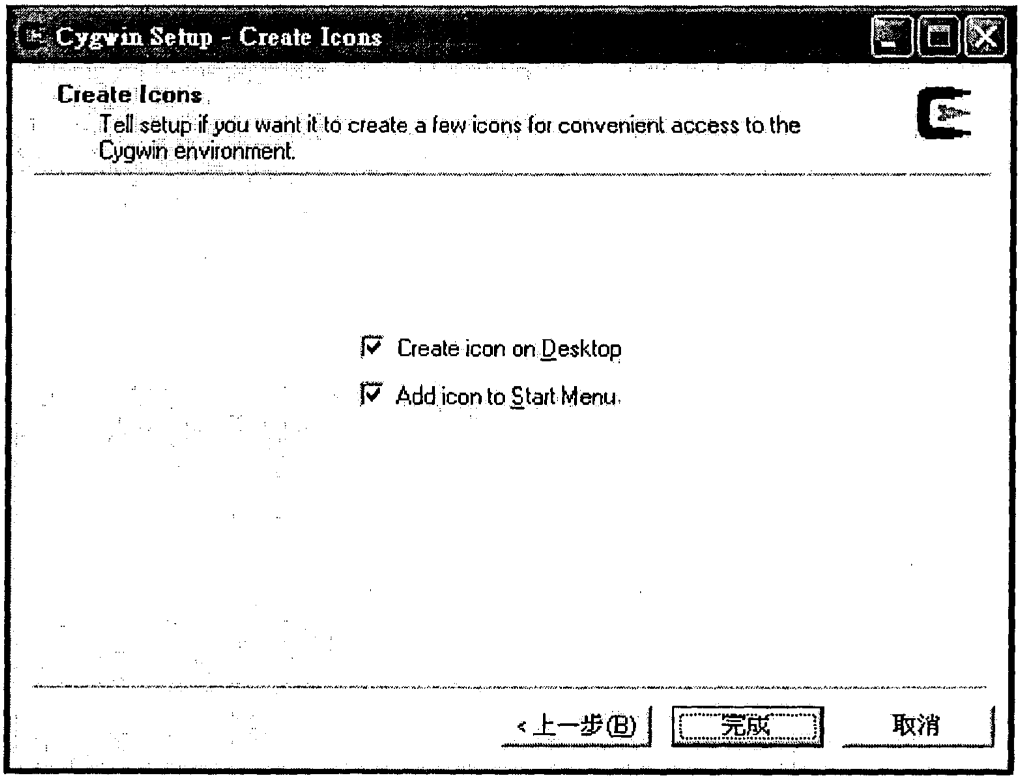
This page displays the progress of the download or installation.

Checking MD5 for _update-info-dir-00226-1

Progress:
Total:
Disk:

Figure A.11 Installation Progress

14. When setup is done, it will be shown as following figure. Click "Finish".



Tell setup if you want it to create a few icons for convenient access to the Cygwin environment.

☑ Create icon on Desktop

☑ Add icon to Start Menu

Figure A.12 Create Icons

15. Click "OK" to finish the cygwin setup program.

NS2 setup

1. Click Cygwin icon on the desktop.

2. For the first time execution, it will generate some environment parameter setting files. In this example, smallko is my login name to windows system. Therefore, the cygwin will create a folder named ‘§smallko‘¨ under home directory. (The actual path for smallko folder is: c:\cygwin\home\smallko) It should be noticed that the login name can not have any space in your name. For example, ‘§A B‘¨ may cause errors when you install NS2.

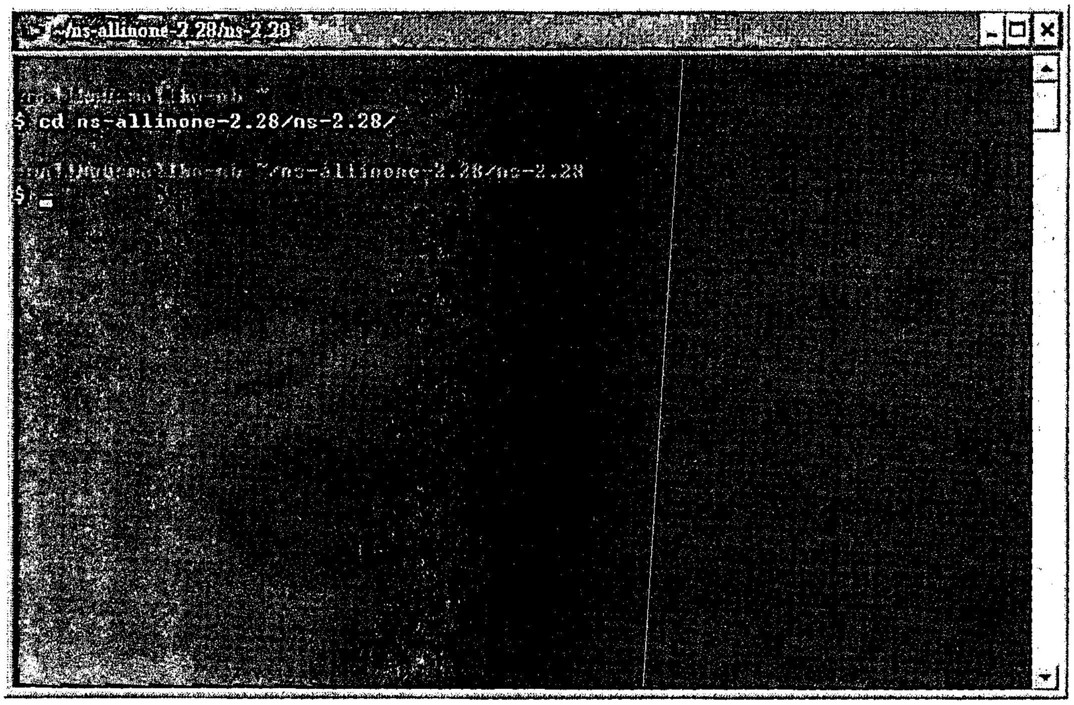
3. Download NS-2 fromhttp://prdownloads.sourceforge.net/nsnam/ns-allinone-2.29.2.tar.gz?download.

4. Decompress the ns-allinone-2.29.rar.

5. Move this folder under c:\cygwin\home\smallko. (P.S. smallko is my login name)

6. Open a cygwin window

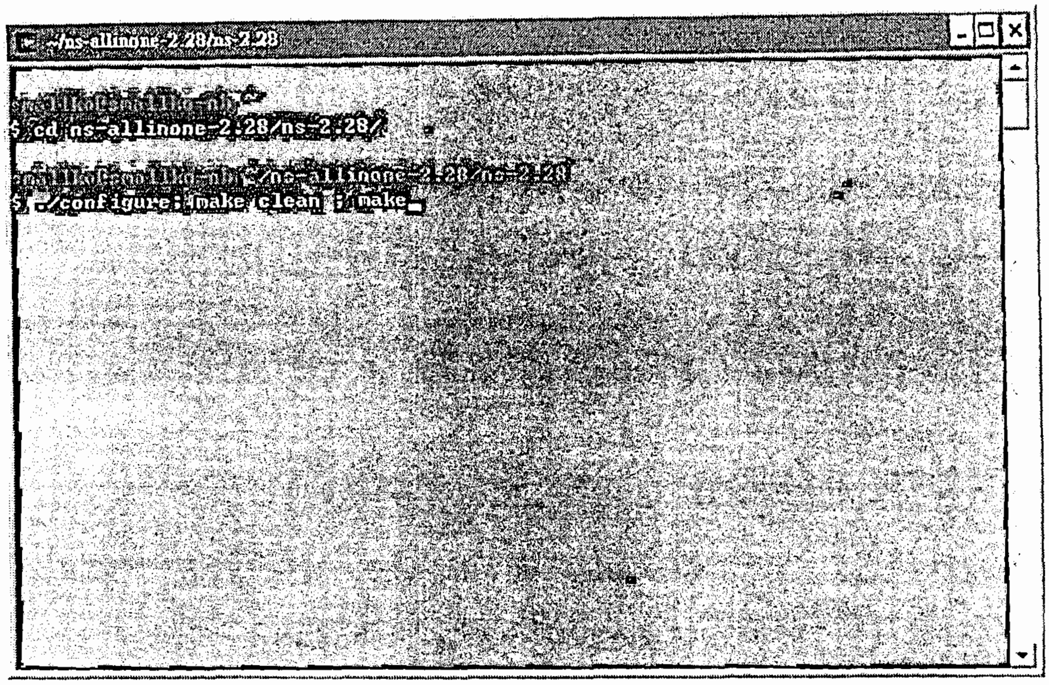
7. Change the path to ns-allinone-2.29/ns-2.29



```
~/ns-allinone-2.28/ns-2.28

[illegible] ~
$ cd ns-allinone-2.28/ns-2.28/

[illegible] ~/ns-allinone-2.28/ns-2.28
$ _
```

Figure A.13 Changing Path

8. Run the command "./configure; make clean; make"

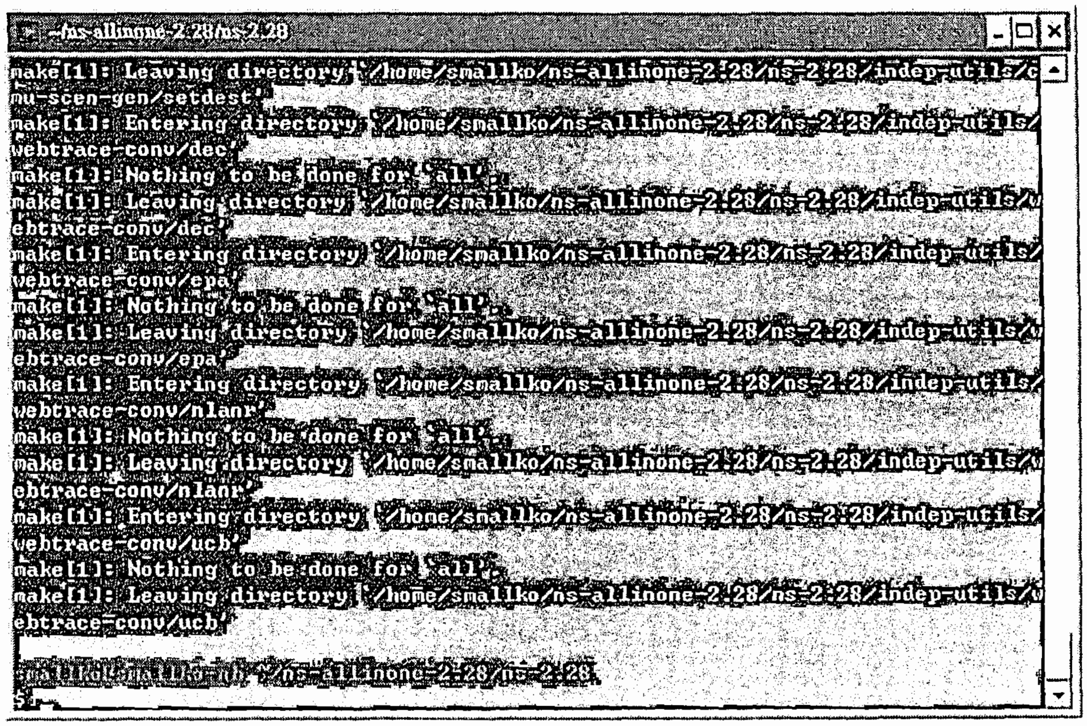


```
~/ns-allinone-2.28/ns-2.28

[illegible] ~
$ cd ns-allinone-2.28/ns-2.28/

[illegible] ~/ns-allinone-2.28/ns-2.28
$ ./configure; make clean ; make
```

Figure A.14 Configure NS2

9. Please be patient. It will take some time to finish the compilation.

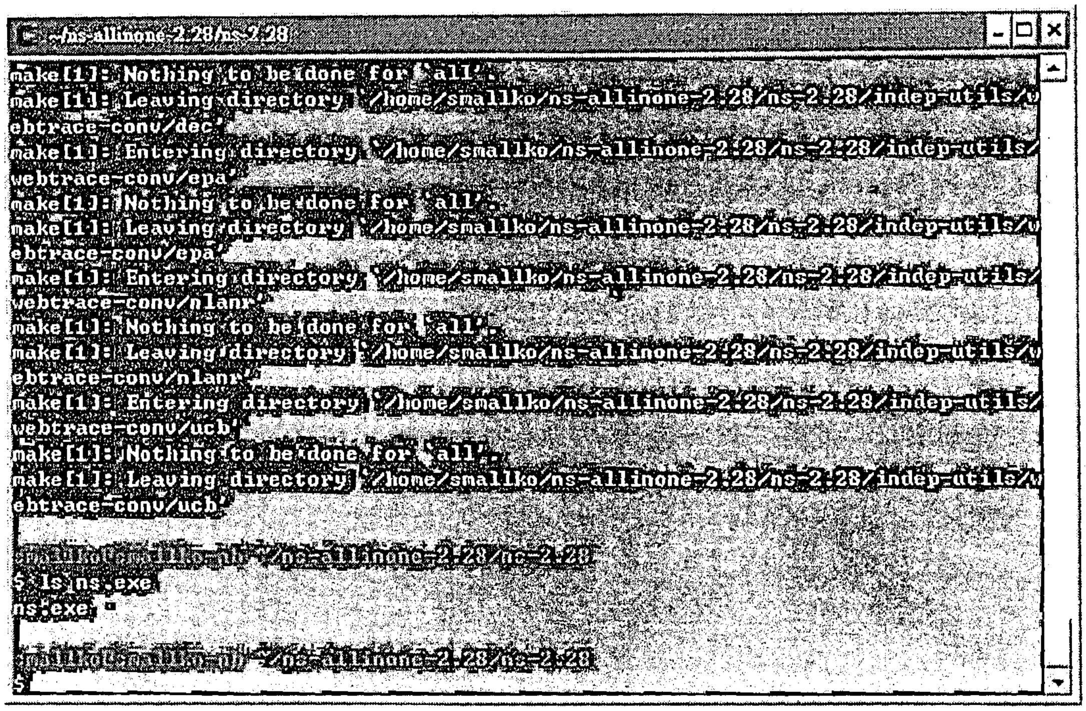
10. When it is done, it should look like as follows.



```
~/ns-allinone-2.28/ns-2.28
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/c
mu-scen-gen/setdest'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/dec'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/dec'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/epa'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/epa'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/nlanr'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/nlanr'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/ucb'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/ucb'

[illegible] ~/ns-allinone-2.28/ns-2.28
$
```

Figure A.15 Compilation Complete

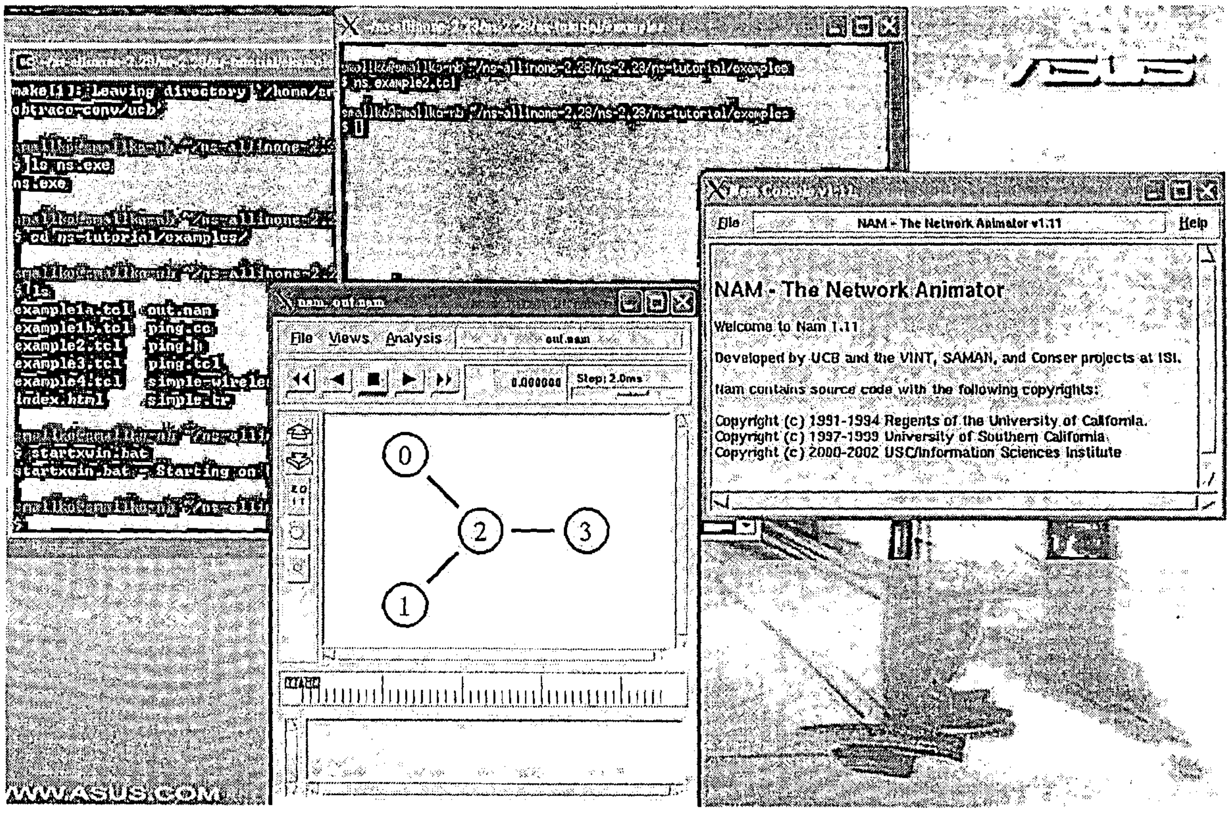
11. To make sure that you have successfully installed myNS2, you need to check whether you can find ns.exe under ns-allinone-2.29/ns-2.29



```
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/dec'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/epa'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/epa'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/nlanr'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/nlanr'
make[1]: Entering directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/
webtrace-conv/ucb'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/smallko/ns-allinone-2.28/ns-2.28/indep-utils/w
ebtrace-conv/ucb'

smallko@smallko-nb ~/ns-allinone-2.28/ns-2.28
$ ls ns.exe
ns.exe

smallko@smallko-nb ~/ns-allinone-2.28/ns-2.28
$
```

Figure A.16 Checking NS.exe

12. Copy the .bashrc to c:/cygwin/home/smallko
13. Run the example script to test whether you have setup the path or not.



```
make[1]: Leaving directory `/home/sm
ebtrace-conv/ucb'

smallko@smallko-nb ~/ns-allinone-2.2
$ ls ns.exe
ns.exe

smallko@smallko-nb ~/ns-allinone-2.2
$ cd ns-tutorial/examples/

smallko@smallko-nb ~/ns-allinone-2.2
$ ls
example1a.tcl  out.nam
example1b.tcl  ping.cc
example2.tcl   ping.h
example3.tcl   ping.tcl
example4.tcl   simple-wirele
index.html     simple.tr

smallko@smallko-nb ~/ns-alli
$ startxwin.bat
startxwin.bat - Starting on

smallko@smallko-nb ~/ns-alli
$
```

```
smallko@smallko-nb ~/ns-allinone-2.28/ns-2.28/ns-tutorial/examples
$ ns example2.tcl

smallko@smallko-nb ~/ns-allinone-2.28/ns-2.28/ns-tutorial/examples
$
```

NAM - The Network Animator

Welcome to Nam 1.11

Developed by UCB and the VINT, SAMAN, and Conser projects at ISI.

