An IPsec Gateway Based on the Intel IXP2400 Network Processor

Marc Brooker

October 18, 2005

Declaration

This report and the project on which it is based is entirely my own work.
I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the works of others has been attributed, cited and referenced.
I have not allowed, nor will allow, any other student to copy my work with the intention of passing it off as their own.
I acknowledge that plagiarism is wrong, and declare that this report, and the project on which it is based, is entirely my own work.

Marc Brooker

Abstract

The Intel IXP2400 is a powerful and flexible network stream processor which promises to offer superior performance to currently deployed solutions while remaining cost effective. This report presents the design of a Virtual Private gateway based on the IXP2400 processor, implementing a subset of the Internet Protocol Security (IPsec) protocol.

The design is chosen through analysis and comparison of multiple possible designs. Performance of the design is optimised by performing manual design space exploration on a performance critical section of the system.

A framework for performance evaluation by simulation is presented. This framework is used to perform a complete analysis of the performance of the gateway. Performance and Quality of Service factors such as throughput, delay and jitter and measured for a variety of packet loads and configurations.

A complete description of the Internet Protocol Security standard is presented, along with an analysis of the algorithms required for implementation of the standard.

Acknowledgements

Several individuals and institutions provided invaluable assistance to the author during the completion of this project.

Mr Neco Ventura Mr Ventura, in his capacity as adviser provided extremely valuable insight and assistance to the author during both the implementation and writing phases of the project and provided the equipment required for the project.

Eskom Eskom Transmissions North East has paid for the author to attend the University of Cape Town for four years.

Kate McWilliams Miss McWilliams has been extremely patient in providing assistance to the author.

Leslie Lamport, et al. Thanks to Leslie Lamport for LaTeX, Donald Knuth for TeX and Thomas Esser for teTeX.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Privacy and information security are extremely important features of current generation data and communications networks. The increasing reliance of companies, governments and individuals on broadband packet networks has created a great demand for security and privacy services for these networks. Users are unwilling, however, to trade reduced network speed for improved privacy.

In order to meet the demands of users for privacy and data security without sacrificing network throughput, systems must be design which can perform significant processing tasks on packets at line speed. Recent developments in network stream processor technology have brought devices to market which can meet the task of line speed data processing at a price users can afford.

By exploiting the incredible power of these devices, this project aims to create a Virtual Private Networking gateway capable of protecting data on broadband networks without reducing throughput.

1.2 Aims of the Project

The aims of the project are to investigate, design and test the software for the implementation of a Virtual Private Networking gateway based on the Intel IXP2400 Network Processor.

During the course of completion of the project, the following goals will be achieved:

- Investigate the feasibility of using the Intel IXP2400 in a dedicated Virtual Private Networking gateway.

- Investigate the effects such a gateway will have on network Quality of Service (QoS).
- Perform measurements of the real-world performance of the VPN gateway.
- Investigate the social and ethical implications of cryptography.
- Report on the findings of each of these investigations.
- Conclude on the implications of results and measurements.

1.3 Justification of Project Goals

1.3.1 Technical Justification

The recent rise in deployment of high speed networks has greatly increased the demand for systems which can process packets on these networks in real time. In response to these growing demands, various semiconductor companies introduced lines of Network Processors - special purpose microprocessors designed for packet and cell processing on broadband networks.

Virtual Private Networking protocols have risen in importance recently as companies, governments and individuals become aware of the importance of privacy, authentication and data integrity. A very effective way of providing these services securely and cost-effectively to all users on a network and remote access users is by making use of a Virtual Private Networking (VPN) gateway, which can provide both pre-configured and opportunistic privacy and authentication services to all data transmitted over a public network.

The convergence of virtual private networking technology, high speed networks and line speed packet processing promises to increase the utility of broadband networks and play a crucial role in ensuring that packet based high speed data networks can fulfill the communication requirements of any client.

Currently, however, high speed VPN hardware is extremely expensive and is unlikely to be affordable by small and medium business users, leading them to rely on traditional circuit switched networks for their secure communications and remote access needs. The development of an affordable, high speed Virtual Private Networking gateway would ensure that broadband networks can meet the requirements of all prospective users at a price they can afford.

In order for a VPN gateway to be useful, it must not prevent interactive protocols used over protected connections from functioning acceptably. Ad-

dressing Quality of Service (QoS) concerns inside the gateway would ensure that interactive protocols, such as Voice over IP, remain useful.

1.3.2 Social Justification

Broadband data network technology promises to bring high speed access to the information and communication potential of the internet to all potential users, especially the poor and those who live in rural areas. Universal deployment of these networks would provide an extremely powerful tool for social upliftment through education and access to communication technology.

With deployment of these technologies, however, comes concerns about the privacy and integrity of communications flowing over these networks. Data streams flowing over broadband networks could be copied (tapped) untraceably at nearly any point on the network.

Addressing these privacy concerns is essential to the public acceptance of data networks. Users will not rely on these networks for their telecommunications needs if they do not trust that their communications will remain private. Some potential users, such as rural hospitals, can not rely on data networks at all if they are not assured that data transmitted over these networks will remain completely private and confidential.

An affordable, high speed, Virtual Private Networking gateway would play a large role in ensuring that information travelling over all data networks can be trusted to remain private and uncorrupted.

1.4 Ethical Implications of Virtual Private Networks

The most important ethical issue surrounding the design and deployment of Virtual Private Networks is their use of cryptography for privacy and authentication. The use of cryptography has extremely important ethical and moral implications, which need to be carefully considered before a system which uses it can be developed.

Legitimate Users of Crytography

Most users who depend on cryptography for privacy, integrity and authentication services have a legitimate need for the services this technology provides. Legitimate users of cryptography include governments, companies, professionals and individuals.

Governments Governments rely on encryption for the protection of state secrets, secure communication within the government (especially during times of war) and secure communication with the governments of other countries. Cryptography can also play a critical role in the protection of public health records and other information which is protected by secrecy legislature.

Companies Companies use encryption to protect trade secrets and other intellectual property and protect themselves against industrial espionage.

Professionals Professionals such as Lawyers, Doctors and Engineers have an ethical, and in some cases legal, responsibility to protect information about their clients.

Individuals Many individuals seek to protect their privacy against both casual eavesdroppers and determined adversaries. Individuals might also rely on cryptography to protect their constitutional right to privacy.

Illegitimate Users of Cryptography

Criminals Organised crime organisations could use encryption to protect records of their crimes and communications regarding past and future crimes. Virtual Private Networks, coupled with VoIP, could thwart law enforcement surveillance and wiretapping.

Terrorists Terrorist organisations are likely to use encryption to prevent intelligence agencies from intercepting their communications.