Nam contains source code with the following copyrights:

Copyright (c) 1991-1994 Regents of the University of California.
Copyright (c) 1997-1999 University of Southern California
Copyright (c) 2000-2002 USC/Information Sciences Institute

Figure A.17 Nam window

14. If you see the error message like '§ns command not found'", no worry about this. Sometimes even you have setup the path, but it does not work. You can copy the ns.exe (nam.exe) to the same place as the simulation script. Run the simulation with"./ns.exe" and it will be ok.

Testing

1. To initial graphical mode. (Type startxwin.bat).
2. Run the example tcl script. (Change to ~/ns-allinone-2.28/ns-2.28/ns-tutorial/examples. Then run the command "ns example2.tcl")
3. If you can see the above figure, congratulations. You have successfully install cygwin + ns-2.28 under windows platform.

A-3 Download and Install UCBT

1. Get ns-allinone-2.29.tar.gz
   ftp://ftp.isi.edu/nsnam/ns-allinone-2.29.tar.gz

2. Get ucbt-xx.tgz
   http://www.ececs.uc.edu/~cdmc/ucbt/src

3. tar zxvf ns-allinone-2.29.tar.gz

4. cd ns-allinone-2.29/ns-2.29/
   tar zxvf../../ucbt-xx.tgz
   Assume ucbt-xx.tgz and ns-allinone-2.29.tar.gz is in the same directory.
5. cd ucbt-xx/
   . /install-bt
   Or
   . /install-bt -d # enable debug
   Or
   . /install-bt -t # install tcl-debug

   ucbt-xx will be linked as Bluetooth.

6. If you want debug enabled, while the debug option is not enable in the above step,
   cd patch/ns-2.29
   ./enable-ns-debug.sh

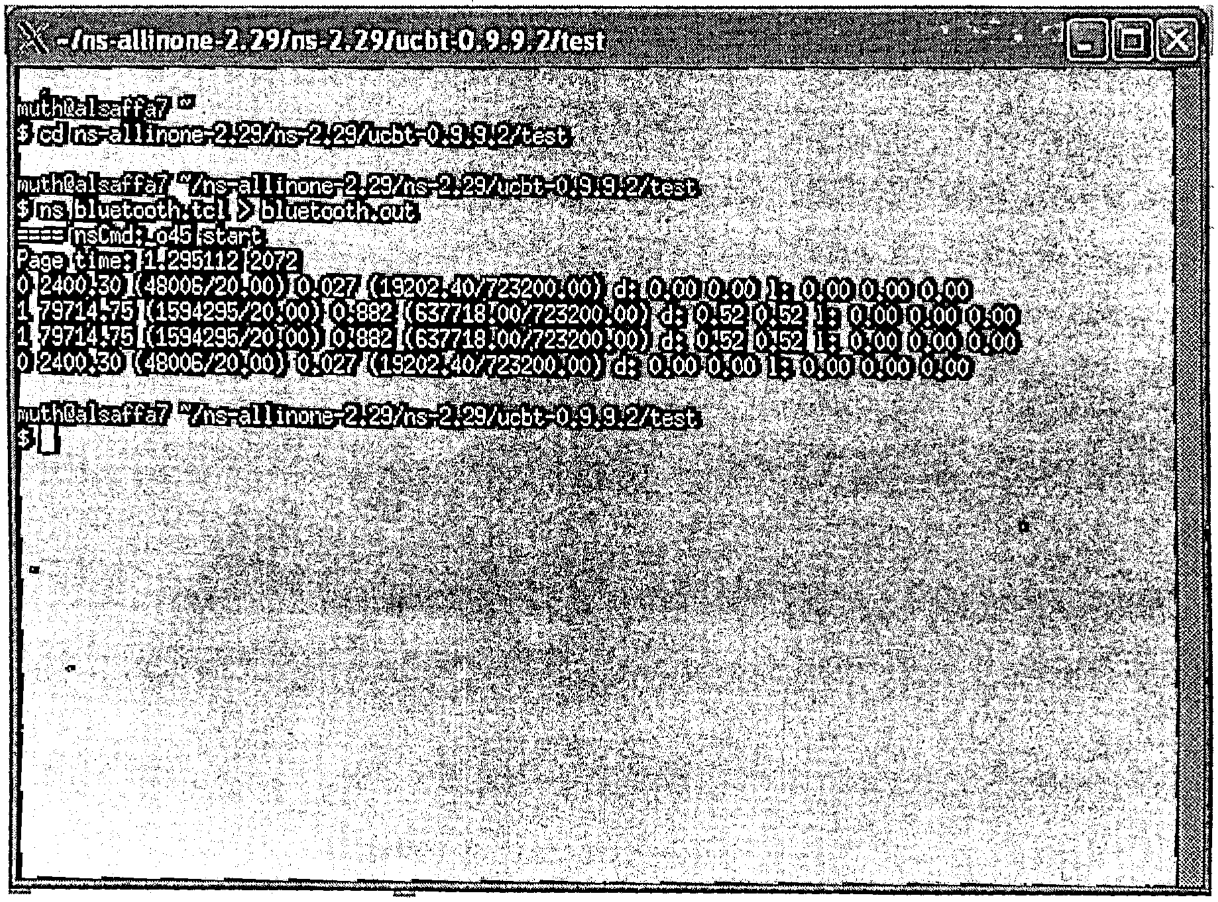
7. Try some tests:
   cd test/
   ../../ns test.tcl > test.out

8. To generate mobile scenario:
   cd tools && make
   Then you can use the modified setdest program.

9. To recompile if you make some changes to the source code:
   make # in bluetooth/

A-4 Running IPSec over Bluetooth

1) cd ns-allinone-2.29/ns-2.29/
2) cd ucbt-0.9.9.2
3) cd project
4) The following Files will be present:
   a) Bluetooth
   b) MD5
   c) SHA1
   d) DES-MD5
   e) 3DES-MD5
5) To run TCL file : ns *file-name*.tcl > *file-name*.out



```
~/ns-allinone-2.29/ns-2.29/ucbt-0.9.9.2/test

muth@alsaffa7 ~
$ cd ns-allinone-2.29/ns-2.29/ucbt-0.9.9.2/test

muth@alsaffa7 ~/ns-allinone-2.29/ns-2.29/ucbt-0.9.9.2/test
$ ns bluetooth.tcl > bluetooth.out
==== nsCmd: _o45 start
Page time: 1.295112 2072
0 2400.30 (48006/20.00) 0.027 (19202.40/723200.00) d: 0.00 0.00 l: 0.00 0.00 0.00
1 79714.75 (1594295/20.00) 0.882 (637718.00/723200.00) d: 0.52 0.52 l: 0.00 0.00 0.00
1 79714.75 (1594295/20.00) 0.882 (637718.00/723200.00) d: 0.52 0.52 l: 0.00 0.00 0.00
0 2400.30 (48006/20.00) 0.027 (19202.40/723200.00) d: 0.00 0.00 l: 0.00 0.00 0.00

muth@alsaffa7 ~/ns-allinone-2.29/ns-2.29/ucbt-0.9.9.2/test
$ 
```

Figure A.18 Running TCL & Result

6) To run NAM file: nam *file-name*.nam

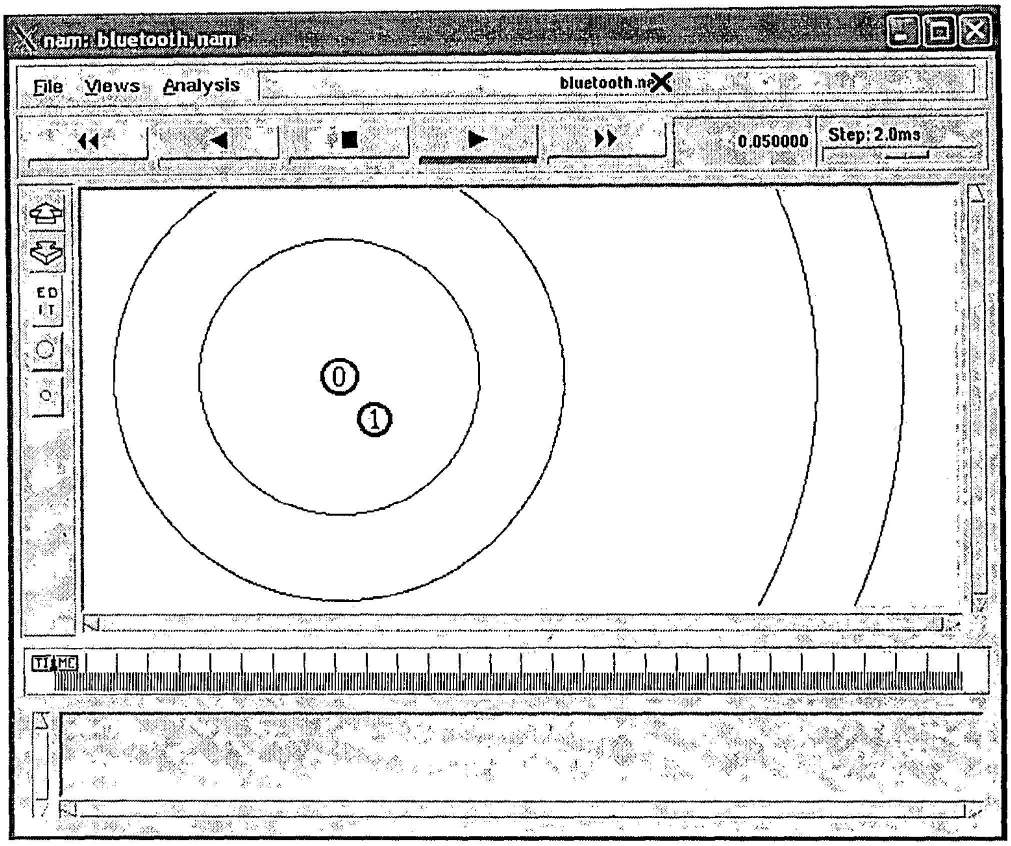


Figure A.19 Transmission of Data

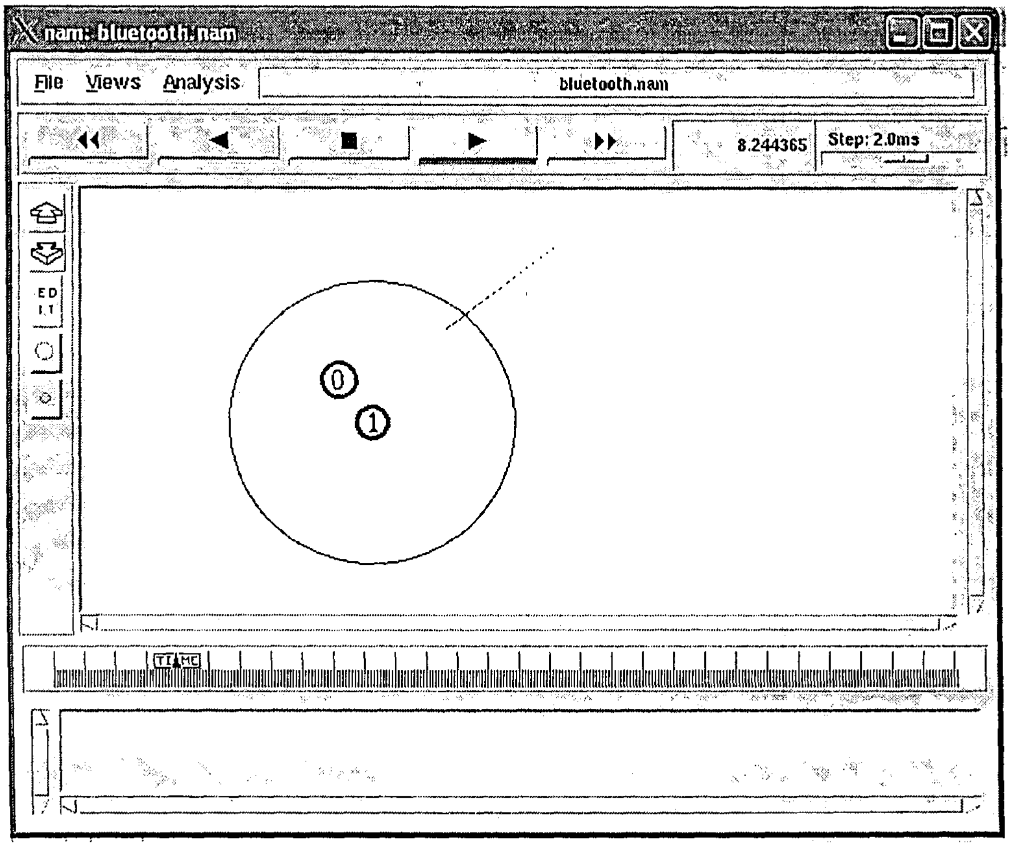


Figure A.20 Receiving of Data

Appendix-B

Code

B Classes Implemented for IPSec Based Bluetooth Security Architecture

We have divided our work into following main modules

B-1Network Simulator NS-2

Class TCP

This class is already implemented in NS-2 simulator; it was used and preferred as the TCP/IP stack which IPSec will be written over it.
TCP calss can be foud in the path.\...\...\ns-allinone-2.29\ns-2.29\tcp

TCL class

```
set ns [new Simulator]
set tf [open tcp.tr w]
$ns trace-all $tf

set nf [open tcp.nam w]
$ns namtrace-all-wireless $nf 7 7

Simulator set MacTrace_ ON
Simulator set RouterTrace_ ON

$ns node-config -macType Mac/BNEP

set node(0) [$ns node 0]
set node(1) [$ns node 1]

$node(0) set-statist 10 30 1
$node(1) set-statist 10 30 1

$node(0) rt AODV
$node(1) rt AODV

$node(0) LossMod BlueHoc
 $node(0) trace-all-NULL on
 $node(0) trace-all-POLL on

$ns at 0.0002 "$node(0) on"
$ns at 0.0005 "$node(1) on"

set tcp0 [new Agent/TCP]
$ns attach-agent $node(0) $tcp0
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0
```

```
set null0 [new Agent/TCPSink]
$ns attach-agent $node(1) $null0

$tcp0 set packetSize_ 2000
$ns connect $tcp0 $null0

set nscmd "$ftp0 start"

[$node(1) set l2cap_] set ifq_limit_ 30
[$node(0) set l2cap_] set ifq_limit_ 40

set ifq_ [new Queue/DropTail]
$ifq_ set limit_ 20

$ns at 0.01 "$node(0) make-bnep-connection $node(1) DH5 DH3 noqos $ifq_
$nscmd"

$ns at 30.1 "finish"

proc finish {} {
    global node ns nf tf
      $node(0) print-all-stat
      $node(1) print-all-stat
    $close nf
    exit 0
}

$ns run
```

B-2 IPSec

There needed to be a mechanism, which allowed us to add IPsec functionality to TCP/IP stack
The IPsec class is the central part, which does the whole standard conform processing of the incoming and outgoing IP traffic. It uses a set of databases (SPD and SAD) to determine the flow of the IP packets. The main processing is then done in the AH and ESP module.
A small cryptographic library contains all the functionality used to encrypt, decrypt or to authenticate the packets.

Class IPSec

```
#include "ipsec/debug.h"
```

```
#include "ipsec/ipsec.h"
#include "ipsec/util.h"
#include "ipsec/sa.h"
#include "ipsec/ah.h"
#include "ipsec/esp.h"

/**
 * IPsec input processing
 *
 * This function is called by the before BNEP and after TCP/IP  when a packet arrives
having AH or ESP in the
 * protocol field. A SA lookup gets the appropriate SA which is then passed to the packet
processing
 * funciton ipsec_ah_check() or ipsec_esp_decapsulate(). After successfully processing
an IPsec packet
 * an check together with an SPD lookup verifies if the packet was processed acording
the right SA.
 *
 * @param  packet         pointer used to access the intercepted original packet
 * @param  packet_size    length of the intercepted packet
 * @param  payload_offset pointer used to return offset of the new IP packet relative to
original packet pointer
 * @param  payload_size   pointer used to return total size of the new IP packet
 * @param  databases      Collection of all security policy databases for the active IPsec
device
 * @return int                    return status code
 */
int ipsec_input(unsigned char *packet, int packet_size,
           int *payload_offset, int *payload_size,
                            db_set_netif *databases)
{
        int ret_val      = IPSEC_STATUS_NOT_INITIALIZED;   /* by default, the
return value is undefined */
        int dummy        = packet_size;                          /* dummy
operation to avoid compiler warnings */
        sad_entry            *sa ;
        spd_entry            *spd ;
        ipsec_ip_header      *ip ;
        ipsec_ip_header      *inner_ip ;
        __u32                spi ;

        IPSEC_LOG_TRC(IPSEC_TRACE_ENTER,
                "ipsec_input",
                            ("*packet=%p, packet_size=%d, len=%u,
*payload_offset=%d, *payload_size=%d databases=%p",
                              (void *)packet, packet_size, (int)*payload_offset,
(int)*payload_size, (void *)databases)
                                   );

      IPSEC_DUMP_BUFFER(" INBOUND ESP or AH:", packet, 0, packet_size);

      ip = (ipsec_ip_header*)packet ;
      spi = ipsec_sad_get_spi(ip) ;
      sa = ipsec_sad_lookup(ip->dest, ip->protocol, spi, &databases->inbound_sad) ;

      if(sa == NULL)
      {
            IPSEC_LOG_AUD("ipsec_input", IPSEC_AUDIT_FAILURE, ("no
matching SA found")) ;
            IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input", ("return =
%d", IPSEC_STATUS_FAILURE) );
            return IPSEC_STATUS_FAILURE;
      }

      if(sa->mode != IPSEC_TUNNEL)
      {
            IPSEC_LOG_ERR("ipsec_input", IPSEC_STATUS_FAILURE,
("unsupported transmission mode (only IPSEC_TUNNEL is supported)") );
            IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input", ("return =
%d", IPSEC_STATUS_FAILURE) );
            return IPSEC_STATUS_FAILURE;
      }

      if(sa->protocol == IPSEC_PROTO_AH)
      {
            ret_val = ipsec_ah_check((ipsec_ip_header *)packet, payload_offset,
payload_size, sa);
            if(ret_val != IPSEC_STATUS_SUCCESS)
            {
                  IPSEC_LOG_ERR("ipsec_input", ret_val, ("ah_packet_check()
failed") );
                  IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input",
("ret_val=%d", ret_val) );
                  return ret_val;
            }

      } else if (sa->protocol == IPSEC_PROTO_ESP)
      {
            ret_val = ipsec_esp_decapsulate((ipsec_ip_header *)packet,
payload_offset, payload_size, sa);
            if(ret_val != IPSEC_STATUS_SUCCESS)
```

```
                {
                        IPSEC_LOG_ERR("ipsec_input", ret_val,
("ipsec_esp_decapsulate() failed") );
                        IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input",
("ret_val=%d", ret_val) );
                        return ret_val;
                }

        } else
        {
                IPSEC_LOG_ERR("ipsec_input", IPSEC_STATUS_FAILURE, ("invalid
protocol from SA") );
                IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input",
("ret_val=%d", IPSEC_STATUS_FAILURE) );
                return IPSEC_STATUS_FAILURE;
        }

        inner_ip = (ipsec_ip_header *)(((unsigned char *)ip) + *payload_offset) ;

        spd = ipsec_spd_lookup(inner_ip, &databases->inbound_spd) ;
        if(spd == NULL)
        {
                IPSEC_LOG_AUD("ipsec_input", IPSEC_AUDIT_FAILURE, ("no
matching SPD found")) ;
                IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input",
("ret_val=%d", IPSEC_STATUS_FAILURE) );
                return IPSEC_STATUS_FAILURE;
        }

        if(spd->policy == POLICY_APPLY)
        {
                if(spd->sa != sa)
                {
                        IPSEC_LOG_AUD("ipsec_input",
IPSEC_AUDIT_SPI_MISMATCH, ("SPI mismatch") );
                        IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input",
("return = %d", IPSEC_AUDIT_SPI_MISMATCH) );
                        return IPSEC_STATUS_FAILURE;
                }
        }
        else
        {
                        IPSEC_LOG_AUD("ipsec_input",
IPSEC_AUDIT_POLICY_MISMATCH, ("matching SPD does not permit IPsec
processing") );
```

```
                        IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input",
("return = %d", IPSEC_STATUS_FAILURE) );
                        return IPSEC_STATUS_FAILURE;
        }

        IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_input", ("return = %d",
IPSEC_STATUS_SUCCESS) );
        return IPSEC_STATUS_SUCCESS;
}


/**
 * IPsec output processing
 *
 * This function is called when outbound packets need IPsec processing. Depending the
SA, passed via
 * the SPD entry ipsec_ah_check() and ipsec_esp_encapsulate() is called to encapsulate
the packet in a
 * IPsec header.
 *
 * @param  packet         pointer used to access the intercepted original packet
 * @param  packet_size    length of the intercepted packet
 * @param  payload_offset pointer used to return offset of the new IP packet relative to
original packet pointer
 * @param  payload_size   pointer used to return total size of the new IP packet
 * @param  src            IP address of the local tunnel start point (external IP address)
 * @param  dst            IP address of the remote tunnel end point (external IP address)
 * @param  spd            pointer to security policy database where the rules for IPsec
processing are stored
 * @return int                    return status code
 */
int ipsec_output(unsigned char *packet, int packet_size, int *payload_offset, int
*payload_size,
            __u32 src, __u32 dst, spd_entry *spd)
{
        int ret_val = IPSEC_STATUS_NOT_INITIALIZED;                /* by default,
the return value is undefined */
        ipsec_ip_header                *ip ;

        IPSEC_LOG_TRC(IPSEC_TRACE_ENTER,
                "ipsec_output",
                                ("*packet=%p, packet_size=%d, len=%u,
*payload_offset=%d, *payload_size=%d src=%lx dst=%lx *spd=%p",
                        (void *)packet, packet_size, *payload_offset, *payload_size,
(__u32) src, (__u32) dst, (void *)spd)
                                );
        ip = (ipsec_ip_header*)packet;

        if((ip == NULL) || (ipsec_ntohs(ip->len) > packet_size))
        {
                IPSEC_LOG_DBG("ipsec_output",
IPSEC_STATUS_NOT_IMPLEMENTED, ("bad packet ip=%p, ip->len=%d (must not
be >%d bytes)", (void *)ip, ipsec_ntohs(ip->len), packet_size) );

                IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_output", ("return =
%d", IPSEC_STATUS_BAD_PACKET) );
           return IPSEC_STATUS_BAD_PACKET;
        }

        if((spd == NULL) || (spd->sa == NULL))
        {
                /** @todo invoke IKE to generate a proper SA for this SPD entry */
                IPSEC_LOG_DBG("ipsec_output",
IPSEC_STATUS_NOT_IMPLEMENTED, ("unable to generate dynamically an SA (IKE
not implemented)") );

                IPSEC_LOG_AUD("ipsec_output", IPSEC_STATUS_NO_SA_FOUND,
("no SA or SPD defined")) ;
                IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_output", ("return =
%d", IPSEC_STATUS_NO_SA_FOUND) );
           return IPSEC_STATUS_NO_SA_FOUND;
        }

        switch(spd->sa->protocol) {
                case IPSEC_PROTO_AH:
                                IPSEC_LOG_MSG("ipsec_output", ("have to encapsulate
an AH packet")) ;
                                ret_val = ipsec_ah_encapsulate((ipsec_ip_header *)packet,
payload_offset, payload_size, spd->sa, src, dst);

                                if(ret_val != IPSEC_STATUS_SUCCESS)
                                {
                                        IPSEC_LOG_ERR("ipsec_output", ret_val,
("ipsec_ah_encapsulate() failed"));
                                }
                        break;

                case IPSEC_PROTO_ESP:
                                IPSEC_LOG_MSG("ipsec_output", ("have to encapsulate
an ESP packet")) ;
```

```
                    ret_val = ipsec_esp_encapsulate((ipsec_ip_header *)packet,
payload_offset, payload_size, spd->sa, src, dst);

                    if(ret_val != IPSEC_STATUS_SUCCESS)
                    {
                         IPSEC_LOG_ERR("ipsec_output", ret_val,
("ipsec_esp_encapsulate() failed"));
                    }
               break;

          default:
                    ret_val = IPSEC_STATUS_BAD_PROTOCOL;
                    IPSEC_LOG_ERR("ipsec_output", ret_val, ("unsupported
protocol '%d' in spd->sa->protocol", spd->sa->protocol));
     }

     IPSEC_LOG_TRC(IPSEC_TRACE_RETURN, "ipsec_output", ("ret_val=%d",
ret_val) );
     return ret_val;
}
```

Class AH (IP Authentication Header)

The AH functions are used to authenticate IPsec traffic. All functions work in-place (i.g. manipulate directly the original packet without copying any data). For the encapsulation routine, the caller must ensure that space for the new IP and AH headers are available in front of the packet.

Definition in file **ah.c**.

```
#include <string.h>
#include "ipsec/ipsec.h"
#include "ipsec/util.h"
#include "ipsec/debug.h"
#include "ipsec/sa.h"
#include "ipsec/md5.h"
#include "ipsec/sha1.h"
#include "ipsec/ah.h"
```

Functions

Int ipsec_ah_check (ipsec_ip_header *outer_packet, int *payload_offset, int *payload_size, sad_entry *sa)

Int ipsec_ah_encapsulate (ipsec_ip_header *inner_packet, int *payload_offset, int *payload_size, sad_entry *sa, __u32 src, __u32 dst)

Variables

__u32 ipsec_ah_bitmap = 0
__u32 ipsec_ah_lastSeq = 0

Function Documentation

Int ipsec_ah_check (ipsec_ip_header * outer_packet, int * payload_offset, int * payload_size, sad_entry * sa)

It checks AH header and ICV (RFC 2402). Mutable fields of the outer IP header are set to zero prior to the ICV calculation.

Parameters:

- Outer_packet: pointer used to access the (outer) IP packet which hast to be checked.
- payload_offset: pointer used to return offset of inner (original) IP packet relative to the start of the outer header.
- payload_size: pointer used to return total size of the inner (original) IP packet.
- SA: pointer to security association holding the secret authentication key

Returns:

- IPSEC_STATUS_SUCCESS: packet could be authenticated.
- IPSEC_STATUS_FAILURE: packet is corrupted or ICV does not match.
- IPSEC_STATUS_NOT_IMPLEMENTED: invalid mode (only IPSEC_TUNNEL mode is implemented).

Int ipsec_ah_encapsulate (ipsec_ip_header * inner_packet, int * payload_offset, int * payload_size, sad_entry * sa, __u32 src, __u32 dst)

It adds AH and outer IP header calculates ICV (RFC 2402).

Parameters:

- inner_packet: pointer used to access the (outer) IP packet which hast to be checked.
- payload_offset: pointer used to return offset of inner (original) IP packet relative to the start of the outer header.
- payload_size: pointer used to return total size of the inner (original) IP packet.
- Src: IP address of the local tunnel start point (external IP address).
- DST: IP address of the remote tunnel end point (external IP address).
- SA: pointer to security association holding the secret authentication key.

Returns:

- IPSEC_STATUS_SUCCESS packet could be authenticated.
- IPSEC_STATUS_FAILURE packet is corrupted or ICV does not match.
- IPSEC_STATUS_NOT_IMPLEMENTED invalid mode (only IPSEC_TUNNEL mode is implemented).

Variable Documentation

- __u32 ipsec_ah_bitmap = 0 (save session state to detect replays - must be 32 bits. Note: must be initialized with zero (0x00000000) when a new SA is established).
- __u32 ipsec_ah_lastSeq = 0 (save session state to detect replays Note: must be initialized with zero (0x00000000) when a new SA is established).

Class ESP (Encapsulating Security Protocol)

This module contains the Encapsulating Security Payload code. All functions work in-place (i.g. mainipulate directly the original packet without copying any data). For the encapsulation routine, the caller must ensure that space for the new IP and ESP header are available in front of the packet.

Definition in file esp.c.

```
#include <string.h>
#include "ipsec/ipsec.h"
#include "ipsec/util.h"
#include "ipsec/debug.h"
#include "ipsec/sa.h"
#include "ipsec/des.h"
#include "ipsec/md5.h"
#include "ipsec/sha1.h"
#include "ipsec/esp.h"
```

Functions
__u8 ipsec_esp_get_padding (int len)
ipsec_status ipsec_esp_decapsulate (ipsec_ip_header *packet, int *offset, int *len, sad_entry *sa)
ipsec_status ipsec_esp_encapsulate (ipsec_ip_header *packet, int *offset, int *len, sad_entry *sa, __u32 src_addr, __u32 dest_addr)

Variables
__u32 ipsec_esp_bitmap = 0
__u32 ipsec_esp_lastSeq = 0

Function Documentation

ipsec_status ipsec_esp_decapsulate (ipsec_ip_header * packet, int * offset, int * len, sad_entry * sa)

It decapsulates an IP packet containing an ESP header.

Parameters:

- Packet: pointer to the ESP header.
- Offset: pointer to the offset which is passed back.
- Len: pointer to the length of the decapsulated packet.
- Sa: pointer to the SA.

Returns:

- IPSEC_STATUS_SUCCESS: if the packet could be decapsulated properly
- IPSEC_STATUS_FAILURE: if the SA's authentication algorithm was invalid or if ICV comparison failed
- IPSEC_STATUS_BAD_PACKET: if the decryption gave back a strange packet

Ipsec_status ipsec_esp_encapsulate (ipsec_ip_header * packet, int * offset, int * len, sad_entry * sa, __u32 src_addr, __u32 dest_addr)

It encapsulates an IP packet into an ESP packet which will again be added to an IP packet.

Parameters:

- Packet pointer to the IP packet.
- Offset: pointer to the offset which will point to the new encapsulated packet.
- Len: pointer to the length of the new encapsulated packet.
- SA: pointer to the SA.
- src_addr: source IP address of the outer IP header.
- dest_addr: destination IP address of the outer IP header.

Returns:

- IPSEC_STATUS_SUCCESS if the packet was properly encapsulated.
- IPSEC_STATUS_TTL_EXPIRED if the TTL expired.
- IPSEC_STATUS_FAILURE if the SA contained a bad authentication algorithm.

__u8 ipsec_esp_get_padding (int len)
It returns the number of padding needed for a certain ESP packet size.

Parameters:

- Len: the length of the packet

Returns:

- The length of padding needed

Variable Documentation

- __u32 ipsec_esp_bitmap = 0 (save session state to detect replays - must be 32 bits. Note: must be initialized with zero (0x00000000) when a new SA is established).
- __u32 ipsec_esp_lastSeq = 0 (save session state to detect replays Note: must be initialized with zero (0x00000000) when a new SA is established).

Class SA (security Association)

This module contains the Security Association code. Here we implement the Security Association concept from RFC 2401. Both SPD and SAD are implemented. At the time we do not support IKE and SA bundling. For having maximum flexibility two physically different tables (SPD and SAD) were implemented. They both provide functions to manipulate the database during run-time, so that a later IKE or SA-bundling could be implemented.

The SPD contains the selector fields on which each IP packet needs to be checked. After outbound packets found their SPD entry, they can access the SA via the SA pointer. Inbound packets can access their SA directly by applying the SPI to the SAD (by performing an SAD lookup). Each IPsec enabled device needs to have its own set of SPD and SAD for each, inbound and outbound processing.

Definition in file sa.c.

```
#include <string.h>
#include "ipsec/debug.h"
#include "ipsec/util.h"
#include "ipsec/sa.h"
#include "ipsec/ah.h"
#include "ipsec/esp.h"
```

Data Structures
Struct ipsec_in_ip_struct

Typedefs
Typedef ipsec_in_ip_struct ipsec_in_ip

Functions
db_set_netif * ipsec_spd_load_dbs (spd_entry *inbound_spd_data, spd_entry *outbound_spd_data, sad_entry *inbound_sad_data, sad_entry *outbound_sad_data)
ipsec_status ipsec_spd_release_dbs (db_set_netif *dbs) .
spd_entry * ipsec_spd_get_free (spd_table *table) .
spd_entry * ipsec_spd_add (__u32 src, __u32 src_net, __u32 dst, __u32 dst_net, _u8 proto, __u16 src_port, __u16 dst_port, __u8 policy, spd_table *table) .
ipsec_status ipsec_spd_add_sa (spd_entry *entry, sad_entry *sa) .
ipsec_status ipsec_spd_del (spd_entry *entry, spd_table *table) .
spd_entry * ipsec_spd_lookup (ipsec_ip_header *header, spd_table *table) .
void ipsec_spd_print_single (spd_entry *entry) .

void ipsec_spd_print (spd_table *table) .
sad_entry * ipsec_sad_get_free (sad_table *table).
sad_entry * ipsec_sad_add (sad_entry *entry, sad_table *table).
ipsec_status ipsec_sad_del (sad_entry *entry, sad_table *table).
sad_entry * ipsec_sad_lookup (__u32 dest, __u8 proto, __u32 spi, sad_table *table).
void ipsec_sad_print_single (sad_entry *entry).
void ipsec_sad_print (sad_table *table) .
__u32 ipsec_sad_get_spi (ipsec_ip_header *header).
ipsec_status ipsec_spd_flush (spd_table *table, spd_entry *def_entry).
ipsec_status ipsec_sad_flush (sad_table *table).

Variables
db_set_netif db_sets [IPSEC_NR_NETIFS]

Typedef Documentation
Typedef struct ipsec_in_ip_struct ipsec_in_ip: IPsec in IP structure - used to access headers inside SA .

Function Documentation

sad_entry* ipsec_sad_add (sad_entry * entry, sad_table * table)

It adds a Security Association to SA table.The SA entries is added to a statically allocated array of SAD structs. The size is defined by IPSEC_MAX_SAD_ENTRIES, so there cannot be added more entries added as this constant. The order of the entries within the table is not the same as the order within the array. The "table functionality" is implemented in a linked-list, so one must follow the links of the structure to get to the next entry.

Parameters:
- Entry: pointer to the SA structure which will be copied into the table.
- Table: pointer to the table where the SA is added.

Returns:
- A pointer to the added entry when adding was successful.
- NULL when the entry could not have been added (no free entry or duplicate).

Ipsec_status ipsec_sad_del (sad_entry * entry, sad_table * table)

It deletes a Security Association from an SA table.This function is simple. If the pointer is within the range of the table, then the entry is cleared. If the pointer does not match, nothing happens.

Parameters:

- Entry: Pointer to the SA entry which needs to be deleted.
- Table: Pointer to the SA table.

Returns:

- IPSEC_STATUS_SUCCESS: entry was deleted properly.
- IPSEC_STATUS_FAILURE: entry could not be deleted because not found, or invalid pointer.

ipsec_status ipsec_sad_flush (sad_table * table)

Parameters:

- Table: pointer to the SAD table.

Returns:

- IPSEC_STATUS_SUCCESS: if the flush was successful

sad_entry* ipsec_sad_get_free (sad_table * table)

It gives back a pointer to the next free entry from the given SA table.

Parameters:

- Table: pointer to the SA table.

Returns:

- Pointer to the free entry if one was found.
- NULL if no free entry was found

__u32 ipsec_sad_get_spi (ipsec_ip_header * header)

Parameters:

- Header: pointer to the IP header having an IPsec header as payload.

Returns:

- The SPI if one could be extracted
- 0 if no SPI could be extracted (not IPsec packet)

sad_entry* ipsec_sad_lookup (__u32 dest, __u8 proto, __u32 spi, sad_table * table)

It gives back a pointer to a SA matching the SA selectors.For incoming packets the IPsec packet must be checked against the inbound SAD and for outgoing packets the packet must be checked against the outbound SAD. It Implements simply by loops over all entries and returns the first match.

Parameters:

- Dest: destination IP address.
- Proto: IPsec protocol.
- Spi: Security Parameters Index.
- Table: pointer to the SAD table.

Returns:

- Pointer to the SA entry if one matched.
- NULL if no matching entry was found

void ipsec_sad_print (sad_table * table)

Parameters:

- Table: pointer to the SAD table which will be printed

Returns:

- Void

void ipsec_sad_print_single (sad_entry * entry)

Parameters:

- Entry: pointer to the SA entry which will be printed

Returns:

- Void

Ipsec_status ipsec_spd_add_sa (spd_entry * entry, sad_entry * sa)

It adds a Security Association to a Security Police.

Parameters:

- Entry: pointer to the SPD entry where the SA should be added.
- Sa: a pointer to the SA which is added to the SPD

Returns:

- IPSEC_STATUS_SUCCESS: the entry was added successfully.

Ipsec_status ipsec_spd_del (spd_entry * entry, spd_table * table)

It deletes a Security Policy from an SPD table.This function is simple. If the pointer is within the range of the table, then the entry is cleared. If the pointer does not match, nothing happens.

Parameters:

- Entry: Pointer to the SPD entry which needs to be deleted.
- Table: Pointer to the SPD table.

Returns:

- IPSEC_STATUS_SUCCESS: entry was deleted properly.
- IPSEC_STATUS_FAILURE: entry could not be deleted because not found, or invalid pointer.

Ipsec_status ipsec_spd_flush (spd_table * table, spd_entry * def_entry)

It flushes SPD table and sets a new default entry.

Parameters:

- Table: pointer to the SPD table
- def_entry: pointer to the default entry.

Returns:

- IPSEC_STATUS_SUCCESS: if the flush was successful.
- IPSEC_STATUS_FAILURE: if the flush failed.

Db_set_netif* ipsec_spd_load_dbs (spd_entry * inbound_spd_data, spd_entry * outbound_spd_data, sad_entry * inbound_sad_data, sad_entry * outbound_sad_data)

This function initializes the database set, allocated in a per-network manner.The data which is passed by the pointers should not be used by other functions except the ones of the SA-module.

The data passed can be viewed as a place where the SA-module can store its data (Security Policies or Security Associations). The tables which are passed to the function can already be filled up with static configuration data. You can use the SPD_ENTRY and the SAD_ENTRY macro to do this in a nice way.

Parameters:

- inbound_spd_data pointer: to a table where inbound Security Policies will be stored
- outbound_spd_data pointer: to a table where outbound Security Policies will be stored
- inbound_sad_data pointer: to a table where inbound Security Associations will be stored
- outbound_sad_data pointer: to a table where outbound Security Associations will be stored

Returns:

- Pointer to the initialized set of DB's if the setup was successful.
- NULL if loading failed.

spd_entry* ipsec_spd_lookup (ipsec_ip_header * header, spd_table * table)

It returns a pointer to an SPD entry which matches the packet. Inbound packets must be checked against the inbound SPD and outbound packets must be checked against the outbound SPD.

Parameters:

- Header: Pointer to an IP packet which is checked.
- Table: Pointer to the SPD inbound/outbound table.

Returns:

- Pointer to the matching SPD entry.
- NULL if no entry matched.

ipsec_status ipsec_spd_release_dbs (db_set_netif * dbs)

This function is used to release the structure allocated in ipsec_spd_load_dbs(). The tables which were allocated in ipsec_spd_load_dbs() can now be freely used.