It is clear that use of encryption by criminals is not desirable. Terrorism, however, is more subjective. Insurgents in countries with oppressive governments are likely to find themselves classified as terrorists by those governments. Few outside these governments, however, would argue that use of encryption by those seeking freedom is undesirable.

In his testimony to the Economic Policy Subcommittee of the United States house of representatives during hearings about export controls on encryption in October 1993, Phil Zimmerman (author of Pretty Good Privacy, the first wide spread encryption product for personal computers) said:

> "Some Americans don't understand why I should be this concerned about the power of Government. But talking to people in Eastern Europe, you don't have to explain it to them. They already get it– and they don't understand why we don't. I want to read you a quote from some E-mail I got last week from someone

> in Latvia, on the day that Boris Yeltsin was going to war with his Parliament:
>
> *"Phil I wish you to know: let it never be, but if dictatorship takes over Russia your PGP is widespread from Baltic to Far East now and will help democratic people if necessary. Thanks."* "

Cryptography and The Categorical Imperative

The categorical imperative is an important part of the philosophical basis of deontological ethics as defined by Immanuel Kant. Kant's first formulation of the categorical imperative - "Act only according to that maxim by which you can at the same time will that it would become a universal law."[1, 2] - is a useful tool for the analysis of the ethics of cryptography.

This formulation allows a person to ethically perform a particular act if, and only if, the rule under which they justify that act could apply to all. Based on this concept it is only ethically acceptable to research and use cryptography if such use can be extended to all possible researchers and users of cryptography without causing a greatly undesirable state of affairs. It is clear that this is not the case for cryptography and, based on this imperative, it is ethically acceptable for an individual to research or use cryptographic technology.

Conclusion

It is clear that preventing the general use of cryptography would lead to a more undesirable state of affairs than allowing free use of cryptographic technology. Researching, deploying and using cryptographic products is ethically allowable, if the end which will be achieved by the use of these technologies is ethically justifiable.

Most users of cryptography have a legitimate need for the services that this technology provides. Provision of cryptographic technology to legitimate users can be seen as a social good. Prevention of research into cryptography and technologies using cryptography is unlikely to prevent the harmful elements of society from gaining access to these technologies.

Chapter 2

Literature Review

2.1 Introduction

An extensive review of literature relevent to the implementation of Virtual Private Network protocols, particularly IPsec, on embedded systems was conducted. Papers relevent to the AES and SHA-1 algorithms, relevant multithreaded and embedded design issues and past work on network processors were also considered for inclusion in this review.

Extensive literature exists on the theoretical aspects of the algorithms and techniques used for design and implementation of the project. Very little literature exists, however, on practical and theoretical aspects of designing and developing applications for network processors.

2.2 Design and Implementation of IPsec Gateways

2.2.1 IPsec-based end-to-end VPN deployment over UMTS

In "IPsec-based end-to-end VPN deployment over UMTS"[4], Xenakis and Merakos describe the design and implementation of an end-to-end Virtual Private Network using IPsec over the UMTS (3G mobile) network.

The paper is concerned with the overall network design issues surrounding deployment of IPsec on mobile devices (MEs) attached to the UMTS network. Both qualitative and quantitative analyses of the performance and Quality of Service impact of IPsec are presented, along with a simulation environment for performance analysis based on the OPNET network simula-

tor. The proposed network architecture is not implemented and the results obtained are derived purely from the theoretically expected behaviours of network nodes.

The authors found that IPsec privacy services have a significantly higher cost in terms of network performance and processing time than authentication and data integrity services. It is concluded that

> "Security features may have an adverse impact on aspects of quality of service offered to the end-users and the system capacity. Data protection increases the required bandwidth, and security transformations reduce the performance in terms of throughput and delay."[4]

> "Clearly, there is a need to carefully choose the proper configuration of IPsec that is well suited for the application of interest. By trading off security with throughput-delay performance, a system engineer can work out a solution that balances the system real-time requirements."[4]

While the design requirements presented in this project differ greatly from those in the paper (dedicated network processing hardware versus low power mobile terminals), we have come to similar conclusions regarding the deployment of Virtual Private Networks.

2.2.2 Secure Wireless Gateway

In the paper "Secure Wireless Gateway"[5], Godber and Dasgupta present the design of an inexpensive IPsec gateway and access point for 802.11b wireless LANs.

The gateway presented in the paper was based on commodity hardware and software — a 133MHz Pentium laptop running OpenBSD 2.9. Routing, NAT, firewall and IPsec services on the gateway machine were provided with standard software provided with the OpenBSD operating system. Client-side IPsec services were provided by the native IPsec implementation on Windows 2000.

The authors found that the gateway as designed could be used with existing client side hardware and software and could provide effective security for 802.11b networks. The authors conclude that

> "With appropriate hardware support, a sufficiently fast processor and hardware-accelerated cryptography, this gateway solution could easily be constructed on a simple to install and configure embedded gateway platform. This embedded gateway could

> easily replace today's standard, vulnerable wireless access points which are quickly being mapped, probed, and very likely abused by potentially malicious individuals."[5]

While the IPsec gateway presented in this project does not provide all of the functionality required to replace existing access points it could be extended and paired with embedded access point hardware to provide secure access to wireless LANs.

2.2.3 Implementing IPsec

In "Implementing IPsec"[6], Keromytis, Ioannidis and Smith describe the design and implementation of the IPsec protocols on several POSIX compliant operating systems. The implementation is presented in the form of a set of kernel patches for the target operating systems, along with some support programs.

A performance analysis of the implementation was performed, based on testing the implementation with real-world IP traffic and recording throughput rates. While this paper focuses on the details of the implementation of the protocols in the OpenBSD and Linux kernels, it provided some insight into efficient methods for the implementation of IPsec.

2.2.4 Other IPsec Implementations

The most widely used IPsec implementations are those built into commodity operating systems for PCs and servers. Documentation of the design process of these protocol implementations is either not publically available or non-existant. This lack of literature makes analysis and comparison of these designs difficult, without extensive analysis of source code or functionality. IPsec implementations are available for Microsoft Windows 2000, XP and 2003, OpenBSD, FreeBSD, Apple Mac OSX and Linux.

The IPsec implementations built into commodity operating systems all appear very similar in functionality and it is unlikely that they differ widely in design.

Microsoft Windows

A security analysis of the design and implementation of IPsec in Microsoft Windows 2000 is presented in "Microsoft Windows 2000 Internet Protocol Security Review"[7], a technical report by Network Associates. The architecture splits IPsec functionality between Kernel Mode and User Mode, with

per-packet processing implemented in kernel mode and management and Security Association maintenance implemented in user mode.