Parameters:

- Dbs pointer: to the set of databases got by ipsec_spd_load_dbs() which has to be released.

Returns:

- IPSEC_STATUS_SUCCESS: if release was successful.
- IPSEC_STATUS_FAILURE: if release was not successful.

Variable Documentation

- db_set_netif db_sets[IPSEC_NR_NETIFS] (This structure holds sets of databases used by one network interface. Each successful call of ipsec_spd_load_dbs() will return a pointer to an entry of this structure array. One entry holds pointers to a inbound and outbound SPD and SAD table.)

SHA1 (US Secure Hash Algorithm)

Definition in file sha1.c.

```
#include <string.h>
#include "ipsec/sha1.h"
#include "ipsec/debug.h"
```

Defines

```
#define Xupdate(a, ix, ia, ib, ic, id)
#define SHA_CBLOCK (SHA_LBLOCK*4)
```

```
#define SHA_LAST_BLOCK (SHA_CBLOCK-8)
#define SHA_LBLOCK (SHA_CBLOCK/4)
#define HOST_c2l(c, l)
#define HOST_p_c2l(c, l, n)
#define HOST_p_c2l_p(c, l, sc, len)
#define HOST_c2l_p(c, l, n)
#define HOST_l2c(l, c)
#define INIT_DATA_h0 0x67452301UL
#define INIT_DATA_h1 0xefcdab89UL
#define INIT_DATA_h2 0x98badcfeUL
#define INIT_DATA_h3 0x10325476UL
#define INIT_DATA_h4 0xc3d2e1f0UL
#define K_00_19 0x5a827999UL
#define K_20_39 0x6ed9eba1UL
#define K_40_59 0x8f1bbcdcUL
#define K_60_79 0xca62c1d6UL
#define F_00_19(b, c, d) ((((c) ^ (d)) & (b)) ^ (d))
#define F_20_39(b, c, d) ((b) ^ (c) ^ (d))
#define F_40_59(b, c, d) (((b) & (c)) | (((b)|(c)) & (d)))
#define F_60_79(b, c, d) F_20_39(b,c,d)
#define BODY_00_15(i, a, b, c, d, e, f, xi)
#define BODY_16_19(i, a, b, c, d, e, f, xi, xa, xb, xc, xd)
#define BODY_20_31(i, a, b, c, d, e, f, xi, xa, xb, xc, xd)
#define BODY_32_39(i, a, b, c, d, e, f, xa, xb, xc, xd)
#define BODY_40_59(i, a, b, c, d, e, f, xa, xb, xc, xd)
#define BODY_60_79(i, a, b, c, d, e, f, xa, xb, xc, xd)
#define X(i) XX##i
```

Functions

```
unsigned char * SHA1 (const unsigned char *d, unsigned long n, unsigned char *md)
void sha1_block_host_order (SHA_CTX *c, const void *p, int num)
void sha1_block_data_order (SHA_CTX *c, const void *p, int num)
void SHA1_Update (SHA_CTX *c, const void *data_, unsigned long len)
void SHA1_Transform (SHA_CTX *c, const unsigned char *data)
void SHA1_Final (unsigned char *md, SHA_CTX *c)
void SHA1_Init (SHA_CTX *c)
void hmac_sha1 (unsigned char *text, int text_len, unsigned char *key, int key_len, unsigned char *digest)
```

Define Documentation

```
#define BODY_00_15 ( i, a, b, c, d, e, f, xi )

#define BODY_16_19 ( i, a, b, c, d, e, f, xi, xa, xb, xc, xd )
Value:
Xupdate(f,xi,xa,xb,xc,xd); \
(f)+=(e)+K_00_19+ROTATE((a),5)+F_00_19((b),(c),(d)); \
(b)=ROTATE((b),30);

#define BODY_20_31 ( i, a, b, c, d, e, f, xi, xa, xb, xc, xd )
Value:
Xupdate(f,xi,xa,xb,xc,xd); \
(f)+=(e)+K_20_39+ROTATE((a),5)+F_20_39((b),(c),(d)); \
(b)=ROTATE((b),30);

#define BODY_32_39 ( i, a, b, c, d, e, f, xa, xb, xc, xd )
Value:
Xupdate(f,xa,xa,xb,xc,xd); \
(f)+=(e)+K_20_39+ROTATE((a),5)+F_20_39((b),(c),(d)); \
(b)=ROTATE((b),30);

#define BODY_40_59 ( i, a, b, c, d, e, f, xa, xb, xc, xd )
Value:
Xupdate(f,xa,xa,xb,xc,xd); \
(f)+=(e)+K_40_59+ROTATE((a),5)+F_40_59((b),(c),(d)); \
(b)=ROTATE((b),30);

#define BODY_60_79 ( i, a, b, c, d, e, f, xa, xb, xc, xd )
Value:
Xupdate(f,xa,xa,xb,xc,xd); \
(f)=xa+(e)+K_60_79+ROTATE((a),5)+F_60_79((b),(c),(d)); \
(b)=ROTATE((b),30);

#define F_00_19 ( b, c, d )   ((((c) ^ (d)) & (b)) ^ (d))
#define F_20_39 ( b, c, d )   ((b) ^ (c) ^ (d))
#define F_40_59 ( b, c, d )   (((b) & (c)) | (((b)|(c)) & (d)))
#define F_60_79 ( b, c, d )   F_20_39(b,c,d)

#define HOST_c2l ( c, l )
Value:
(l =(((unsigned long)(*((c)++)))<<24),       \
l|=(((unsigned long)(*((c)++)))<<16),        \
l|=(((unsigned long)(*((c)++)))<< 8),        \
l|=(((unsigned long)(*((c)++)))   ),        \
l)

#define HOST_c2l_p ( c, l, n   )
Value:
{                        \
            l=0; (c)+=n;                         \
            switch (n) {                         \
            case 3: l =((unsigned long)(*(--(c))))<< 8;    \
            case 2: l|=((unsigned long)(*(--(c))))<<16;    \
            case 1: l|=((unsigned long)(*(--(c))))<<24;    \
                } }

#define HOST_l2c ( l, c   )
Value:
(*((c)++)=(unsigned char)(((l)>>24)&0xff),       \
            *((c)++)=(unsigned char)(((l)>>16)&0xff),      \
            *((c)++)=(unsigned char)(((l)>> 8)&0xff),      \
            *((c)++)=(unsigned char)(((l)    )&0xff),      \
            l)

#define HOST_p_c2l ( c, l, n   )
 Value:
{                        \
            switch (n) {                         \
            case 0: l =((unsigned long)(*((c)++)))<<24;    \
            case 1: l|=((unsigned long)(*((c)++)))<<16;    \
            case 2: l|=((unsigned long)(*((c)++)))<< 8;    \
            case 3: l|=((unsigned long)(*((c)++)));        \
                } }

#define HOST_p_c2l_p ( c, l, sc, len   )
Value:
{                        \
            switch (sc) {                        \
            case 0: l =((unsigned long)(*((c)++)))<<24;    \
                if (--len == 0) break;           \
            case 1: l|=((unsigned long)(*((c)++)))<<16;    \
                if (--len == 0) break;           \
            case 2: l|=((unsigned long)(*((c)++)))<< 8;    \
                } }
```

```
#define INIT_DATA_h0  0x67452301UL
#define INIT_DATA_h1  0xefcdab89UL
#define INIT_DATA_h2  0x98badcfeUL
#define INIT_DATA_h3  0x10325476UL
#define INIT_DATA_h4  0xc3d2e1f0UL
#define K_00_19  0x5a827999UL
#define K_20_39  0x6ed9eba1UL
#define K_40_59  0x8f1bbcdcUL
#define K_60_79  0xca62c1d6UL
#define SHA_CBLOCK   (SHA_LBLOCK*4)
#define SHA_LAST_BLOCK   (SHA_CBLOCK-8)
#define SHA_LBLOCK   (SHA_CBLOCK/4)
#define X ( i  )    XX##i
#define Xupdate (  a,   ix,   ia,  ib,   ic,  id   )
```

Function Documentation

void hmac_sha1 (unsigned char * text, int text_len, unsigned char * key, int key_len, unsigned char * digest)
unsigned char* SHA1 (const unsigned char * d, unsigned long n, unsigned char * md)
void sha1_block_data_order (SHA_CTX * c, const void * p, int num)
void sha1_block_host_order (SHA_CTX * c, const void * p, int num)
void SHA1_Final (unsigned char * md, SHA_CTX * c)
void SHA1_Init (SHA_CTX * c)
void SHA1_Transform (SHA_CTX * c, const unsigned char * data)
void SHA1_Update (SHA_CTX * c, const void * data_, unsigned long len)

Class MD5 (Message-Digest Algorithm)

Definition in file md5.c.

```
#include <string.h>
#include "ipsec/md5.h"
#include "ipsec/debug.h"
```

Defines

```
#define  INIT_DATA_A  (unsigned long)0x67452301L
#define  INIT_DATA_B  (unsigned long)0xefcdab89L
#define  INIT_DATA_C  (unsigned long)0x98badcfeL
#define  INIT_DATA_D  (unsigned long)0x10325476L
#define  HOST_c2l(c, l)
#define  HOST_p_c2l(c, l, n)
```

```
#define HOST_p_c2l_p(c, l, sc, len)
#define HOST_c2l_p(c, l, n)
#define HOST_l2c(l, c)
#define F(b, c, d)  ((((c) ^ (d)) & (b)) ^ (d))
#define G(b, c, d)  ((((b) ^ (c)) & (d)) ^ (c))
#define H(b, c, d)  ((b) ^ (c) ^ (d))
#define I(b, c, d)  (((~(d)) | (b)) ^ (c))
#define R0(a, b, c, d, k, s, t)
#define R1(a, b, c, d, k, s, t)
#define R2(a, b, c, d, k, s, t)
#define R3(a, b, c, d, k, s, t)
#define X(i)  XX##i
```

Functions

unsigned char * MD5 (const unsigned char *d, unsigned long n, unsigned char *md)
void MD5_Init (MD5_CTX *c)
void md5_block_host_order (MD5_CTX *c, const void *p, int num)
void md5_block_data_order (MD5_CTX *c, const void *p, int num)
void MD5_Update (MD5_CTX *c, const void *data_, unsigned long len)
void MD5_Transform (MD5_CTX *c, const unsigned char *data)
void MD5_Final (unsigned char *md, MD5_CTX *c)
void hmac_md5 (unsigned char *text, int text_len, unsigned char *key, int key_len, unsigned char *digest)

Define Documentation

```
#define F (b, c, d   )    ((((c) ^ (d)) & (b)) ^ (d))
#define G (b, c, d   )    ((((b) ^ (c)) & (d)) ^ (c))
#define H (b, c, d   )    ((b) ^ (c) ^ (d))

#define HOST_c2l ( c, l   )
Value:
(l =(((unsigned long)(*((c)++)))   ),          \
            l|=(((unsigned long)(*((c)++)))<< 8),          \
            l|=(((unsigned long)(*((c)++)))<<16),          \
            l|=(((unsigned long)(*((c)++)))<<24),          \
            l)

#define HOST_c2l_p ( c, l, n   )
Value:
{                        \
            l=0; (c)+=n;                        \
            switch (n) {                        \
            case 3: l =((unsigned long)(*(--(c))))<<16;    \
                    case 2: l|=((unsigned long)(*(--(c))))<< 8;    \
                    case 1: l|=((unsigned long)(*(--(c))));        \
                        } }

#define HOST_l2c ( l, c  )
Value:
(*((c)++)=(unsigned char)(((l)   )&0xff),     \
                    *((c)++)=(unsigned char)(((l)>> 8)&0xff),     \
                    *((c)++)=(unsigned char)(((l)>>16)&0xff),     \
                    *((c)++)=(unsigned char)(((l)>>24)&0xff),     \
                    l)

#define HOST_p_c2l ( c, l, n  )
Value:
{                        \
                    switch (n) {                   \
                    case 0: l =((unsigned long)(*((c)++)));        \
                    case 1: l|=((unsigned long)(*((c)++)))<< 8;    \
                    case 2: l|=((unsigned long)(*((c)++)))<<16;    \
                    case 3: l|=((unsigned long)(*((c)++)))<<24;    \
                        } }

#define HOST_p_c2l_p ( c, l, sc, len  )
 Value:
{                        \
                    switch (sc) {                  \
                    case 0: l =((unsigned long)(*((c)++)));        \
                        if (--len == 0) break;            \
                    case 1: l|=((unsigned long)(*((c)++)))<< 8;    \
                        if (--len == 0) break;            \
                    case 2: l|=((unsigned long)(*((c)++)))<<16;    \
                        } }

#define I ( b, c, d  )   (((~(d)) | (b)) ^ (c))
#define INIT_DATA_A   (unsigned long)0x67452301L
#define INIT_DATA_B   (unsigned long)0xefcdab89L
#define INIT_DATA_C   (unsigned long)0x98badcfeL
#define INIT_DATA_D   (unsigned long)0x10325476L

#define R0 ( a, b, c, d, k, s, t  )
Value:
{ \
    a+=((k)+(t)+F((b),(c),(d))); \
    a=ROTATE(a,s); \
    a+=b; };\
#define R1 ( a, b, c, d, k, s, t )
Value:
{ \
    a+=((k)+(t)+G((b),(c),(d))); \
    a=ROTATE(a,s); \
    a+=b; };

#define R2 ( a, b, c, d, k, s, t )
Value:
{ \
    a+=((k)+(t)+H((b),(c),(d))); \
    a=ROTATE(a,s); \
    a+=b; };

#define R3 ( a, b, c, d, k, s, t )
Value:
{ \
    a+=((k)+(t)+I((b),(c),(d))); \
    a=ROTATE(a,s); \
    a+=b; };

#define X ( i ) XX##i
```

Function Documentation

void hmac_md5 (unsigned char * text, int text_len, unsigned char * key, int key_len, unsigned char * digest)

Parameters:

- Text: pointer to data stream.
- text_len: length of data stream.
- Key: pointer to authentication key.
- key_len: length of authentication key
- Digest: caller digest to be filled in 128-bit.

Returns:
void

unsigned char* MD5 (const unsigned char * d, unsigned long n, unsigned char * md)
void md5_block_data_order (MD5_CTX * c, const void * p, int num)
void md5_block_host_order (MD5_CTX * c, const void * p, int num)
void MD5_Final (unsigned char * md, MD5_CTX * c)
void MD5_Init (MD5_CTX * c)

void MD5_Transform (MD5_CTX * c, const unsigned char * data)
void MD5_Update (MD5_CTX * c, const void * data_, unsigned long len)

Class DES and 3DES in CBC Mode

Definition in file des.c.

```
#include <string.h>
#include "ipsec/des.h"
#include "ipsec/debug.h"
```

Defines

```
#define DES_KEY_SZ  (sizeof(DES_cblock))
#define DES_SCHEDULE_SZ  (sizeof(DES_key_schedule))
#define ITERATIONS  16
#define HALF_ITERATIONS  8
#define c2l(c, l)
#define c2ln(c, l1, l2, n)
#define l2c(l, c)
#define HDRSIZE  4
#define n2l(c, l)
#define l2n(l, c)
#define l2cn(l1, l2, c, n)
#define ROTATE(a, n)  (((a)>>(n))+((a)<<(32-(n))))
#define LOAD_DATA_tmp(a, b, c, d, e, f)  LOAD_DATA(a,b,c,d,e,f,g)
#define LOAD_DATA(R, S, u, t, E0, E1, tmp)
#define D_ENCRYPT(LL, R, S)
#define PERM_OP(a, b, t, n, m)
#define IP(l, r)
#define FP(l, r)
#define NUM_WEAK_KEY  16
#define HPERM_OP(a, t, n, m)
```

Functions

```
void DES_cbc_encrypt (const unsigned char *input, unsigned char *output, long length, DES_key_schedule *schedule, DES_cblock *ivec, int enc)
void DES_ncbc_encrypt (const unsigned char *input, unsigned char *output, long length, DES_key_schedule *schedule, DES_cblock *ivec, int enc)
void DES_encrypt1 (DES_LONG *data, DES_key_schedule *ks, int enc)
void DES_encrypt2 (DES_LONG *data, DES_key_schedule *ks, int enc)
void DES_encrypt3 (DES_LONG *data, DES_key_schedule *ks1, DES_key_schedule *ks2, DES_key_schedule *ks3)
```

void DES_decrypt3 (DES_LONG *data, DES_key_schedule *ks1, DES_key_schedule *ks2, DES_key_schedule *ks3)
void DES_ede3_cbc_encrypt (const unsigned char *input, unsigned char *output, long length, DES_key_schedule *ks1, DES_key_schedule *ks2, DES_key_schedule *ks3, DES_cblock *ivec, int enc)
void DES_set_odd_parity (DES_cblock *key)
int DES_check_key_parity (const_DES_cblock *key)
int DES_is_weak_key (const_DES_cblock *key)
int DES_set_key (const_DES_cblock *key, DES_key_schedule *schedule)
int DES_key_sched (const_DES_cblock *key, DES_key_schedule *schedule)
int DES_set_key_checked (const_DES_cblock *key, DES_key_schedule *schedule)
void DES_set_key_unchecked (const_DES_cblock *key, DES_key_schedule *schedule)
void cipher_3des_cbc (unsigned char *text, int text_len, unsigned char *key, unsigned char *iv, int mode, unsigned char *output)

Variables

const DES_LONG DES_SPtrans [8][64]
int _shadow_DES_check_key
const unsigned char odd_parity [256]
DES_cblock weak_keys [NUM_WEAK_KEY]
const DES_LONG des_skb [8][64]