FreeSWAN

FreeSWAN (and projects derived from it, such as OpenSWAN) is a widely used implementation of IPsec for the Linux operating system. Packet processing is performed in kernel mode by a kernel module, KLIPS, which interfaces with the TCP/IP stack of the Linux kernel. Key exchange and security association maintenance are provided by userspace daemons.

Commercial Embedded IPsec Gateways

Many network equipment manufacturers offer complete product lines of embedded IPsec gateways and network devices with built in Virtual Private Networking capabilities. Compared to the performance of software IPsec implementations running on commodity server hardware, the performance of dedicated embedded IPsec gateways is extremely good.

For example, the Cisco ASA 5540 security appliance combines a high speed IPsec gateway with extensive firewall and bridging capabilities. The maximum IPsec throughput of the ASA5540 is rated at 325 $MBit$ per second - approximately three times faster than the highest rate delivered by this project. In October 2005, the listed price of the ASA 5540, including user licences, was approximately US$18000 (R117000).

2.3 Performance Analysis of Systems

The papers presented above all contain performance analyses of the designs presented and some discussion of the performance analysis methodologies used. “IPsec-based end-to-end VPN deployment over UMTS”, in particular, provides an extensive discussion of performance analysis.

2.3.1 An Approach for Quantitative Analysis of Application Specific Dataflow Architectures

The paper “An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures”[8] by Kienhuis, Deprettere, Vissers and van der Wolf presents “an approach for quantitative analysis of application-specific dataflow architectures” which “allows the designer to rate design alternatives in a quantitative way and therefore supports him in the design process to find better performing architectures.”

The paper focuses on the optimisation of signal processing applications on specific data flow orientated systems, however some of the methods presented are widely applicable to performance analysis of software implementations. The method used for development of the simulation environment is widely applicable. The approach to system analysis by simulation used in this project was partially based on the techniques described in this paper.

2.3.2 Design Space Exploration for Real-Time Embedded Stream Processors

In "Design Space Exploration for Real-Time Embedded Stream Processors"[9], Rajagopal, Cavallaro and Rixner present a framework for performing design space explorations on hardware designs. Design space exploration is a technique for guided or automated exploration of a design parameter space using simulation, qualitative or quantitative models to optimise a design for a particular task.

In the paper, the design space exploration technique was used to perform optimisation of the design of stream-processor hardware for a particular application. A similar, though non-automated, technique of design space exploration was used in this project to explore the relationship between system performance and design parameters.

2.3.3 IPsec Forwarding Application Level Benchmark

The "IPsec Forwarding Application Level Benchmark"[10] implementation agreement is a standard for the testing of network processor based IPsec implementations published by the Network Processor Forum. The agreement defines test procedures, standard performance metrics and results reporting procedures for performance testing of IPsec implementations.

The test procedure described in the agreement uses two identical devices as the devices under test (DUT), a packet generator and a traffic analyser. Packets are generated by the generator at the highest rate that does not cause packet loss and parameters such as throughput and delay are recorded by the packet analyser.

The testing procedure recommended by this agreement is very similar to the one implemented in this project. This document provides a basis for the validity of the results of the testing procedure used to analyse the design detailed in this project.

2.4 Comparison of Virtual Private Networking Protocols

A complete comparison of Virtual Private Networking protocols is presented in Appendix B. Literature relevant to each protocol is listed and analysed. The comparison includes functionality, security and performance analyses of each protocol.

2.5 Conclusion

The design of IPsec implementations for a wide variety of applications and systems is widely documented, with literature covering nearly all aspects of the implementation of the IPsec protocols. Very little literature exists, however, discussing the design aspects of implementing processing-time bound applications such as IPsec on network processors. While it is clear that IPsec implementations for network processors have been designed and completed in industry, these designs are not documented in publically accessable literature.

Performance analysis of embedded systems and applications is well covered by existing literature — extensive documentation exists covering previous achievements in this area. The existing literature was found to be extremely helpful in the development of simulation and analysis approaches in this project.

Chapter 3

Comparison of Alternative Designs

3.1 Required Operations

Each packet received from the internal network has to be processed through a number of steps before being dispatched to the external network. These steps are:

Receive The packet is received from the Media and Switch Fabric (see Section A.2.3 on page A-6) interface and copied to the IXP2400's memory.

Encrypt The packet contents are encrypted using the AES algorithm.

Hash A cryptographic hash of the encrypted packet's contents is calculated using the SHA-1 algorithm.

Tunnel The encrypted packet is encapsulated within a tunnel mode IPsec packet (see Section C.1.3 on page C-3) and the ESP header and footer are added.

Transmit The packet is copied from the IXP2400's memory, through the MSF and out over the network.

3.2 Naive Design

3.2.1 Motivation for Design

The most obvious design is to use a single processing core to perform all of the required steps in order. This design would map well onto traditional

microprocessors which can only process a single instruction at a time and do not provide support for efficient multithreading.

The naive design is the most simple of the designs presented here, and will likely be the easiest to implement and debug.

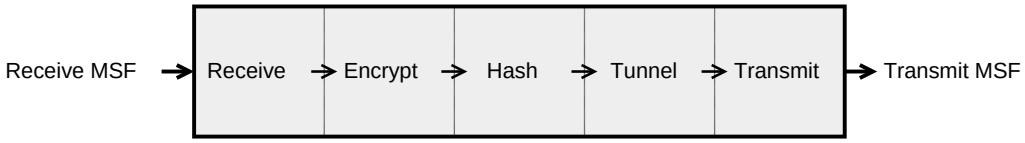


Figure 3.1: *Naive* packet processing design

3.2.2 Design Details

In the naive design, one packet at a time will be processed completely before the next one is accepted from the MSF receive buffer. All the processing will be handled by a single Microengine program running on just one of the IXP2400's eight Microengines.

3.2.3 Analysis of Design

The naive design makes very inefficient use of the resources of the IXP2400. Only one of the processor's eight available packet processing engines (Microengines) are used, severely limiting the possible throughput of this design.

The time taken for a single packet of size P to be processed completely and dispatched to the network is $(\tau_r(P) + \tau_e(P) + \tau_h(P) + \tau_m(P) + \tau_t(P))\,\mu s$ where $\tau_r(P)$, $\tau_e(P)$, $\tau_h(P)$, $\tau_m(P)$ and $\tau_t(P)$ are the time take in microseconds to receive, encrypt, hash, encapsulate and transmit a packet of size P bits, respectively. The throughput of the design is therefore

$$R_{naive} = \frac{P}{\tau_r(P) + \tau_e(P) + \tau_h(P) + \tau_m(P) + \tau_t(P)}$$

megabits per second, for a packet stream with mean packet size P.

The time taken to copy a packet from the receive buffer and copy a packet to the transmit buffer is two orders of magnitude shorter than the processing time required for the encrypt, hash and encapsulate operations. The transmit and receive operations can therefore be ignored when calculating the transmit rate. The times taken by the processing functions are linearly related to the packet size, suggesting that the throughput rate will not be sensitive to the

packet size. Therefore, the throughput rate of the design will be

$$R_{naive} = \frac{1}{\tau_e + \tau_h + \tau_m}$$

megabits per second where τ_e, τ_h and τ_m are the times taken by each processing task to process a single bit, in microseconds.

3.3 Pipelined Design

3.3.1 Motivation for Design

Pipelining is a design technique for continuous processes which can greatly improve the bandwidth of the process at the cost of increased process latency. It is widely used in most modern microprocessor designs (especially larger processors such as the Intel Pentium line) and is readily applicable to nearly any process. Pipeling is especially suited to the task of packet processing, as the result of the processing of one packet does not depend on the results of processing of any other packet. This independence means that the design can be extremely simple, without any of the complexity required to pipeline interdependent processes.

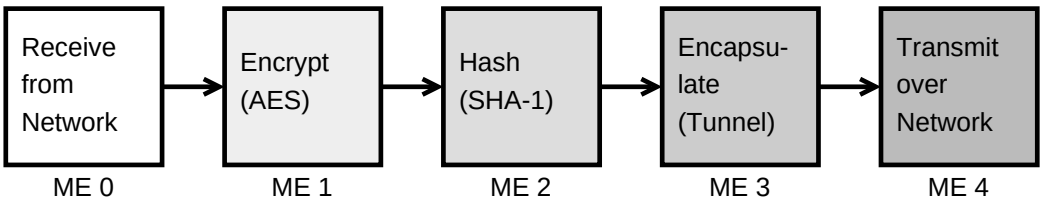


Figure 3.2: *Pipelined* packet processing design

3.3.2 Design Details

As discussed in Section A.2 on page A-2, the IXP2400 has eight simple DSP-like packet processing cores, called Microengines. Each Microengine has a very limited amount of fast memory and instruction store. The most obvious mapping of the required packet processing tasks to Microengines is to map each task to a single core.

Packets are passed through this pipeline in such a way that each stage of the pipeline processes one packet at a time, with a total of five packets in flight. The pipeline is advanced when the slowest processing stage of the line has completed it's task.

3.3.3 Analysis of Pipeline Design

The complexity and time required for the five steps of the pipeline vary greatly. The *receive* and *transmit* steps of the process take up to two orders of magnitude less time than the *hash* and *encrypt* steps. This effectively reduces the throughput of the entire pipeline to that of the slowest step in the line.

Simulation results revealed that the *encrypt* step of the pipeline takes the longest time. If the mean packet size is P bits and the time taken to encrypt a packet of size P is $\tau_e(P)$ microseconds, then the pipeline will advance every $\tau_e(P)\mu s$ and the time taken to encrypt a packet will be $5\tau_e(P)\mu s$. When the pipeline is full, a packet will be issued each time the pipeline steps, which leads to a mean throughput of $R_{pipeline} = \frac{P}{\tau_e(P)}$ megabits per second, where $R_{pipeline}$ is the throughput rate, P is the mean packet size and $\tau_e(P)$ is the time required to encrypt a packet of size P. As discussed in Section 3.2.3, $\tau_e(P)$ is a linear function of P, therefore the throughput of the pipelined design will be

$$R_{pipeline} = \frac{1}{\tau_e}$$

megabits per second, where τ_e is the time required to encrypt a single bit.

The pipeline design only uses five of the eight Microengines present on the IXP2400 processor - four of which are likely to be underutilised as they need to wait for the encryption step to complete before continuing with the next packet in the stream.

3.3.4 Superscalar Design

> "If one pipeline is good, then surely two pipelines are better." [11]

In order to extend the pipelined design to make better use of the available resources, more than one pipeline can be used for the most time consuming processing tasks. Illustrated in Figure 3.3, the multi-pipeline duplicates the *encrypt*, *hash* and *encapsulate* operations into two independent pipelines. Each pipeline independently receives packets from the a dedicated *receive* Microengine, processes it and transfers it to the *transmit* Microengine for transmission over the network.

3.3.5 Design Details

As discussed in Section 3.2.3 on page 13, the *receive* and *transmit* stages of the pipeline are several orders of magnitude faster than the remainder

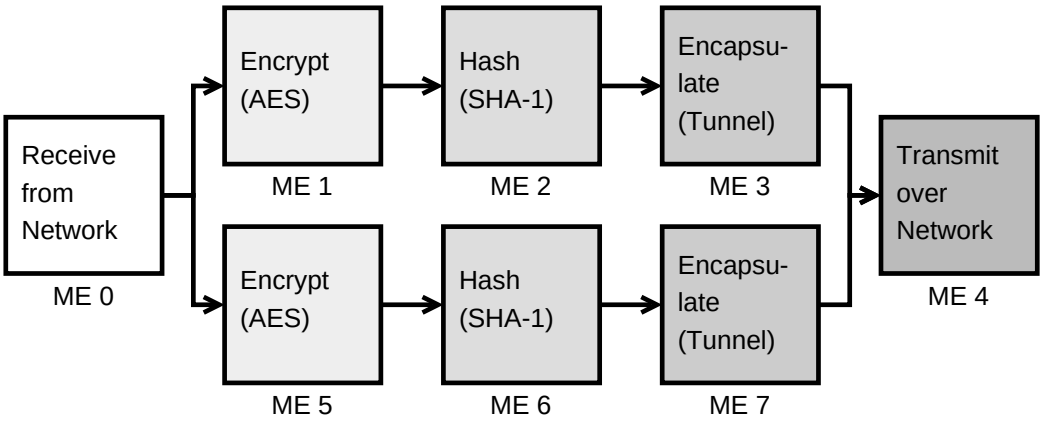


Figure 3.3: *Multi-pipeline* packet processing design

of the pipeline stages. It is therefore clear that, by duplicating the three pipeline stages which take up the majority of the required processing time, the throughput of the design has been doubled, with no effect on packet latency.

The throughput of the superscalar design is $R_{superscalar} = 2R_{pipeline} = \frac{2}{\tau_e}$ megabits per second and the latency for a single packet is unchanged from that of the pipelined design at $5\tau_e(P)$.

While the superscalar design is twice as fast in throughput as the pipelined design and makes better use of the available resources on the IXP2400, it does not take full advantage of the data level parallelism inherent in network packet streams. The speed is further constrained by forcing all pipeline stages to wait for one time consuming stage to complete it's task.