Define Documentation

```
#define c2l ( c, l )
Value:
(l =((DES_LONG)(*((c)++)))    , \
          l|=((DES_LONG)(*((c)++)))<< 8L, \
          l|=((DES_LONG)(*((c)++)))<<16L, \
          l|=((DES_LONG)(*((c)++)))<<24L)

#define c2ln ( c, l1, l2, n )
Value:
{ \
          c+=n; \
          l1=l2=0; \
          switch (n) { \
          case 8: l2 =((DES_LONG)(*(--(c))))<<24L; \
          case 7: l2|=((DES_LONG)(*(--(c))))<<16L; \
          case 6: l2|=((DES_LONG)(*(--(c))))<< 8L; \
          case 5: l2|=((DES_LONG)(*(--(c))));    \
          case 4: l1 =((DES_LONG)(*(--(c))))<<24L; \
          case 3: l1|=((DES_LONG)(*(--(c))))<<16L; \
          case 2: l1|=((DES_LONG)(*(--(c))))<< 8L; \
          case 1: l1|=((DES_LONG)(*(--(c))));    \
                } \
            }

#define D_ENCRYPT ( LL, R, S )
Value:
{\
    LOAD_DATA_tmp(R,S,u,t,E0,E1); \
    t=ROTATE(t,4); \
    LL^=\
    DES_SPtrans[0][(u>> 2L)&0x3f]^ \
        DES_SPtrans[2][(u>>10L)&0x3f]^ \
        DES_SPtrans[4][(u>>18L)&0x3f]^ \
        DES_SPtrans[6][(u>>26L)&0x3f]^ \
        DES_SPtrans[1][(t>> 2L)&0x3f]^ \
        DES_SPtrans[3][(t>>10L)&0x3f]^ \
        DES_SPtrans[5][(t>>18L)&0x3f]^ \
        DES_SPtrans[7][(t>>26L)&0x3f]; }

#define DES_KEY_SZ  (sizeof(DES_cblock))
#define DES_SCHEDULE_SZ  (sizeof(DES_key_schedule))

#define FP ( l, r )
Value:
{ \
    DES_LONG tt; \
    PERM_OP(l,r,tt, 1,0x55555555L); \
    PERM_OP(r,l,tt, 8,0x00ff00ffL); \
    PERM_OP(l,r,tt, 2,0x33333333L); \
    PERM_OP(r,l,tt,16,0x0000ffffL); \
    PERM_OP(l,r,tt, 4,0x0f0f0f0fL); \
    }

#define HALF_ITERATIONS  8
#define HDRSIZE  4

#define HPERM_OP ( a, t, n, m )
Value:
((t)=((((a)<<(16-(n)))^(a))&(m)),\
    (a)=(a)^(t)^(t>>(16-(n))))

#define IP ( l, r )
Value:
{ \
    DES_LONG tt; \
    PERM_OP(r,l,tt, 4,0x0f0f0f0fL); \
    PERM_OP(l,r,tt,16,0x0000ffffL); \
    PERM_OP(r,l,tt, 2,0x33333333L); \
    PERM_OP(l,r,tt, 8,0x00ff00ffL); \
    PERM_OP(r,l,tt, 1,0x55555555L); \
    }

#define ITERATIONS 16

#define l2c ( l, c )
Value:
(*((c)++)=(unsigned char)(((l)    )&0xff), \
                 *((c)++)=(unsigned char)(((l)>> 8L)&0xff), \
                 *((c)++)=(unsigned char)(((l)>>16L)&0xff), \
                 *((c)++)=(unsigned char)(((l)>>24L)&0xff))
#define l2cn ( l1, l2, c, n )
Value:
{ \
                c+=n; \
                switch (n) { \
                case 8: *(--(c))=(unsigned char)(((l2)>>24L)&0xff); \
                case 7: *(--(c))=(unsigned char)(((l2)>>16L)&0xff); \
                case 6: *(--(c))=(unsigned char)(((l2)>> 8L)&0xff); \
                case 5: *(--(c))=(unsigned char)(((l2)    )&0xff); \
                case 4: *(--(c))=(unsigned char)(((l1)>>24L)&0xff); \
                case 3: *(--(c))=(unsigned char)(((l1)>>16L)&0xff); \
                case 2: *(--(c))=(unsigned char)(((l1)>> 8L)&0xff); \
                case 1: *(--(c))=(unsigned char)(((l1)    )&0xff); \
                    } \
                }

#define l2n ( l, c )
Value:
(*((c)++)=(unsigned char)(((l)>>24L)&0xff), \
                 *((c)++)=(unsigned char)(((l)>>16L)&0xff), \
                 *((c)++)=(unsigned char)(((l)>> 8L)&0xff), \
                 *((c)++)=(unsigned char)(((l)    )&0xff))

#define LOAD_DATA ( R, S, u, t, E0, E1, tmp )
Value:
u=R^s[S ]; \
    t=R^s[S+1]

#define LOAD_DATA_tmp ( a, b, c, d, e, f )   LOAD_DATA(a,b,c,d,e,f,g)

#define n2l ( c, l )
Value:
(l =((DES_LONG)(*((c)++)))<<24L, \
            l|=((DES_LONG)(*((c)++)))<<16L, \
            l|=((DES_LONG)(*((c)++)))<< 8L, \
            l|=((DES_LONG)(*((c)++))))

#define NUM_WEAK_KEY   16

#define PERM_OP ( a,  b,  t,  n,  m  )
Value:
((t)=((((a)>>(n))^(b))&(m)),\
     (b)^=(t),\
     (a)^=((t)<<(n)))

#define ROTATE ( a,  n  )    (((a)>>(n))+((a)<<(32-(n))))
```

Function Documentation

void cipher_3des_cbc (unsigned char * text, int text_len, unsigned char * key, unsigned char * iv, int mode, unsigned char * output)

3DES-CBC function calculates a digest from a given data buffer and a given key.

Parameters:
- Text: pointer to input data.
- text_len: length of input data.
- Key: pointer to encryption key (192 bits).
- IV: initialization vector.
- Mode: defines whether encryption or decryption should be performed.
- output: en- or decrypted input data

Returns:
void

void DES_cbc_encrypt (const unsigned char * input, unsigned char * output, long length, DES_key_schedule * schedule, DES_cblock * ivec, int enc)
int DES_check_key_parity (const_DES_cblock * key)
void DES_decrypt3 (DES_LONG * data, DES_key_schedule * ks1, DES_key_schedule * ks2, DES_key_schedule * ks3)
void DES_ede3_cbc_encrypt (const unsigned char * input, unsigned char * output, long length, DES_key_schedule * ks1, DES_key_schedule * ks2, DES_key_schedule * ks3, DES_cblock * ivec, int enc)
void DES_encrypt1 (DES_LONG * data, DES_key_schedule * ks, int enc)
void DES_encrypt2 (DES_LONG * data, DES_key_schedule * ks, int enc)
void DES_encrypt3 (DES_LONG * data, DES_key_schedule * ks1, DES_key_schedule * ks2, DES_key_schedule * ks3)
int DES_is_weak_key (const_DES_cblock * key)
int DES_key_sched (const_DES_cblock * key, DES_key_schedule * schedule)

void DES_ncbc_encrypt (const unsigned char * input, unsigned char * output, long length, DES_key_schedule * schedule, DES_cblock * ivec, int enc)
int DES_set_key (const_DES_cblock * key, DES_key_schedule * schedule)
int DES_set_key_checked (const_DES_cblock * key, DES_key_schedule * schedule)
void DES_set_key_unchecked (const_DES_cblock * key, DES_key_schedule * schedule)
void DES_set_odd_parity (DES_cblock * key)

Variable Documentation

```
int _shadow_DES_check_key
const DES_LONG des_skb[8][64] [static]
const DES_LONG DES_SPtrans[8][64]

const unsigned char odd_parity[256] [static]
Initial value:
{
 1, 1, 2, 2, 4, 4, 7, 7, 8, 8, 11, 11, 13, 13, 14, 14,
 16, 16, 19, 19, 21, 21, 22, 22, 25, 25, 26, 26, 28, 28, 31, 31,
 32, 32, 35, 35, 37, 37, 38, 38, 41, 41, 42, 42, 44, 44, 47, 47,
 49, 49, 50, 50, 52, 52, 55, 55, 56, 56, 59, 59, 61, 61, 62, 62,
 64, 64, 67, 67, 69, 69, 70, 70, 73, 73, 74, 74, 76, 76, 79, 79,
 81, 81, 82, 82, 84, 84, 87, 87, 88, 88, 91, 91, 93, 93, 94, 94,
 97, 97, 98, 98,100,100,103,103,104,104,107,107,109,109,110,110,
112,112,115,115,117,117,118,118,121,121,122,122,124,124,127,127,
128,128,131,131,133,133,134,134,137,137,138,138,140,140,143,143,
145,145,146,146,148,148,151,151,152,152,155,155,157,157,158,158,
161,161,162,162,164,164,167,167,168,168,171,171,173,173,174,174,
176,176,179,179,181,181,182,182,185,185,186,186,188,188,191,191,
193,193,194,194,196,196,199,199,200,200,203,203,205,205,206,206,
208,208,211,211,213,213,214,214,217,217,218,218,220,220,223,223,
224,224,227,227,229,229,230,230,233,233,234,234,236,236,239,239,
241,241,242,242,244,244,247,247,248,248,251,251,253,253,254,254}

DES_cblock weak_keys[NUM_WEAK_KEY] [static]
Initial value:
{
	{0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01},
	{0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFE},
	{0x1F,0x1F,0x1F,0x1F,0x0E,0x0E,0x0E,0x0E},
	{0xE0,0xE0,0xE0,0xE0,0xF1,0xF1,0xF1,0xF1},

	{0x01,0xFE,0x01,0xFE,0x01,0xFE,0x01,0xFE},
	{0xFE,0x01,0xFE,0x01,0xFE,0x01,0xFE,0x01},
    {0x1F,0xE0,0x1F,0xE0,0x0E,0xF1,0x0E,0xF1},
    {0xE0,0x1F,0xE0,0x1F,0xF1,0x0E,0xF1,0x0E},
    {0x01,0xE0,0x01,0xE0,0x01,0xF1,0x01,0xF1},
    {0xE0,0x01,0xE0,0x01,0xF1,0x01,0xF1,0x01},
    {0x1F,0xFE,0x1F,0xFE,0x0E,0xFE,0x0E,0xFE},
    {0xFE,0x1F,0xFE,0x1F,0xFE,0x0E,0xFE,0x0E},
    {0x01,0x1F,0x01,0x1F,0x01,0x0E,0x01,0x0E},
    {0x1F,0x01,0x1F,0x01,0x0E,0x01,0x0E,0x01},
    {0xE0,0xFE,0xE0,0xFE,0xF1,0xFE,0xF1,0xFE},
    {0xFE,0xE0,0xFE,0xE0,0xFE,0xF1,0xFE,0xF1}}
```