3.4 Parallel Design

3.4.1 Motivation

Streams of data packets have a large amount of inherent data level parallelism. Network processors include multiple processing cores to allow the programmer and system designer to take advantage of the available parallelism. The Intel IXP2400, which is used in this project, includes eight Microengines - simple processor cores designed to perform packet processing tasks in parallel.

Any software design which seeks to make efficient use of a network stream processor must exploit the data level parallelism inherent in network data flows.

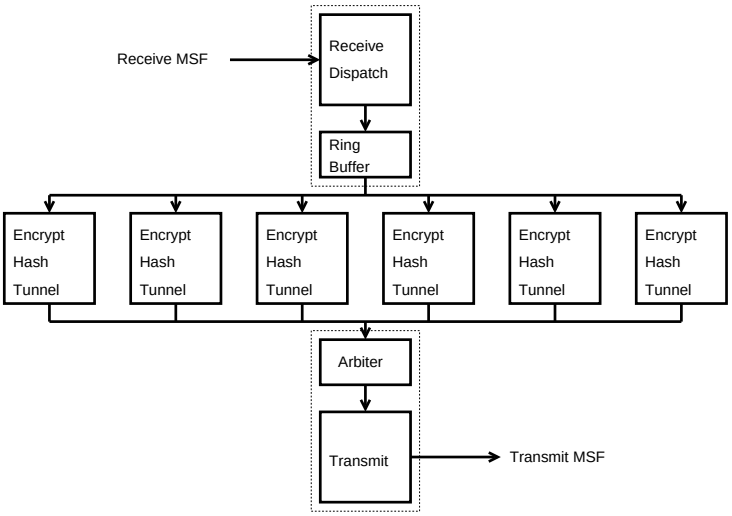


Figure 3.4: *Parallel* packet processing design

3.4.2 Design Details

In contrast to the pipelined design presented above, in the parallel design a single Microengine was programmed to perform the *encrypt*, *hash* and *encapsulate* steps. Joining tasks in this way, it is possible to have six Microengines perform the time consuming parts of the packet preparation process in parallel, greatly increasing the efficiency of use of the resources available on the IXP2400.

A single Microengine was used to receive packets from the MSF interface. Once the receive process is complete for a single packet, it is placed by the receive Microengine (producer) into a large ring buffer. Idle processing Microengines (consumers) can then take packets out of this buffer and process them.

Another Microengine was dedicated to the transmission of completed IPsec packets over the network. This microengine also provides an arbitration service which decides which completed packet to send first. Arbitration at this stage could be used to implement explicit Quality of Service models on the gateway.

3.4.3 Analysis of the Parallel Design

Assuming a network traffic distribution that will keep the MSF input buffer partially full at all times, the internal receive buffer will always contain enough packets that the processing microengines will not have to wait for data to process. The internal receive buffer therefore decouples the time

taken to receive a packet from the processing time.

Each processing Microengine will transmit packets into the internal transmit buffer with a rate of $R_{ME} = \frac{P}{\tau_e(P)+\tau_h(P)+\tau_m(P)}$ where P is the mean packet size, $\tau_e(P)$, $\tau_h(P)$ and $\tau_m(P)$ are the time taken to encrypt, hash and encapsulate a packet of size P, in microseconds, respectively. Given that $\tau(P)$ is a linear function of P (see Section 3.2.3 on page 13), the overall packet transmission rate of the parallel design is

$$R_{parallel} = \frac{6}{\tau_e + \tau_h + \tau_m}$$

megabits per second.

The derivation of the mean packet delay and the probability that the receive buffers will be full for a given traffic distribution for this design is fairly complicated, and can be found from the simulation data presented in Chapter 5.

3.5 Comparison Of Designs

In order to compare the throughput rates of the three design alternatives presented here it must be assumed that, in each case, the packet arrival rate is identical to the packet departure rate. This ensures that there will always be a packet available in the receive buffer when one is required and that the receive buffer never overflows. This is not a reasonable assumption for absolute performance analysis, but greatly simplifies the analysis of relative performance at no cost to accuracy.

Comparison of Transmission Rates

The throughput rates for the four designs discussed above, are

$$R_{naive} = \frac{1}{\tau_e + \tau_h + \tau_m}$$

$$R_{superscalar} = \frac{2}{\tau_e}$$

$$R_{parallel} = \frac{6}{\tau_e + \tau_h + \tau_m}$$

It is clear that $R_{parallel} = 6R_{naive}$ - the parallel design is potentially six times faster than the naive design. Comparing the superscalar and parallel designs, the relationship $\frac{R_{parallel}}{R_{superscalar}} = \frac{3\tau_e}{\tau_e+\tau_h+\tau_m}$ can be calculated. Since the encrypt

step is the most computationally intensive step, $3\tau_e > (\tau_e + \tau_h + \tau_m)$ and $R_{parallel} > R_{naive}$.

In terms of throughput rate, the parallel design is the fastest, the superscalar design is second and the naive design is slowest - slower than the parallel design by a factor of six.

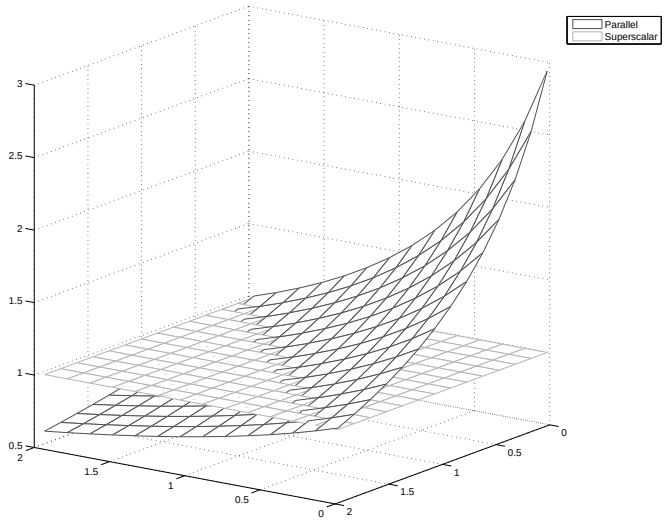


Figure 3.5: Performance of Superscalar Design vs Parallel Design

Figure 3.5 illustrates the relationship between the throughput performance of the superscalar design and the parallel design. Relative performance is graphed on the vertical axis. The x axis is the relationship $\frac{\tau_e}{\tau_h}$ and the y axis is the relationship $\frac{\tau_e}{\tau_m}$. It is clear from this graph that where $\tau_h + \tau_m < 2\tau_e$, the parallel design is superior.

3.6 Conclusion

The parallel design, presented in Section 3.4 on page 16, is the most efficient of the design alternatives presented above in terms of throughput and use of the IXP2400's available resources. The complexity of this design is similar to that of the superscalar design, but is much higher than the complexity of the naive design.

As the complexity of all the designs is manageable, the parallel design was chosen as the best of the four alternatives. The implementation and analysis of this design is presented in the following chapters.

3.7 Design Optimisation of Packet Processor

3.7.1 Introduction

The packet processing program performs all the packet processing operations required to encrypt, hash and encapsulate the data stream. In the final software design, six instances of this program run, one per Microengine, to take advantage of data level parallelism inherent in packet streams.

Transferring data to or from the DRAM and SRAM interfaces of the IXP2400 is a relatively slow process and it is likely that the data processing program will spend a significant amount of time waiting for data to be transferred over these interfaces. As illustrated in Table 3.1 and further discussed in Section A.2.1 on page A-4, copying data to external memory has a minimum latency of 150 to 300 clock cycles, plus transfer time.

Table 3.1: IXP2400 Types and Properties of Memory (from [12])

Memory Type	**Maximum Size**	**Access Latency**	**Bus Width**
Local	2560 bytes	3	4
Scratch	16K	60	4
SRAM	256M	150	4
DRAM	2G	300	8

In a single threaded design, time taken waiting for memory transfers is wasted and can greatly reduce the effective processing speed of the Microengine. Memory access times can dominate execution times for processes which need to access a significant amount of memory, or make a large number of small writes to memory. This effect can be reduced through judicious use of write and read combining, or by executing multiple threads on each Microprocessor core.

The IXP2400 Microengine provide zero-overhead context switching for up to eight threads per Microengine core. Threads which are ready to run can be executed by those which are currently waiting on memory access. Using multithreading can greatly reduce the performance effect of external memory access and improve the effective processing rate of the Microengines.

Multithreaded programs can make better use of data level parallelism and increase the number of packets that can be processed concurrently.

3.7.2 Limited Memory Sizes

The largest challenge faced when writing multithreaded programs for the IXP2400 Microengines is the severe limitation on the size of fast memory.

Per-Microengine *local* memory is similar in speed to the L1 cache of traditional large microprocessors, but is severely limited in size to 2560 bytes. *Scratch* memory is shared between the Microengine cores and offers similar throughput and latency to the L2 cache on large microprocessors, such as the Intel Pentium 4 (see Section A.2.1 on page A-4).

External memory interfaces (SRAM and DRAM) have less bandwidth than the internal memory interfaces and have longer access times. Access latency times for these buses can be worse than those listed in Table 3.1 if the external buses are accessed concurrently by more than one Microengine.

If all the data required by a program cannot fit into *local* and *scratch* memory, some data must be moved to external memory. Which data is copied to external memories can be found empirically by analysing data access patterns. If possible, small tables or datasets which are accessed frequently must be stored in one of the fast local memories, while large datasets which are accessed infrequently should be stored in one of the slower external memories.

The extremely limited size of fast memory present on the IXP2400 creates a tradeoff between running multiple threads and running fewer threads faster by placing their working set into faster memory ares. In order to fully optimise a program for the IXP2400, a balance must be found between parallelism and memory usage.

3.7.3 Design Alternatives

Two alternative designs for the packet processing Microengine program were designed, implemented and tested. These designs represent two alternative approaches to the optimisation of the program. The first design sacrifices execution speed per thread for more threads, while the second optimises the execution time per thread, but reduces the number of threads that can fit into the available memory area.

Eight Threads, Slower Memory

In this design, all eight possible hardware threads supported on the IXP2400 Microengine were used to take maximum advantage of data level parallelism in the input data. Eight packets and sets of temporary variables need to be stored in memory, along with a single copy of the 4 kilobyte lookup tables required for AES. Some of the temporary variables required for the processing units were stored in external SRAM and packet data was read one SHA-1 block (320 bytes) at a time from DRAM.

Accessing external memory increases the time it takes to process a single packet and reduces the throughput of the packet processor.

Three Threads, Fast Memory

The second design utilised three hardware threads per Microengine. More of the working set of each program can be placed into fast memory areas, reducing the number of accesses to the external memory bus.

All of the temporary variables which are required for packet processing were kept in fast local RAM, along with the tables required for the AES and SHA-1 algorithms. Packet data is read into local memory for processing 320 bytes (one SHA-1 block) at a time.

This design reduces the amount of time it takes to process a single packet, effectively increasing the possible throughput of each thread of the packet processor.

3.7.4 Analysis of Design Alternatives

Quantitative analysis of the performance of Microengine programs is extremely difficult. One reason for this difficulty is that factors such as memory access times vary greatly with load on the memory buses and this load varies dynamically with the execution patterns of multiple threads on multiple cores.

In order to establish which design performs better in real-world conditions, a simulation was performed (based on the models presented in [8] and [13] for performing simulations of design alternatives) using the cycle-accurate simulator built into the Intel Developer Workbench development environment. A mixture of different sized packets, similar to real-world Internet packet loads was created and run through the two alternative packet processors. The simulation results were recorded for approximately 60 ms, after which the behaviour of the system was found to be constant. The simulation environment used was the one developed for evaluation of the final design, detailed in Section 5.1 on page 36.

The results of the simulation are presented in Figure 3.6. The graph shows the amount of data transmitted by the entire system, consisting of six packet processing Microengines with each engine running the eight or thread thread variants of the packet processing program.

The system based on the eight thread packet processing program transmitted approximately 770kB of data in 0.05 seconds, corresponding to a data rate of 15400kB per second, or 120Mbit per second. The mean throughput