B-3 UCBT

Class BNEP

```
/*
* Note -- The bridge function of GN/NAP is supposed to implement here.
 * We rather use the L3 approach to do it. That is, GN/NAP and BR are routers
 * instead of bridges. This design simplifies the simulator implementation
 * significantly, and we don't need to handle ARP for BT devices.
 * Bridge function presented here is quite minimum.
 */

#include "l2cap.h"
#include "bnep.h"
#include "packet.h"
#include "mac.h"
#include "gridkeeper.h"
#include "lmp-piconet.h"
#include "scat-form.h"

#define BUFFSIZE 1024

int hdr_bnep::offset_;

static class BNEPHeaderClass:public PacketHeaderClass {
 public:
  BNEPHeaderClass():PacketHeaderClass("PacketHeader/BNEP",
                                      sizeof(hdr_bnep)) {
      bind_offset(&hdr_bnep::offset_);
  }
} class_bnephdr;

static class BNEPClass:public TclClass {
 public:
  BNEPClass():TclClass("Mac/BNEP") {}
  TclObject *create(int, const char *const *) {
       return new BNEP();
  }
} class_bnep;

////////////////////////////////////////////////////
//              BridgeTable                      //
////////////////////////////////////////////////////
int BridgeTable::lookup(int ad)
{
  BrTableEntry *wk = _table;
  while (wk) {
       if (wk->addr == ad) {
          return wk->port;
       }
       wk = wk->next;
  }

  return -1;
}

void BridgeTable::dump()
{
  int cntr = 0;
  printf("\nRridge Table:\n");

  BrTableEntry *wk = _table;
  while (wk) {
       printf("%d:%d %f\n", wk->addr, wk->port, wk->ts);
       cntr++;
       wk = wk->next;
  }

  printf("Total %d entries.\n\n", cntr);
}

void BridgeTable::add(int ad, int p)
{
  BrTableEntry *wk = _table;
  while (wk) {
       if (wk->addr == ad) {
          wk->port = p;
          wk->ts = Scheduler::instance().clock();
          return;
       }
```

```
        wk = wk->next;
    }

    wk = new BrTableEntry(ad, p, (Scheduler::instance().clock()));
    wk->next = _table;
    _table = wk;
}

void BridgeTable::remove(int addr)
{
    if (!_table) {
        return;
    }

    BrTableEntry *wk = _table;
    if (_table->addr == addr) {
        _table = _table->next;
        delete wk;
        return;
    }

    BrTableEntry *par = _table;
    wk = _table->next;
    while (wk) {
        if (wk->addr == addr) {
            par->next = wk->next;
            delete wk;
            return;
        }
        par = wk;
        wk = wk->next;
    }
}

// purge any entry old than t
void BridgeTable::remove(double t)
{
    if (!_table) {
        return;
    }

    BrTableEntry *wk = _table;
    if (_table->ts <= t) {
        _table = _table->next;
        delete wk;
        return remove(t);       // not likely to happen since header is newer.
```

```
  }

  BrTableEntry *par = _table;
  wk = _table->next;
  while (wk) {
        if (wk->ts <= t) {
           par->next = wk->next;
           delete wk;
           wk = par->next;
        } else {
           par = wk;
           wk = wk->next;
        }
  }
}

//////////////////////////////////////////////////////
//                  BNEP                          //
//////////////////////////////////////////////////////
int BNEP::trace_all_bnep_ = 1;

BNEP::BNEP()
: _timer(this), sendTimer(this), inqCallback(this)
{
  bind("onDemand_", &onDemand_);

  rolemask_ = ROLEMASK;
  role_ = 0;
  numRole_ = 0;
  _chan = 0;

  nb_ = 0;
  nb_num = 0;
  waitForInq_ = 0;
  numConnReq_ = 0;
  schedsend = 0;

  onDemand_ = 0;
  _in_make_pico = 0;

  num_conn = 0;
  num_conn_max = 8;
  _conn = new Connection *[num_conn_max];
  int i;
  for (i = 0; i < num_conn_max; i++) {
        _conn[i] = 0;
```

```
    }

    _current = 0;
    trace_me_bnep_ = 0;
}

void BNEP::setup(bd_addr_t ad, LMP * l, L2CAP * l2, SDP * s, BTNode * node)
{
    bd_addr_ = ad;
    lmp_ = l;
    l2cap_ = l2;
    sdp_ = s;
    // l2cap_->bnep_ = this;
    node_ = node;
}

void BNEPSendTimer::handle(Event *)
{
    _bnep->handle_send();
}

void BNEPTimer::handle(Event *)
{
    _bnep->piconet_sched();
}

void BNEPInqCallback::handle(Event *)
{
    _bnep->inq_complete();
}

void BNEP::addSchedEntry(Piconet * pico, double len)
{
    if (!_current) {
        _current = new BNEPSchedEntry(pico, len);
        _numSchedEntry = 1;
        return;
    }

    BNEPSchedEntry *wk = _current;
    do {
        if (wk->pico == pico) {         // do an update for existing entry.
            wk->length = len;
            return;
        }
    } while ((wk = wk->next) != _current);
    wk = new BNEPSchedEntry(pico, len);

    // add to the end
    wk->next = _current;
    wk->prev = _current->prev;
    _current->prev->next = wk;
    _current->prev = wk;
    _numSchedEntry++;
}

void BNEP::removeSchedEntry(Piconet * pico)
{
    if (!_current) {
        return;
    }
    BNEPSchedEntry *wk = _current;
    if (_current->pico == pico) {
        if (_current == _current->next) {      // singleton
            _current = NULL;
        } else {
            wk->next->prev = wk->prev;
            wk->prev->next = wk->next;
            _current = _current->next;
        }
        delete wk;
        _numSchedEntry--;
        return;
    }
    do {
        if (wk->pico == pico) {
            wk->next->prev = wk->prev;
            wk->prev->next = wk->next;
            delete wk;
            _numSchedEntry--;
            return;
        }
    } while ((wk = wk->next) != _current);
}

void BNEP::disableScan()
{
    if (!_current) {
        return;
    }
    BNEPSchedEntry *wk = _current;
```

```
  do {
       if (!wk->pico) {
          if (wk->length > 0) {
               wk->length = -wk->length;
          }
          return;
       }
  } while ((wk = wk->next) != _current);
}

void BNEP::enableScan(double len)
{
  if (!_current) {
       addSchedEntry(NULL, len);
       return;
  }

  BNEPSchedEntry *wk = _current;
  do {
       if (!wk->pico) {
          wk->length = len;
          return;
       }
  } while ((wk = wk->next) != _current);

  addSchedEntry(NULL, len);
}

void BNEP::piconet_sched()
{
  int cntr = _numSchedEntry;
  Scheduler & s = Scheduler::instance();
  if (!_current) {
       return;
  }
  if (waitForInq_) {
       printf("waitForInq_\n");
       s.schedule(&_timer, &_ev, 30E-3);
       return;
  }
  _current = _current->next;

  printf("%d %f bnepSched p:%x %f\n", bd_addr_, s.clock(),
         (unsigned int) _current->pico, _current->length);

  lmp_->wakeup(_current->pico);
  while (_current->length <= 0 && cntr-- > 0) {     // mask out disabled
        _current = _current->next;
  }

  s.schedule(&_timer, &_ev, _current->length);
}

void BNEP::inq_complete()
{
  if (nb_) {
        lmp_->destroyNeighborList(nb_);
  }
  nb_ = lmp_->getNeighborList(&nb_num);
  waitForInq_ = 0;
  make_connections();
}

void BNEP::make_connections()
{
  if (nb_num < 1) {
        inq(1, 7);
        return;
  }

  int n = 1;
  if (canBeMaster()) {
        n = MIN(nb_num, 7);
  }

  Bd_info *wk = nb_;
  for (int i = 0; i < n; i++) {
        connect(wk->bd_addr_);
        wk = wk->next_;
  }
  numConnReq_ = n;
  schedsend = 0;
}

void BNEP::inq(int to, int num)
{
  lmp_->HCI_Inquiry(lmp_->giac_, to, num);
  lmp_->addInqCallback(&inqCallback);
  waitForInq_ = 1;
}
```

```
// Note:
// Well, BCAST packets probably is the first higher layer packet arrived, since
// Routing Agent and LL module will send them first before a data packet can
// be sent. (LL does not send ARP pkt any more.)
//
// Senarios for onDemand Scatternet formation:
// 1. no port/conn.
//    a. check L2CAP and LMP to see if any link exists. If so, add
//       BNEP conn/port quickly. send the packet.
//    b. no Links. retrieve neighbor list. Check capability. If canBeMaster,
//       if has neighbor, page them, otherwise, inquiry and paging.
//       If canBePANU, page one of neighbor and do a role switch upon
//       connection setup.

void BNEP::bcast(Packet * p)
{
   int i;

   // if num of bnep <= N do inqiry and paging

   if (onDemand_) {              // try to format the scatternet on demand
        if (num_conn < 1) {
           _q.enque(p);
           // _curPkt = p;
           make_piconet();
           return;
        } else if (canBeMaster() && num_conn < 2) {
           _q.enque(p);
           // _curPkt = p;
           make_piconet();
           return;
        }
   }

   printf("BNEP::bcast():num_conn:%d\n", num_conn);

   for (i = 0; i < num_conn_max; i++) {
        if (!_conn[i]) {
           continue;
        }
        _conn[i]->cid->enque(p->copy());
   }
   Packet::free(p);
}
```

```
#if 0
   // three cases: 1. bcast.

if (role_ == PANU) {
   if (_master_bd_addr) {
   } else if (_ondemand) {
        // inquiry and page
   } else {
        // drop the packet.
   }
} else if (role_ == NAP || role_ == GN) {
   // if no bridge, mac desn't match slaves. if _ondemand
   // inquiry and page
   //
   //
   if ((slot = findPort(mh->macDA())) >= 0) {
        _conn[slot]->cid->enque(p);
   } else {                        // MAC_BROADCAST
        bcast(p);
   }
} else {                           // BR
   if (_master) {
   }

}
#endif

void BNEP::make_piconet()
{
   if (_in_make_pico) {
        return;
   }

   _in_make_pico = 1;

   // Check if Links exists.  If so, make BNEP conn out of them.
   //ConnectionHandle *wk = l2cap_->_connhand;
   // while (wk) {
   // }

   // grab neighbor list from LMP
   if (nb_) {
        lmp_->destroyNeighborList(nb_);
   }
   nb_ = lmp_->getNeighborList(&nb_num);
```

```
  if (isMaster()) {
  } else if (isPANU()) {
       if (isBridge()) {
       }
  }

  if (canBeMaster()) {
       if (nb_num < 2) {
         if (lmp_->suspendCurPiconetReq()) {
              inq(1, 4);
         } else {
              inq(1, 4);
         }
       } else {
         make_connections();
       }
       return;
  } else if (canBePANU()) {
       if (canBeBridge()) {
       }
       if (nb_num < 1) {
         if (lmp_->suspendCurPiconetReq()) {
              inq(1, 3);
         } else {
              inq(1, 3);
         }
       } else {
         make_connections();
       }
       return;
  }
}

BNEP::Connection * BNEP::addConnection(L2CAPChannel * ch)
{
  Connection *c = new Connection(ch);
  if (num_conn == num_conn_max) {
       num_conn_max += num_conn_max;
       Connection **nc = new Connection *[num_conn_max];
       memset(nc, 0, sizeof(Connection *) * num_conn_max);
       memcpy(nc, _conn, sizeof(Connection *) * num_conn);
       delete[] _conn;
       _conn = nc;
  }
  num_conn++;
  for (int i = 0; i < num_conn_max; i++) {
        if (_conn[i] == 0) {
           _conn[i] = c;
           c->port = i;
           return c;
        }
   }
   return NULL;
}

void BNEP::removeConnection(L2CAPChannel * ch)
{
   Connection *c = lookupConnection(ch);
   if (c) {
        removeConnection(c);
   }
}

void BNEP::removeConnection(BNEP::Connection * c)
{
   _br_table.remove(c->daddr);

   num_conn--;
   _conn[c->port] = 0;
   delete c;
}

void BNEP::portLearning(int fromPort, Packet * p)
{
   hdr_ip *ip = HDR_IP(p);
   // hdr_cmn *ch = HDR_CMN(p);
   //hdr_mac *mh = HDR_MAC(p);

   //FIXME:put source ip addr as an alternative ??
   _br_table.add(ip->saddr(), fromPort);
   // basically, mac_addr == ip_addr in ns.
   // _br_table.add(mh->macSA(), fromPort);
}

int BNEP::findPortByIp(int ip)
{
   return _br_table.lookup(ip);
}

int BNEP::findPort(int macDA)
{
   // only if macDA is the other end of the link
```

```
  // otherwise use _br_table.lookup(macDA);
  Connection *conn = lookupConnection(macDA);
  if (conn) {
        return conn->port;
  } else {
        return -1;
  }
}

BNEP::Connection * BNEP::lookupConnection(bd_addr_t addr)
{
  for (int i = 0; i < num_conn_max; i++) {
        if (_conn[i] && _conn[i]->daddr == addr) {
           return _conn[i];
        }
  }
  return NULL;
}

BNEP::Connection * BNEP::lookupConnection(L2CAPChannel * ch)
{
  for (int i = 0; i < num_conn_max; i++) {
        if (_conn[i] && _conn[i]->cid == ch) {
           return _conn[i];
        }
  }
  return NULL;
}

L2CAPChannel *BNEP::lookupChannel(bd_addr_t addr)
{
  for (int i = 0; i < num_conn_max; i++) {
        if (_conn[i] && _conn[i]->cid->remote() == addr) {
           return _conn[i]->cid;
        }
  }
  return NULL;
}

void BNEP::handle_send()
{
  if (!_in_make_pico) {
        return;
  }
  _in_make_pico = 0;
  _send();
```

```
}

void BNEP::_send()
{
   Packet *p;
   while ((p = _q.deque())) {
         hdr_ip *ip = HDR_IP(p);

         // hdr_cmn *ch = HDR_CMN(p);
         hdr_mac *mh = HDR_MAC(p);
         int slot, i;

         int da = ip->daddr();

         if (mh->macDA() == (int) MAC_BROADCAST) {
            printf("BNEP::bcast():num_conn:%d\n", num_conn);

            for (i = 0; i < num_conn; i++) {
                  _conn[i]->cid->enque(p->copy());
            }
            Packet::free(p);
         } else if ((slot = findPort(da)) >= 0) {
            _conn[slot]->cid->enque(p);
         }
   }
}

void BNEP::schedule_send(int slots)
{
   Scheduler::instance().schedule(&sendTimer, &send_ev,
                                             BTSlotTime * slots);
}

// Master has a scheduler.  The slave is controlled by the master.
// t0: upper layer pkt arrives.
// t1: M: idle->Inq, Page,           S:idle->Scan
// t2: M: page_complete, upper layer   S: conn_ind, upper layer.
// t3: M: master piconet. decide the    S: Know when to return
//        intv, send to S:
// t4: M: send Data                    S: receive bcast DATA
//                                    put link on hold, Inq, page

void BNEP::channel_setup_complete(L2CAPChannel * ch)
{
   Connection *c = lookupConnection(ch);
   if (!c) {
        c = addConnection(ch);
    }
    _br_table.add(c->daddr, c->port);
    c->ready_ = 1;

    fprintf(stdout, "%d %s ", bd_addr_, __FUNCTION__);
    c->dump(stdout);
    fprintf(stdout, "\n");

    if (node_->scatFormator_) {
        node_->scatFormator_->connected(ch->remote());
        // return;
    }
    // Add arp stuff to LL arp table, if it exists.
    // TODO

    // Add routing table entry.
    // add SchedEntry.
    // TODO

    if (c->cid->connhand()->link->piconet->isMaster()) {
        becomeGN();
        _masterPort++;
        if (_masterPort == 1) {
          enableScan(30 * 1E-3);
          addSchedEntry(c->cid->connhand()->link->piconet, 1);
          if (_ev.uid_ <= 0) {
              // piconet_sched();
          }
        }

    } else {
        becomePANU();
#if 0
        enableScan(30 * 1E-3);
        addSchedEntry(c->cid->connhand()->link->piconet, 1);
        if (_ev.uid_ <= 0) {
          piconet_sched();
        }
#endif
    }

    if (!schedsend) {
        schedule_send(12);
        schedsend = 1;
    }
```

```
   if (c->_nscmd) {
        Tcl & tcl = Tcl::instance();
        tcl.eval(c->_nscmd);
        c->_nscmd = 0;
   }

   if (numConnReq_ > 1) {
        numConnReq_--;
   } else {
        _send();
   }
}

void BNEP::disconnect(bd_addr_t addr, uchar reason)
{
   Connection *c = lookupConnection(addr);
   // assert(c && c->cid);
   if (c && c->cid) {
        c->cid->disconnect(reason);
   } else {
        ConnectionHandle *connh = l2cap_->lookupConnectionHandle(addr);
        if (connh && connh->chan) {
           connh->chan->disconnect(reason);
        } else if (connh) {
           lmp_->HCI_Disconnect(connh, reason);
        }
   }

   if (c) {
        removeConnection(c);
   }
}

BNEP::Connection * BNEP::connect(bd_addr_t addr, hdr_bt::packet_type pt,
                                 hdr_bt::packet_type rpt, Queue * ifq)
{
   Connection *c;
   if ((c = lookupConnection(addr))) {
        return c;
   }
#if 0
   if (pt < hdr_bt::NotSpecified) {
        lmp_->defaultPktType_ = pt;
   }
   if (rpt < hdr_bt::NotSpecified) {
        lmp_->defaultRecvPktType_ = rpt;
```

```
  }
#endif

  // In reality, this call will block until the L2CAP Channel
  // is established. The simulator returns as long as Page request
  // is queued.  So, we have a flag in Connection to indicate if the
  // underlying L2CAP Channel setup is completed.
  L2CAPChannel *ch = l2cap_->L2CA_ConnectReq(addr, PSM_BNEP, ifq);

  if (pt < hdr_bt::NotSpecified) {     // try to change pktType
      ch->changePktType(pt);
  }
  if (rpt < hdr_bt::NotSpecified) {
      ch->changeRecvPktType(rpt);
  }

  c = addConnection(ch);
  if (ch->ready_) {
      c->ready_ = 1;
  }

  return c;
}

// In current implentment, bridges are routers. I.e., UCBT adopts a L3
// approach.  When packet is passed down to BNEP.  MacDA() should be the
// othe end of BNEP link, unless it is a broadcasting pkt.  Unlike
// specified in PAN profile, where an external interface may exist,
// a packet to an external interface should be directed to a different MAC
// by the routing agent (hier routing).
// 1. If the packet is bcast, send it to each port.
// 2. Lookup outgoing port for the pkt by its MacDA(), and send to that port,
// 3. otherwise, drop it if no port is found.
void BNEP::sendDown(Packet * p, Handler * h)
{
  int slot;
  hdr_ip *ip = HDR_IP(p);
  hdr_cmn *ch = HDR_CMN(p);
  hdr_bt *bh = HDR_BT(p);
  hdr_mac *mh = HDR_MAC(p);
  hdr_bnep *bneph = HDR_BNEP(p);
  // hdr_tcp *tcp = HDR_TCP(p);
  double now = Scheduler::instance().clock();

  bh->ts_ = now;               // record time stamp, used by BTFCFS
```

```
    node_->recordSend(ch->size(), ip->daddr(), ip->dport(), &bh->hops_,
                    &bh->flow_ts_, &bh->flow_seq_, &bh->flow_ts_lasthop_,
                    &bh->flow_seq_lasthop_, ip->saddr() == bd_addr_);

    bneph->u.ether.prot_type = mh->hdr_type();
    bneph->ext_bit = 0;

    if (trace_all_bnep_ || trace_me_bnep_) {
        fprintf(BtStat::log_, BNEPPREFIX0
                "%d %d:%d->%d:%d %f %d %d %d %d %d\n",
                bd_addr_, ip->saddr(), ip->sport(), ip->daddr(),
                ip->dport(), now,
                ch->next_hop(), bh->hops_, bh->flow_seq_, ch->size(),
                bh->flow_seq_lasthop_);
    }
#if 0
    // UCBT doesn't model ARP since bluetooth link are P to P and the
    // both ends of a link always know the MAC of each other.
    if (mh->hdr_type() == ETHERTYPE_ARP) {     // Arp packet, handle to proxy.
        handle_arp(p);
        return;
    }
#endif
    // int da = ip->daddr();
    // Add BNEP header.
    if (mh->macDA() == (int) MAC_BROADCAST) {
        bneph->type = BNEP_COMPRESSED_ETHERNET_DEST_ONLY;
        bneph->u.ether.daddr = MAC_BROADCAST;
        ch->size() += bneph->hdr_len();
        bcast(p);
        // } else if ((slot = findPort(mh->macDA())) >= 0) {
        // } else if ((slot = findPort(da)) >= 0) {
        // } else if ((slot = findPortByIp(ip->daddr())) >= 0) {
    } else if ((slot = findPortByIp(ch->next_hop())) >= 0) {
        bneph->type = BNEP_COMPRESSED_ETHERNET;
        ch->size() += bneph->hdr_len();
        _conn[slot]->cid->enque(p);
    } else {
        // A possible way to handle it is to bcast the pkt.  However,
        // choose to drop it at this moment.
        // bcast(p);
        if (node_->getRagent()) {
          node_->getRagent()->linkFailed(p);
        }
        // drop(p, "NoPort");
    }
```

```
}
// receive packet from L2CAP CID.
void BNEP::sendUp(Packet * p, Handler * h)
{
  hdr_ip *ip = HDR_IP(p);
  hdr_cmn *ch = HDR_CMN(p);
  hdr_bt *bh = HDR_BT(p);
  hdr_bnep *bneph = HDR_BNEP(p);
  // hdr_tcp *tcp = HDR_TCP(p);

  if (bneph->u.ether.prot_type == ETHER_PROT_SCAT_FORM) {
      node_->scatFormator_->recv(p, HDR_BT(p)->sender);
      return;
  }

  double now = Scheduler::instance().clock();
  node_->recordRecv(ch->size(), ip->daddr(), ip->dport(),
                   bh->hops_, bh->flow_ts_, bh->flow_seq_,
                   bh->flow_ts_lasthop_, bh->flow_seq_lasthop_);
  ch->size() -= bneph->hdr_len();

#if 0
  // set mac frame paramter, we don't need to do so in the simulator.
  mh->hdr_type() = bneph->u.ether.prot_type;
  if (bneph->type == BNEP_COMPRESSED_ETHERNET) {
  } else if (bneph->type == BNEP_COMPRESSED_ETHERNET_DEST_ONLY) {
  } else if (bneph->type == BNEP_COMPRESSED_ETHERNET_SOURCE_ONLY) {
  } else if (bneph->type == BNEP_GENERAL_ETHERNET) {
  }
#endif

  if (trace_all_bnep_ || trace_me_bnep_) {
      fprintf(BtStat::log_, BNEPPREFIX1
              "%d %d:%d->%d:%d %f %f %d %d %d %f %d\n",
              bd_addr_, ip->saddr(),
              ip->sport(), ip->daddr(), ip->dport(),
              now, (now - bh->flow_ts_), bh->hops_, bh->flow_seq_,
              ch->size(), (now - bh->flow_ts_lasthop_),
              bh->flow_seq_lasthop_);

  int fromPort = findPort(HDR_BT(p)->sender);
  portLearning(fromPort, p);
  uptarget_->recv(p);
  return;
}
}
```

Publication

Information Technology Journal, 2007
ISSN 1812-5638
© 2007 Asian Network for Scientific Information

IPSec Based Bluetooth Security Architecture

Al-Muthanna Al-Hawamdeh, Adil Khan and M. Sikander Hayat Khiyal
Department of Computer Science, International Islamic University, Pakistan

Abstract: Bluetooth has recently obtained an unprecedented success in gaining a wide industry support. The main aim of Bluetooth is to eliminate interconnection cables and to connect one device to another via a universal radio link. First generation Bluetooth chipsets are limited in range to approximately 10 m under typical line-of-sight conditions. Future implementations will provide roughly ten times that range. In the present study, a performance assessment, processing delay and throughput of IPSec over Bluetooth based on the Bluetooth Network Encapsulation Protocol (BNEP) operation scenario, is presented. In particular, the performance of the point-to- point link and the effectiveness of IPSec (authentication and encryption) algorithms in Bluetooth. Moreover, some networking issues that may limit Bluetooth applicability in some environments will be pointed out.

Key words: Bluetooth, IPSec, security, BNEP protocol, MD-5, SHA-1, DES, 3DES

INTRODUCTION

Bluetooth was initially designed as an efficient cable replacement technology primarily for handheld devices. Indeed, all the devices belonging to one person can form a PAN (Personal Area Network) using Bluetooth. Bluetooth protocol stack is divided into five layers (Bluetooth, 2001; Saarinen, 2000):

- Bluetooth Core Protocols: including Baseband, LMP, L2CAP and SDP, comprise exclusively Bluetooth-specific protocols developed by the Bluetooth SIG that are required by most of the Bluetooth devices.
- Cable Replacement Protocol: i.e., RFCOMM protocol is based on the ETSI TS 07.10 that emulates serial line control and data signals over Bluetooth Baseband to provide transport capabilities for upper level services.
- Telephony Control Protocols: including TCS Binary and AT-commands are used to define the call control signaling, mobility management procedures and multiple usage models for the Bluetooth devices to establish the speech and data calls and provide FAX and modem services.
- Adopted Protocols: including PPP, UDP/TCP/IP, WAP, WAE, etc. Due to the open nature of the Bluetooth specification, additional protocols (e.g., HTTP, FTP, etc.) can be accommodated in an interoperable fashion.
- Host Controller Interface (HCI): i.e., the boundary between hardware and software provides a uniform command interface to access capabilities of hardware, e.g., Baseband controller, link manager, control and event registers.

The layers of Cable Replacement, Telephony Control and Adopted Protocols form the application-oriented protocols that enable applications to run over the Bluetooth core protocols.

IPSec

The IPSec protocol suite is used to provide privacy and authentication services at the IP layer. It provides a set of security algorithms plus a general framework that allows a pair of communicating entities to use whichever algorithms provide security appropriate for the communication.

The elements describing the set of IPSec protocols are divided into six groups:

- There is the main Architecture, which broadly contain the general concepts, security requirements, definitions and mechanisms defining IPSec technology.
- There is the ESP Protocol and an AH Protocol.
- The Encryption Algorithm, describes how various encryption algorithms are used for ESP.
- The Authentication Algorithm describes how various authentication algorithms are used for both ESP and AH.

Corresponding Author: Al-Muthanna Al-Hawamdeh, Department of Computer Science, International Islamic University, Pakistan
Tel: (0092) 03335417855

- The Key Management.
- The DOI contains values needed for the other elements to relate to each other. This includes for example encryption algorithms, authentication algorithms and operational parameters such as key lifetimes.

IP AUTHENTICATION HEADER (AH)

The IP Authentication Header (AH) is used to provide connectionless integrity and data origin authentication for IP datagram's and to provide protection against replays.

AH may be employed in two ways: transport mode or tunnel mode. The former mode is applicable only to host implementations and provides protection for upper layer protocols. Tunnel mode may be employed in either hosts or security gateways.

The first step of integrity protection is to create a hash by using a keyed hash algorithm, also known as a Message Authentication Code (MAC) algorithm. A standard hash algorithm generates a hash based on a message, while a keyed hash algorithm creates a hash based on both a message and a secret key shared by the two endpoints. The hash is added to the packet and the packet is sent to the recipient. The recipient can then regenerate the hash using the shared key and confirm that the two hashes match, which provides integrity protection for the packet. IPSec uses Hash Message Authentication Code (HMAC) algorithms, which perform two keyed hashes. Examples of keyed hash algorithms are HMAC-MD5 and HMAC-SHA-1. Another common MAC algorithm is AES Cipher Block Chaining MAC-AES-XCBC-MAC-96- (Kent and Atkinson, 2004).

ENCAPSULATING SECURITY PAYLOAD(ESP)

The Encapsulating Security Payload (ESP) header is designed to provide a mix of security services in IPv4 and IPv6. ESP may be applied alone, in combination with the IP Authentication Header (AH), or in a nested fashion (Kent and Atkinson, 2004b).

ESP is used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service and limited traffic flow confidentiality. The set of services provided depends on options selected at the time of Security Association establishment and on the placement of the implementation.

ESP uses symmetric cryptography to provide encryption for IPSec packets. Accordingly, both endpoints of an IPSec connection protected by ESP encryption must use the same key to encrypt and decrypt the packets. When an endpoint encrypts data, it divides the data into small blocks and then performs multiple sets of cryptographic operations using the data blocks and key. Encryption algorithms that work in this way are known as block cipher algorithms. When the other endpoint receives the encrypted data, it performs decryption using the same key and a similar process, but with the steps reversed and the cryptographic operations altered. Examples of encryption algorithms used by ESP are AES-CBChaining, AES Counter Mode, DES and 3DES (Kent and Atkinson, 2004b).

IPSec OVER BLUETOOTH

The proposed idea is that authentication and encryption in Bluetooth to be provided on IP or application level by using IPSec according to RFC 2401 (1998) at the IP level. A protocol like IPSec is most suitable to secure end-to-end IP services like Virtual Private Network (VPN) services. IPSec can be used for any IP connection independent of the particular access method. Here only LAN access using the Bluetooth wireless technology is considered. It is important to notice that the use of link level security and VPN solutions does not exclude each other but rather complement each other.

IPSec, however, can protect any protocol running above IP and any medium which IP runs over. More to the point, it can protect a mixture of application protocols running over a complex combination of media. This is the normal situation for Internet communication; IPSec is the only general solution.