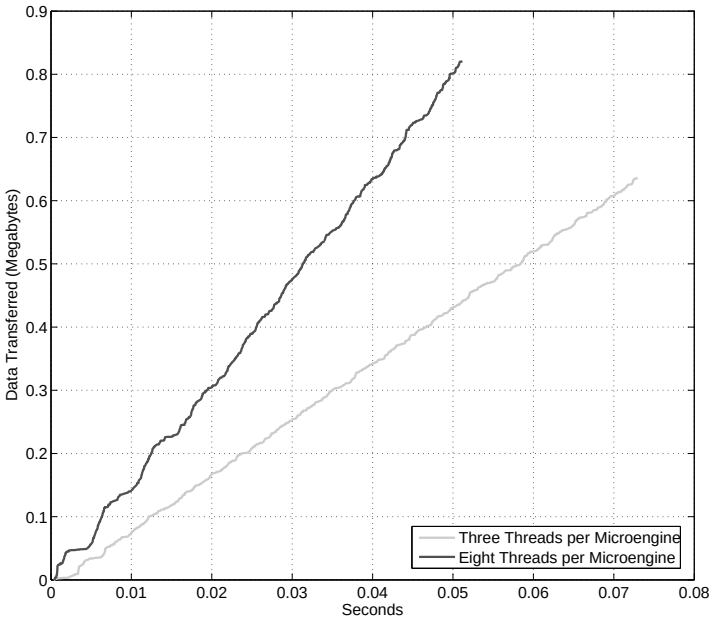
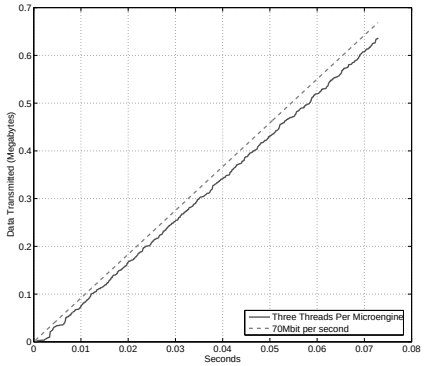
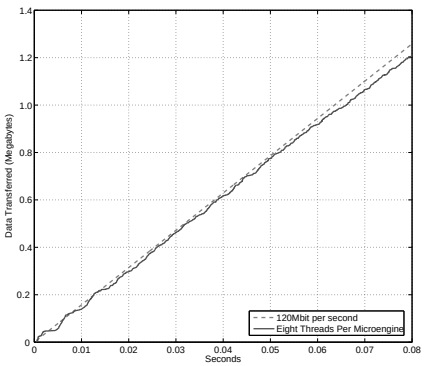


Figure 3.6: Comparative Throughput of Packet Processor Design Alternatives



(a) Three Threads: Throughput approx. 70Mbit/s



(b) Eight Threads: Throughput approx. 120Mbit/s

Figure 3.7: Throughput of Packet Processing Alternatives vs Benchmarks

per microengine thread is therefore $\frac{120}{6\times8} = 2.5\ Mbit$ per second. Approximately three microseconds of Microengine thread time are required per byte - corresponding to approximately 1830 Microengine clock cycles per byte.

The system based on the three thread packet processing program transmitted approximately 432kB of data in 0.05 seconds, at a mean data rate of 8640kB per second or 68Mbit per second. The average throughput per microengine thread is $\frac{68}{6\times3} = 3.75\ Mbit$ per second. This design uses two microseconds of Microengine thread time per byte, on average - corresponding to approximately 1220 clock cycles.

It is clear that despite the three thread design offering 50% superiour throughput per thread, the larger number of threads of the eight thread design outweighs the per-thread advantage of the three thread design. The throughput advantage of the eight thread design can be approximated by

$$\frac{T_{eight}}{T_{three}} = \frac{2.5}{3.75} \times \frac{8}{3} = 1.78$$

This result correllates closesly with the results presented in Table 3.2, indicating that the eight thread design is approximately 80% faster than the three thread design.

Table 3.2: Data Transferred vs Time for Packet Processor Design Alternatives

Seconds	**Three Thread**	**Eight Thread**	**Difference**
0.01	73722	140212	90%
0.02	167986	298226	78%
0.03	254334	463064	82%
0.04	342810	616708	80%
0.05	431936	776190	80%

Chapter 4

Final Software Design

4.1 Introduction

In Chapter 3 four software designs for an IPsec gateway was compared and the best design chosen. The final design makes use of the six of the IXP2400's microengines to perform packet processing in parallel, one microengine to receive packets from the network and control allocation to processing microengines and one Microengine to transmit packets over the network.

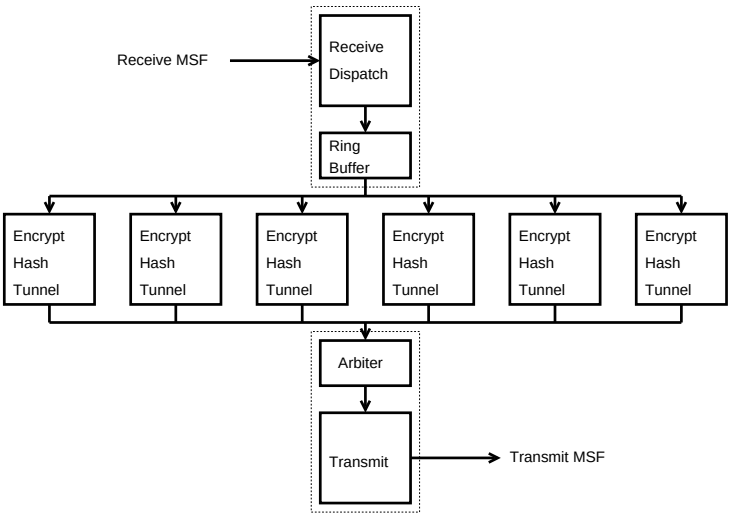


Figure 4.1: Final Packet Processing Structure

Illustrated in Figure 4.1, this design makes optimal use of the processing resources present on the IXP2400. As discussed in Appendix A, the Intel IXP2400 network processor has eight high speed DSP-like cores called Microengines, all eight of which are utilised by this design.

In order to implement this design, three separate Microengine programs must be designed and constructed - the receive program, the processing program and the transmit program.

4.2 The Receive Program

4.2.1 Required Functionality

The Receive microengine program is required to perform two tasks:

1. Receive Packets from the network
    - Interface with the Media and Switch Fabric Interface to receive *mpackets*
    - Reassemble *mpackets* into complete Ethernet packets
    - Copy Ethernet packets into RAM for processing
2. Dispatch Packets to the *Encrypt* Microengines
    - Provide a store for packets awaiting processing
    - Allocate packets to *Encrypt* Microengines in an efficient manner

The two tasks are implemented as separate threads (or contexts) on a single Microengine. Efficient communication between the threads is ensured by utilising shared memory areas.

4.2.2 Reception of Network Packets

The IXP2400's Media and Switch Fabric interface passes network packets to the microengines in the form of *mpackets* - small constant sized packets - which must be reassembled into packets before packet processing. The size of the mpackets is configurable to 64, 128 or 256 bytes per *mpacket*. *Mpackets* are a constant size, but may contain a variably sized payload if a network packet does not fit into a whole number of *mpackets*.