The problems raises is that Bluetooth enabled devices will have the ability to form networks and exchange

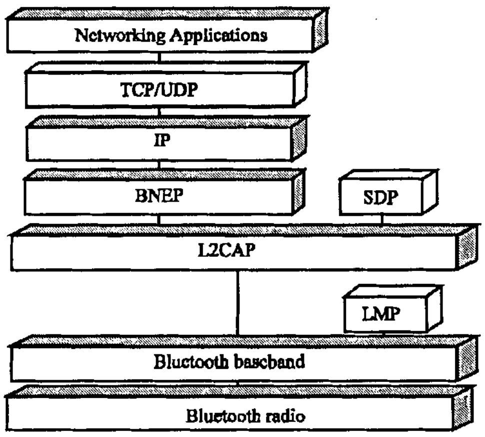


Fig. 1: BNEP Stack

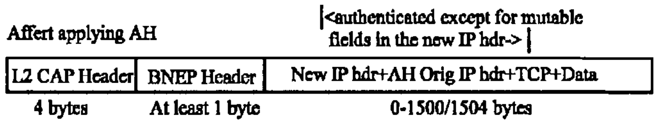

Affert applying AH
|<authenticated except for mutable fields in the new IP hdr-> |
L2 CAP Header
BNEP Header
New IP hdr+AH Orig IP hdr+TCP+Data
4 bytes
At least 1 byte
0-1500/1504 bytes

Fig. 2: BNEP after applying AH

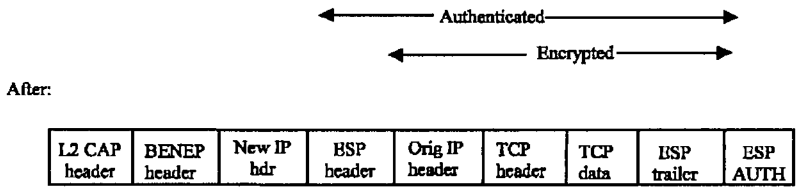


Fig. 3: BNEP after applying ESP

information. For these devices to interoperate and exchange information, a common packet format needs to be defined to encapsulate layer 3 network protocols.

Due to that, a specific packet format used to transport common networking protocols over the Bluetooth media (RFC 894,1996) (RFC 2225,1998) (RFC2734, 1999). The packet format is based on Ethernet/DIX Framing as defined by IEEE 802.3 according to Ether Typ (2006) (Anonymous, 1980) (Inlernat Enigeering Task Force, 1996; 1998; 1999).

The functional requirement for Bluetooth networking encapsulation protocol includes the following according to BNEP specification (2002):

- Support for common networking protocols such as IPv4, IPv6, IPX and other existing or emerging networking protocols.
- Low Overhead -- The encapsulation format SHALL be bandwidth efficient.

The following points illustrate the BNEP header format.

Bnep type: Seven bit Bluetooth Network Encapsulation Protocol.

Type value identifies the type of BNEP header contained in this packet (BNEP specification, 2002).

Extension flag (E): One bit extension flag that indicates if one or more extension headers follow the BNEP Header before the data payload if the data payload exists. If the extension flag is equal to 0×1 then one or more extension headers follows the BNEP header. If the extension flag is equal to 0x0 then the BNEP payload follows the BNEP header (BNEP specification, 2002).

BNEP packet: Based on the BNEP Type (BNEP specification, 2002).

Bluetooth Network Encapsulating Protocol (BNEP) accommodates IP communication by transporting IP packets between two Ethernet-based link layer end-points on an IP segment. It encapsulates the IP packets in BNEP headers, letting the source and destination addresses reflect the Bluetooth end-points and setting the 6-bit Networking Protocol Type field to code for an IP packet in the payload. BNEP finally encapsulates the BNEP packet in an L2CAP header and sends it over the L2CAP connection (Fig. 1).

Figure 2 and 3 show BNEP with an IPv4 packet payload sent using L2CAP after positioning AH header and ESP header for tunnel modes.

CRYPTANALYSIS

In Bluetooth Encryption,several attacks and attempts at cryptanalysis of E0 (Lu and Vaudenay, 2004) and the Bluetooth protocol have been made and a number of vulnerabilities have been found. In 1999, Miia Hermelin and Kaisa Nyberg showed that E0 could be broken in 2^{64} operations (instead of 2^{128}), if a 2^{64} bits output is known. This type of attack was subsequently improved by Kishan Chand Gupta and Palash Sarkar. scott fluhrer, found a theoretical attack with a 2^{80} operations precalculation and a key search complexity of about 2^{65} operations. He deduced that the maximal security of E0 is equivalent to that provided by 65-bit keys and that longer keys do not improve security. Fluhrer's attack is an improvement upon earlier work by golic, bagini and morgani, who devised a 2^{70} operations attack on E0.

Lu and Vaudenay (2005). Published a cryptanalysis of E0 based on a conditional correlation attack. Their best result required the first 24 bits of $2^{23.8}$ frames and 2^{38}

computations to recover the key. The authors assert that "this is clearly the fastest and only practical known-plaintext attack on Bluetooth encrytion compare with all existing attacks (Lu and Vaudenay, 2004). In the authentication scheme of Bluetooth there seems to be some weaknesses.

SAFER+ according to NIST (1998) was submitted as a candidate for the Advanced Encryption Standard and has a block size of 128 bits. The cipher was not selected as a finalist. SAFER+ was included in the Bluetooth standard as an algorithm for authentication and key generation.

Its found that in SAFER+/192 and SAFER+/256, the key schedules do a poor job of getting the whole key involved quickly in the encryption process. SAFER+/192 takes five (of twelve) rounds to get the whole key involved in the encryption process; SAFER+/256 takes nine (of sixteen) rounds to do so. This contrasts with SAFER+/128, where every round is affected by every bit of key.

Due to this slow key diffusion, a meet-in-the-middle attack was found on SAFER+/256. This attack requires work equivalent to about 2^{240} SAFER+/256 encryptions and about $12*2^{24}$ bytes of memory. Also due to this slow key diffusion, a related-key attack was found on SAFER+/256. This attack requires very little memory, $3 * 2^{32}$ chosen plaintexts encrypted under two different keys with a chosen XOR relationship and work approximately equivalent to 2^{200} SAFER+/256 encryptions (Kelsey *et al.*, 1999).

SYSTEM DESIGN

The simulation of the IPSec protocols in NS2 was based on the existing implementation of wireless network NS-2 (The Network Simulator) version 2 and UCBT (Bluetooth extension for NS2) (Fig. 4). UCBT implements a full Bluetooth stack, including Baseband, LMP, L2CAP, BNEP layers (UCBT, 2004).

Among the most important design principles for BT networking are:

- Providing for flexibility in usage as a universal short-range low-cost low-power technology in a variety of different scenarios.
- Ensuring that BT devices made by different manufacturers can inter-operate.

Following the first principle, BT networking topology is built upon the flexible concepts of the scatternet and the piconet. A piconet is an ad hoc collection of BT devices, where one of the devices takes the role of the master of the piconet and the other devices take the role of the slaves. Since each node could be a slave on multiple piconets, a larger network structure may be formed out of multiple piconets. This larger structure is the scatternet.

Following the second design principle, a BT protocol stack has been defined (Fig. 5). Usage profiles have also been defined for different usage scenarios, such as LAN access, to allow devices from different manufacturers to inter-operate. The profiles are collections of messages, procedures, features and parameter settings that must be used in order to provide specific services or usage scenarios for BT (Bluetooth, 2001). The use of profiles somewhat limits the flexibility in terms of usage of higher-layer protocols, network topologies and usage scenarios to what is contained in the profiles.

This is a tradeoff between the two principles discussed. In an effort to make use of existing protocols,

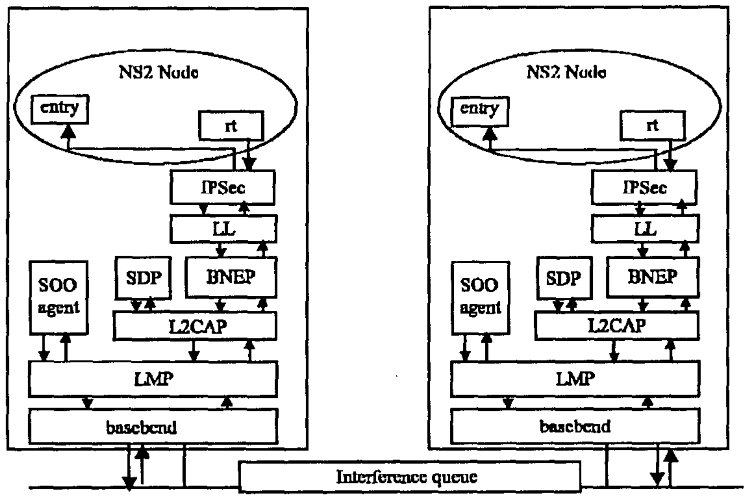


Fig. 4: IPSec architecture in Bluetooth

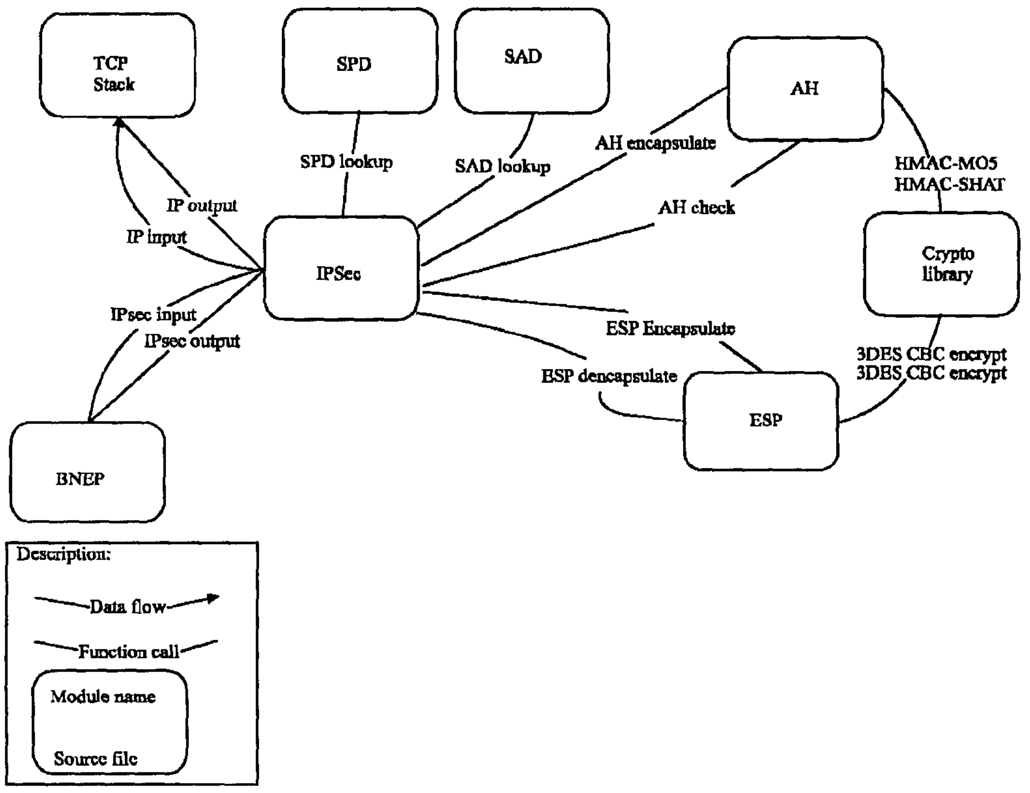


Fig. 5: IPSec system with dependencies

especially those with large installed bases like PPP, BT gives up some efficiency and flexibility that it could have had if more BT specific protocols had been defined. This is a second tradeoff in BT networking design. With future definitions of additional profiles (and maybe additional protocols), such constraints on BT usage might be relaxed.

QoS support: BT provides some support for bandwidth allocation and latency control. FEC (Forward Error Correction) and retransmission mechanisms ensure low error rates at the expense of more overhead and the protocols ensure in-sequence delivery of packets so that reliable, orderly delivery of packets is expected. Link delay and delay jitter are also not expected to be large, once a link has been established.

Mobility support: Partial support for handoff is provided by BT. the fact that a device can be a member of multiple piconets at the same time means that it could in theory perform soft handoffs. However, it is up to the applications to include other necessary features, e.g., the bridging, switching/routing and buffering mechanisms in the backbone.

The IPSec Module is the central part (Fig. 5), which does the whole standard conform processing of the incoming and outgoing IP traffic. It uses a set of data bases (SPD and SAD) to determine the flow of the IP packets. The main processing is then done in the AH and ESP module. A small cryptographic library contains all the functionality used to encrypt, decrypt or to authenticate the packets.

Since there are many security protocols in terms of algorithms in IPSec, we decided on:

- HMAC-MD5 and HMAC-SHA1 to provide origin authentication and integrity for IP packets. MD5 should be preferred because its performance is much better than that of SHA1 (Chaudhry *et al.*, 2002).
- In ESP implementation we support both encryption and authentication. Encryption is done by the widely used 3DES algorithm, which is applied in CBC mode. Pure DES is also implemented. For authentication we use HASH-MAC MD5.

We needed to identify the different modules out of the IPSec architecture so that we were able to characterize the following attributes of the modules:

- Priority
- Dependencies
- Performance sensibility

An important part of our work was to find a suitable IP-stack and Bluetooth stack that is able to carry our IPSec implementation. The Network Simulator NS-2 TCP/IP Stack has all the desired features: modular design, active community and free BSD-style license. As well as UCBT which has all the desired features needed for Bluetooth stack.

From Fig. 5, any inbound data is forwarded to IPSec input function. Depending on the protocol field in the packet header, the entire packet is forwarded to the IP protocol stack. If the packet could be identified as belonging to the suit of IPSec protocols, it is transferred to the IPSec library. Pure IPSec specific processing, such as applying ESP de-/encapsulation or AH de-/encapsulation is done within the IPSec library.

After these steps, the original IP packet is rebuilt by applying new offsets and packet length to the pbuf structure. Then the clear-text packet is passed up to the TCP/IP stack in NS-2.

For outbound packets, all IP based protocols forward their data to IPSec output function. Here the decision is made whether the packet needs IPSec processing or not. Depending on the appropriate Security Association, AH or ESP functionality will encapsulate the packet. After these steps, the packet is forwarded to the BNEP Class of Bluetooth Stack and sent over to the receiver.

The Security Policy Database (SPD) can be accessed from the IPSec module as shown in Fig. 5. This database contains all rules required to decide how to handle packets, which have security associations but also how to handle non-IP traffic. There are several possibilities: any non-IPSec packet can be forwarded to the default protocol handler (in order for connections from non-IPSec nodes are accepted) or any non-IPSec packet can be dropped immediately without wasting CPU time on further analysis.

BASIC CONCEPT OF SECURITY ASSOCIATION

IPSec needs the Security Policy Database and the Security Association Database to process packets correctly.

The SPD defines the packets, to which IPSec needs to be applied. To guarantee that each packet is processed the right way, each IP packet leaving or entering the system must be checked against the SPD. We call this action the SPD lookup. This lookup does nothing except compare the selectors from the database with the ones from the packet. The SPD lookup delivers back the following results:

- BYPASS: This packet is forwarded directly to Bluetooth layers without applying IPSec.
- DISCARD: This packet is discarded, it will be dropped.
- APPLY: This packet requires IPSec processing

If the result of a SPD lookup is BYPASS, the unmodified packet is forwarded to the Bluetooth layers. This is particularly useful if certain protocols such as ICMP should not be protected by IPSec or communication with non-IPSec hosts must be concurrently possible.

The DISCARD rule is returned when the intention is not to process this packet. If this is the case, the packet will be dropped. This means that we simply delete the packet instead of passing it to Bluetooth layers. It is possible to use this feature to build a primitive firewall.

IPSec processing is only needed if the result of the SPD lookup is APPLY. Whenever a packet matches an SPD entry whose policy says APPLY, then there must also be an SA that describes exactly how the packet has to be processed.

A successful SPD lookup provides us with a pointer to the SP over which we can access the SA using a pointer stored in the SP structure. The packet can be processed only after Security Association parameters are successfully negotiated.

When a packet leaves TCP/IP stack, the very first step is an SPD lookup, a determination of how the packet must be processed. When the policy says APPLY, the IPSec process continues. Otherwise the function passes the packet to the Bluetooth stack or returns to the TCP/IP stack without doing anything.After the new IPSec packet has been built, it must be sent out on the Bluetooth stack as shown in Fig. 6.

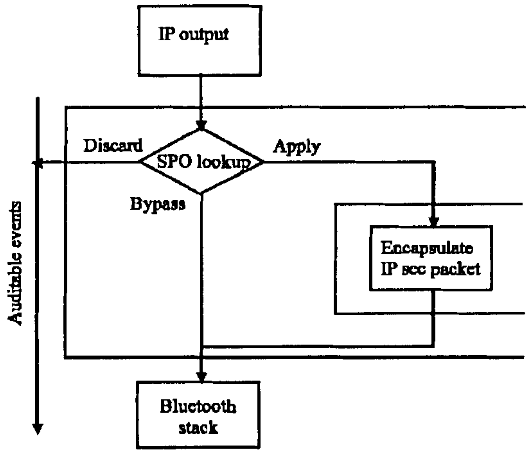


Fig. 6: Outbound processing

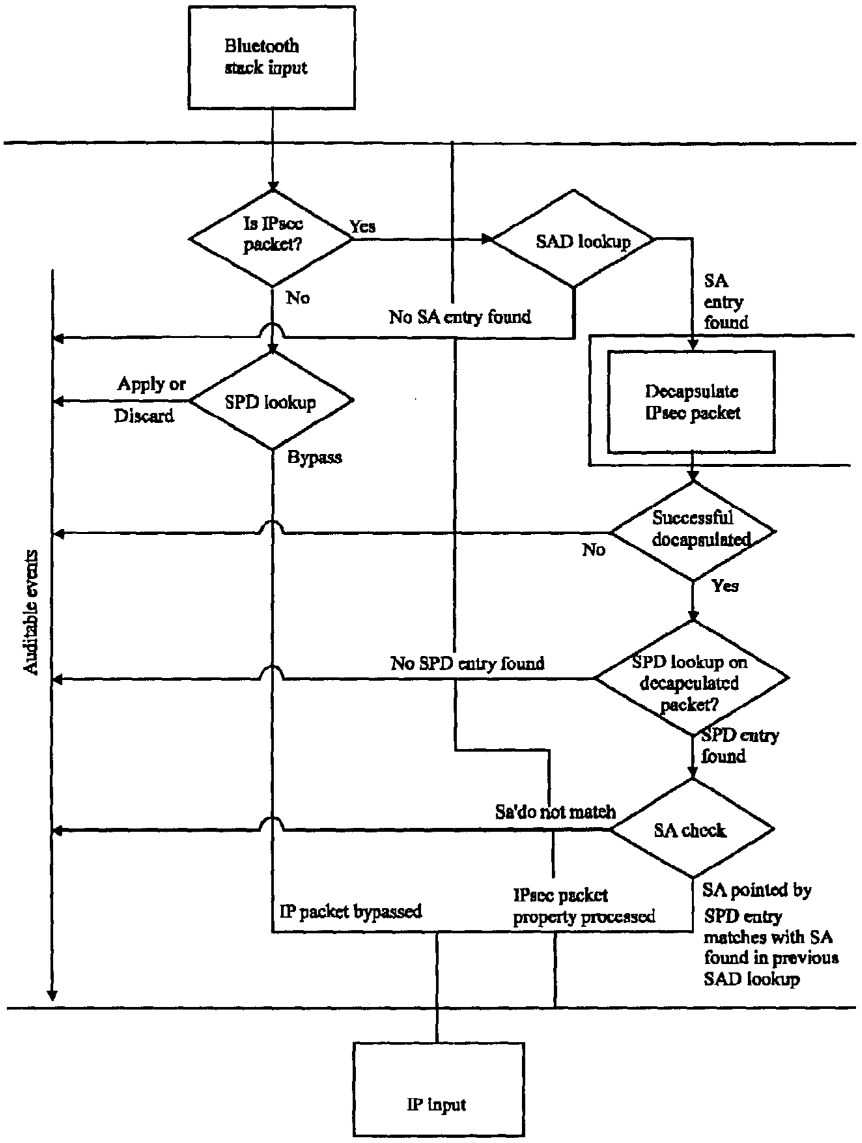


Fig. 7: Inbound processing

From Fig. 7, inbound processing is somewhat different because an incoming IPSec packet already has an SPI, which allows a direct lookup in the SAD table. The reason for using the SPI is straightforward. The incoming IPSec packet may be encrypted and so the SPD lookup, which must be performed on the inner packet data, cannot be performed. The SAD lookup would directly give back an SA if one was found. If no SA is found, then the packet must be discarded.

With the valid SA we are now able to process the packet properly. In inbound processing this corresponds to decapsulation in ESP or integrity checking in AH. After the IPSec packet has been decapsulated, it can be passed on to the TCP/IP.

AH PROCESSING

Authentication is done by the well-known HASH-MAC4 MD5 and HASH-MAC SHA1 algorithms. These are the algorithms requested by the standard. MD5 should be preferred because its performance is much better than that of SHA1.

If one considers that ESP also supports integrity and authentication, one may think that there is no need for AH. This is not true because the authentication and integrity check of AH is a bit more sophisticated. Authentication in AH covers more fields of the packet than ESP does.

AH processing can be split up into inbound and outbound processing.

AH inbound processing itself described step-by-step

- In order to check the integrity and the authentication of the packet, the ICV must be calculated. The ICV calculation in AH also covers the outer IP header. In this header there are so-called mutable fields, which change their value while they are sent across the network. Those fields (Type of Service, Offset, TTL and checksum) must first be set to zero. The ICV fields in the AH header must be backed up and zeroed, so that later comparing remains possible. It becomes clear that AH authentication also covers the source and destination address of the outer IP packet.
- The packet is now ready to be verified and the integrity check value can be calculated over the whole packet. The SA determines the appropriate algorithm and key.
- The calculated ICV can be compared with the one saved in the first step. Processing continues only if the calculated ICV matches the original one.
- The authentication of the packet is now verified and the anti-replay check can be performed. If it is successful, the sequence number (stored in the SA) is incremented. Finally, the offset and packet length are passed back.

AH outbound processing described step-by-step:

- First of all a new AH header is placed in front of the IP packet, leaving a gap between the inner IP header and the AH header. This gap is later used to place the ICV. The AH header fields: next header, length, SPI and sequence number are added.
- After the outer IP header has been constructed, only the source and destination address, version, header length and total length are set. The other fields are set to zero as a preparation for the ICV calculation. Padding is not required because the packet is already aligned.
- The integrity check value can now be calculated and placed into the gab between AH header and inner IP header.
- After the ICV has been calculated, the zeroed fields are now filled with the appropriate values.
- Finally, the offset and the packet length are passed back.

ESP PROCESSING

In our ESP implementation we support both encryption and authentication. Encryption is done by the widely used 3DES algorithm, which is applied in CBC mode. Pure DES is also implemented. For authentication we use HASH-MAC MD5 and HASH-MAC SHA1.

ESP processing can be split up into inbound and outbound processing.

ESP inbound it self described step-by-step:

- A check in the SA structure indicates whether authentication needs to be checked or not. If an authentication algorithm is specified within the SA, the ICV must be calculated and compared with the one stored at the end of the ESP packet. The ICV is calculated over the whole ESP header, IV and encrypted payload. Processing continues only when the packets ICV matches our recalculated one.
- In the next step we have to decrypt the packet. The decryption algorithm and the secret key can be accessed over the SA. Because the packet was encrypted in CBC-mode, the IV must be copied out of the ESP packet. The IV is stored between ESP header and encrypted payload. The decryption happens in-place, so no copying must be done.
- Before everything is done the sequence number counter in the SA is incremented and optionally the same is done with the anti-replay window. To let the caller of the ESP function know about the location and the size of the extracted IP packet, the offset and the packet length are giving back.

ESP outbound processing itself described step-by-step:

- The first step of encapsulation is to test whether the decremented TTL field of the IP header reaches zero. If this is the case, the packet must be discarded in order to prevent endless straying of packets.
- Then we have to calculate how much padding must be added to fulfill the requirements of the encryption algorithm. The right amount of padding bytes is added at the end of the payload. The fields: padding length and next header are appended right after the padding.
- Encryption is performed according to the settings in the SA. After encryption, the used IV is copied in front of the encrypted payload.
- ESP header is added in front of the IV. Inserted are a incremented sequence number and the SPI taken out of the SA.
- The SA must be checked to see if authentication is enabled. If this is the case, then the ICV must be calculated according the SA's settings. The ICV, which is calculated the ESP header, the IV and the encrypted payload, is copied at the end of the payload.

- Outer IP header can be constructed using the tunnels source and destination address given as input arguments to the function. The TOS field is copied from the inner IP header. Finally, the offset and the length are passed back, so that the caller can update its data structure, where the packet is stored.

RESULTS AND PERFORMANCE ANALYSIS

IPSec over Bluetooth scenario was simulated with a constant FTP source on top of TCP with a packet size of 2000 bytes between two nodes (node 0 and node 1).

Table 1 show the Simulations of the system design described in section 5 using different algorithms using the hand-off rate of 60 seconds. HMAC- MD5 has shown to have the highest No of Bytes sent where 3DES HMAC-MD5 has lowest.

The difference in simulations' time varies from one algorithm to another due to the delay required for the authentication or encryption procedure. As delay consumes more time, it affects the No. of packets generated. As shown in Table 1, HMAC-MD5 algorithm has the lowest delay but the highest No. of packets generated, where 3DES HMAC-MD5 algorithm has the highest delay but the lowest No. of packets generated.

It's also noticed that the average packet size is not fixed due to different types of packets (TCP, Authenticated packets, Encrypted Packets, ACK, etc...,) sent between nodes where each has a different size.

Authenticated packets (HMAC-MD5 and HMAC SHA1) almost have the same average packet size while 3DES HMAC-MD5 has higher average packet size in encryption (ESP).

In order to study the Performance of IPSec over Bluetooth, we measured the cumulative sum of packets in each of the cases, as well as the throughput and end 2 end delay imposed by the security protocols IPSec.

Figure 8, depicts the decrement of the TCP packets sequence in different scenarios (No IPSec, MD5, SHA1, DES-MD5 and 3DES-MD5). The reason behind this decrease in No. of packets is the Authentication and Encryption Procedure which includes SPD lookup when the result is DISCARD. This means that we simply delete the packet instead of passing it to Bluetooth layers, so the packet is lost thus more delay is consumed and wasted The Fig. 8a and b scenarios showed that number of packets is decreasing in authentication and encryption compared to Bluetooth with no IPSec. As it is seen HMAC-MD5 performs better than HMAC-SHA1 while sending and receiving packets. As well as DES-MD5 performs better than 3DES-MD5.

Plain Bluetooth packets and HMAC-MD5 share almost the same performance at trade-off rate 7, but later on HMAC-MD5 start losing packets and decrements more.

On the other side, SHA1, DES-MD5 and 3DES-MD5 share almost the same performance at trade-off rate 12 where SHA1 starts catching up with MD5, while DES-MD5 and 3DES-MD5 remain sharing it till trade-off rate 18.

Throughput is defined as the percentage of packets that experience a Bit Error Probability (BEP) that is less than a maximum allowable BEP, BEPth. The BEPth is set to a value of 0.1% (or 10^{-3}), i.e., a packet is considered reliable as long as its BEP is not greater than BEPth. The quality of the link varies from one BT unit to the next within the scatternet. It is highly dependent on the relative spacing of the BT devices within the scatternet. Hence, the percentage of reliable packets has to be determined by collecting the statistic over many realizations of the scatternet. The conservative measure of 10th percentile is used to represent the throughput performance of the BT network.

The throughput results are shown in Fig. 9. There is a significant difference between the simulated scenarios. As we can see, the throughput in HMAC-MD5 and HMAC-SHA1 is not the same due to a better performance from HMAC-MD5. In respect to the encryption, DES-MD5 has a better throughput than 3DES-MD5.

In sending packets, plain Bluetooth packets throughput remain at a rate of 78, HMAC-MD5 throughput is at around 50 while HMAC-SHA1 at a rate of 40. For encryption algorithms, throughput rate is 15 for DES-MD5 and 8 for 3DES-MD5.

These rates decrease while receiving packets; they become around 37 for plain Bluetooth packets, 28 for HMAC-MD5, 20 for HMAC-SHA1, 6 for DES-MD5 and 2 for 3DES-MD5.

Table 1: Retreived initial results

Parameters	No. IPSec	HMAC MD5	HMAC SHA1	DES HMAC- MD5	3DES HMAC- MD5
Simulation start time	1.30513601	1.4471	1.46	1.5374	1.6665
Simulation end time	60.08551301	60.073	59.6643	59.9108	60.0448
Simulation length (Sec)	58.7880377	58.62592863	58.20427423	58.37336781	58.37830156
No of generated packets	4500	3501	24439	658	237
No of lost packets	2260	1753	1222	330	119
No of received packets	2240	1648	1217	328	118
Avg. packet size	1051.7595	1066.1472	1066.2114	1074.5023	1069.6913
No of sent bytes	4698000	3700136	2578664	701480	251668

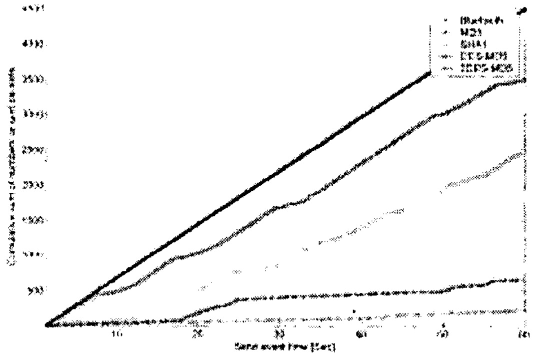


Fig. 8a: Performance of Cumulative sum of numbers of sent packets

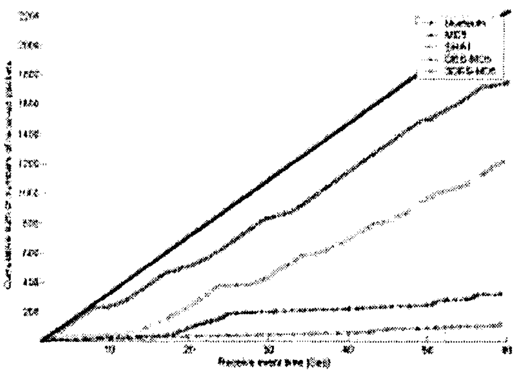


Fig. 8b: Performance of Cumulative sum of numbers of received packets

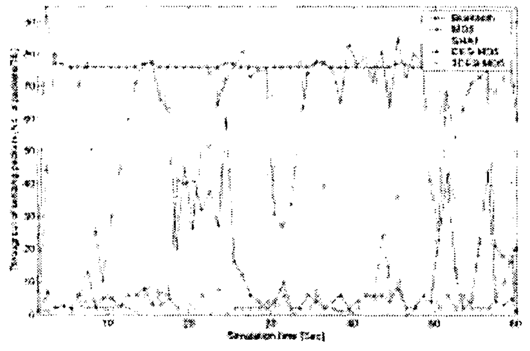


Fig. 9 (a): Throughput of sending packets

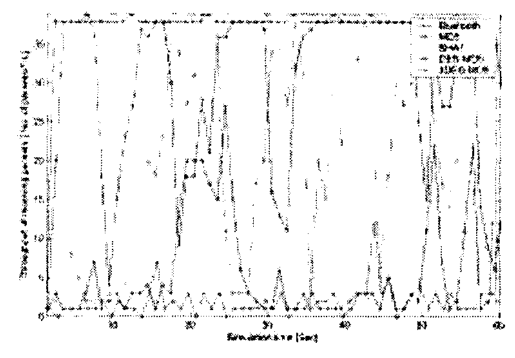


Fig. 9b: Throughput of receiving packets

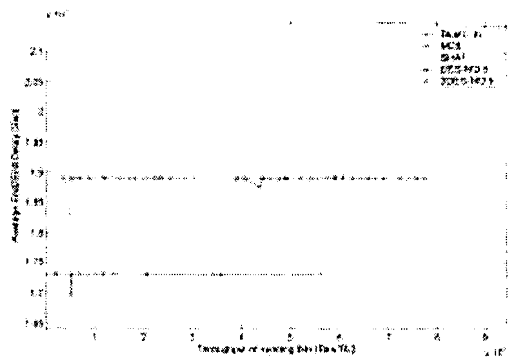


Fig. 10a: Throughputs of senidng bits vs. average simulation end 2 end delay

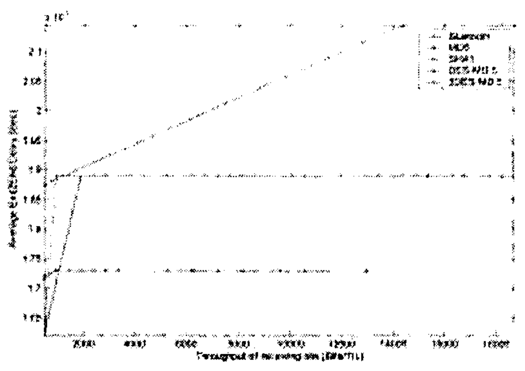


Fig. 10b: Throughputs of receiving bits vs. average simulation end 2 end delay

Contrary to our expectations, the throughput in Bluetooth environment is driven by the effect of:

- The successive sending of constant packets, whether its ACK packet where throughputs increases or encrypted and authenticated packets in which throughput decreases.
- The erratic behavior of delay imposed by the encryption and decryption of the data.
- The erratic behavior of the Bluetooth wireless link, Wireless links are characterized by higher bit error rates and this causes inefficiencies in the operation of TCP. Essentially, any perceived packet loss (occurring because of error or buffer overflow) is construed by a TCP sender as occurring due to buffer overflow., The response of TCP to all such events is to invoke its congestion control procedures, resulting in unnecessary window reduction, which causes a drop in the TCP throughput. Note, though, that some of the packet losses occur due to corrupted packets being dropped by the link layer and invoking the congestion avoidance procedures when these events occur is not desirable.

Figure 10 show a comparison of Throughputs vs. average simulation end 2 end delay. There is a big and noticed difference of end 2 end delays while sending the bits and rise in the throughput. While in receiving bits, both factors also affect the bits transmission.

The main difference noticed raises at throughput of sending and receiving packets, while end 2 end delay almost equal in case of Authentication algorithms, it's also the same as for Encryption algorithms.

CONCLUSION

This study proposed a new Bluetooth security scheme, which allows ad-hoc (PAN) based on Bluetooth technology to communicate with other devices in full secure channel includes authentication and encryption, unlike for the present schemes with weak security (E0 and E1).

In addition, as shown in Fig. 11, the throughput in sending packets is reduced by almost 35-50% for authentication and 80-90% for encryption compared with the cases where IPSec was not used. While in receiving packets, throughput is reduced by almost 30-45% for authentication and 80-95% for Encryption.

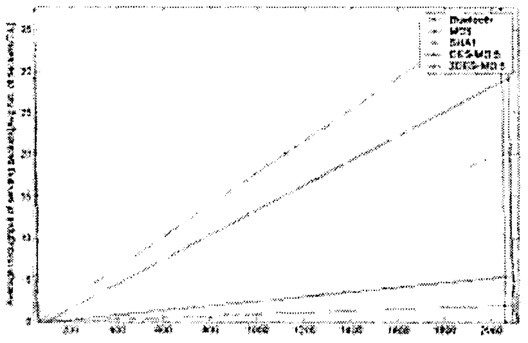


Fig. 11a: Packet size vs. average throughput of sending packets

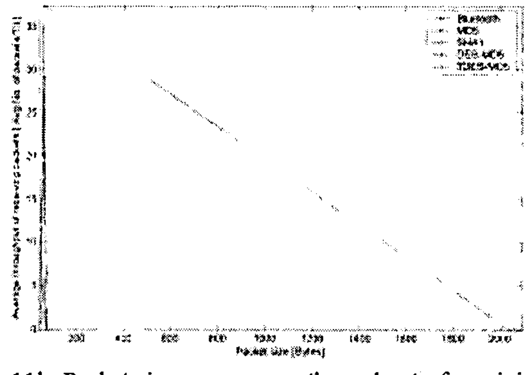


Fig. 11b: Packet size vs. average throughput of receiving packets

The throughput in Bluetooth environment is driven by the effect of the successive sending of constant packets, the erratic behavior of delay imposed by the encryption and decryption of the data, the erratic behavior of the Bluetooth wireless link and Wireless links are characterized by higher bit error rates and this causes inefficiencies in the operation of TCP.

We have not addressed the issues related to Bluetooth radio layer and polling algorithm at the baseband, where problems such as noise, interference and packet loss, may have a significant impact on performance (Misic and Jelena, 2003).

Over all and according to the results in the previous section, it has been proved that HMAC-MD5 performance better than HMAC-SHA1 for authentication and DES/ HMAC-MD5 reliable than 3DES/ HMAC-MD5 for Encryption.

REFERENCES

Anonymous, 1980. The Ethernet-A Local Area Network, September 1980. Version 1.0 Digital Equipment Corporation, Intel Corporation, Xerox Corporation.

Bluetooth, [referred 2001-02-11]. The Bluetooth Specification, v.1.1B. < http: //www. bluetooth. com/developer/ specification/ specification.asp >

BNEP_specification_rev10RC3 [referred 2002-12-17]. <www. grc. upv. es/localdocs /blue/ BNEP_specification_ rev10RC3. pdf >

Chaudhry, G., D. Medhi and J. Qaddour, 2002. Performance Analysis of IPSec Protocol: Encryption and Authentication, IEEE Communications Conference (ICC 2002).

Internet Engineering Task Force, 1996. A Standard for the Transmission of IP Datagram's over Ethernet Networks, RFC 894.

Internet Engineering Task Force, 1998. Classical IP and ARP over ATM, RFC 2225.

Internet Engineering Task Force, 1999. IPv4 over IEEE 1394, RFC2734. <http:// www.iana. org/assignments/ ethernet-numbers>.

Kelsey, J., B. Schneier and D. Wagner, 1999. Key Schedule Weakness in SAFER+, Second AES Candidate Conference.

Lu, Y. and S. Vaudenay. Faster Correlation Attack on Bluetooth Keystream Generator E0, CRYPTO 2004, pp: 407-425.

Misic, B. Vojislav and Jelena, 2003. Polling and Bridge Scheduling Algorithms in Bluetooth. <http:// www.cs.umanitoba.ca/~vmisic/pubs/tr0304.pdf>

Nomination of SAFER+ as Candidate Algorithm for the Advanced Encryption Standard (AES), 1998. Submission document from Cylink Corporation to NIST.

RFC 2401, 1998. Security Architecture for the Internet Protocol, available at<http: //www. ietf. org/rfc /rfc2401. txt>.

Saarinen, M.J., 2000. A Software Implementation of the BlueTooth Encryption. Algorithm E0, < http://www. jyu. fi/~mjos/e0.c >.

The network simulator NS-2. < http://nsnam.isi. edu/nsnam/index. php/User_Information>

UCBT - Bluetooth Extension for NS2 at the University of Cincinnati < http://www.ececs.uc.edu/~cdmc/ucbt>