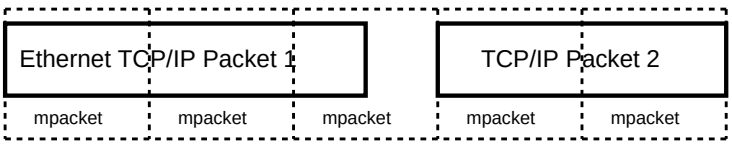


Figure 4.2: Ethernet TCP/IP packets split into *mpackets*

Each *mpacket* has an associated Receive Status Word (RSW). The RSW specifies the size of the mpacket's payload, whether any errors occurred during the reception of the mpacket and whether the mpacket passes a simple parity check. The RSW also includes the SOP and EOP bits. The SOP bit indicate whether the mpacket is the first mpacket of a network packet and the EOP bit indicates whether it is the last. For packets which fit entirely into a single mpacket, both the EOP and SOP bits will be set.

Receive State Machine

The Finite State Machine design model is ideally suited to simple state-based tasks such as the reassembly of network packets from a stream of frames or *mpackets*. This model was used to provide an extremely simple, robust reassembly program which can handle all possible errors and inconsistencies in the *mpacket* stream. The possible states are:

WAIT The program waits for an *mpacket* to arrive with the SOP bit set indicating that it is the first part of a packet.

CONTINUE The program collects *mpackets* and reassembles them into a buffer in RAM.

COMPLETE The final mpacket of the packet is copied into the buffer. Parameters such as the size of the complete packet are recorded.

DONE The packet is dispatched to the processing Microengines.

ERROR All *mpackets* for the rest of the current network packet are discarded.

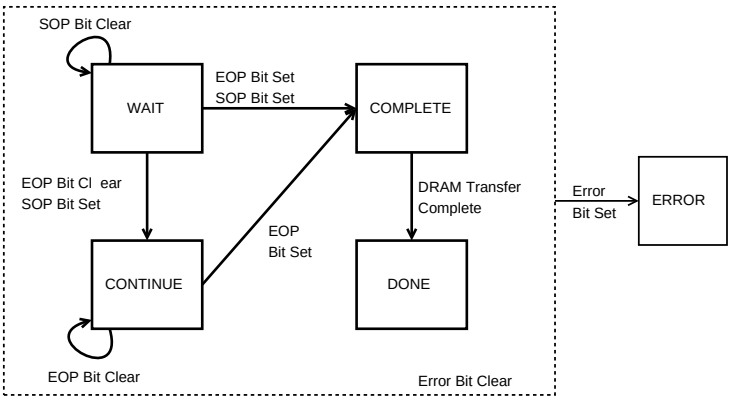


Figure 4.3: State Transition Diagram for Packet Reassembly Process

4.2.3 Allocation of Packets to Processing Units

The receive Microengine allocates received packets to one of the packet processing engines to be encrypted, hashed and encapsulated. Packets must be allocated to available idle Microengines or placed into a buffer if all the processing Microengines are busy.

4.2.4 First In First Out (FIFO) Queue

A FIFO queue was used to store packets awaiting consumption by the processing Microengines. The use of a FIFO queue ensures that packets are serviced in the order in which they arrive, minimising worst-case waiting times when the processing units are saturated with packets. Implementation of the FIFO queue as a ring buffer would minimise the number of memory accesses that are required when adding or removing data from the queue.

Ring Buffers are a simple data structure consisting of a linear array of memory, a read pointer and a write pointer. When data is written to the buffer, it is written at position of the write pointer and the write pointer is advanced. When data is read, the read pointer is advanced to point to the next full slot. If the read and write pointers point to the same element then the buffer is either full or empty and further reading or writing is blocked until the next write or read.

This data structure can be extended to the multiple producer-multiple consumer case by replacing the read and write pointers with a bitmap indicating which slots are full. Fast hardware support for bit-level operations on the IXP2400's Microengines makes this data structure extremely efficient to implement.

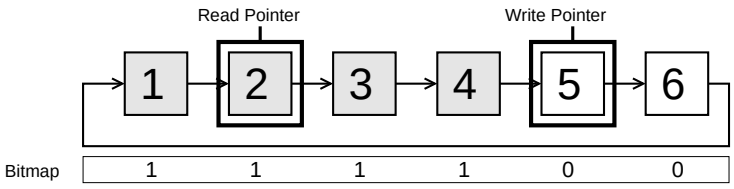


Figure 4.4: Ring Buffer FIFO Queue

A different queue structure could be used to achieve Quality of Service goals for prioritised data streams. The implementation of a priority queue in place of the FIFO queue would not add significant complexity to the design of the packet distribution code. Using a priority queue, packet processing could be prioritised according to size, source, destination or any other criterion.

Monitors

Monitors are a widely used software design pattern that allows multiple parallel processes or threads to have shared access to a limited resource. A monitor consists of a mutex (or mutual exclusion variable) which only one process can hold at a time and a set of functions for interacting with the available resources. Monitors provide the same functionality as semaphores[15], but are simpler to implement in some cases.

A Monitor design pattern was used to ensure that the FIFO queue could safely be accessed by multiple consumers. Providing safe asynchronous multiple consumer access to the packet buffer allows the packet processing units to proceed independently until interaction with the Monitor is required. At this point, if another process is currently accessing the buffer then the process is forced to wait until it can acquire the mutex.

The IXP2400 Microengines provide hardware support for Mutual Exclusion variables (Mutexes), including atomic test-and-set operations.

4.3 The Transmit Program

4.3.1 Required Functionality

The Transmit program is required to perform two tasks:

1. Arbitrate access to transmission resources
2. Send packets to the Media and Switch Fabric (MSF) interface for transmission over the network

The two tasks are implemented in a single thread as they are inherently serial in nature - the next packet cannot be accepted for processing until the previous packet's transmission has been completed.

4.3.2 Arbitration of Transmission Resources

Processing microengines communicate with the transmission microengine program through an extremely simple shared memory interface. Simultaneous access to this interface by multiple processing engines is prevented by the use of a mutual exclusion variable (mutex). Microengines which are ready to transmit packets idle until the mutex becomes available.

When a processing microengine takes possesion of the mutex, the following sequence is followed:

1. The **processing** ME asserts a *ready-to-send* shared variable.

2. In response the **transmission** ME asserts a *clear-to-send* shared variable.

3. The **processing** ME copies the address and size of the packet to shared variables.

4. The **transmission** ME copies the packet to the transmit buffer.

5. The *ready-to-send* and *clear-to-send* shared variables are cleared.

Following the completion of this processes, the transmit buffer is sent out over the network and the processing ME is freed to process another packet.

4.3.3 Transmission of Packets

Data is transferred to the transmit MSF using memory mapped IO. Properly formatted data is copied to a series of *transmit buffers* which, when signalled to do so, the transmit MSF will send out over the network. The complete data transmission process is as follows:

1. Break packet down into one or more fixed size *mpackets*

2. Query MSF to find which transmission buffers are free

3. Copy each *mpacket* to a free transmission buffer

4. Write the status words for the used transmission buffers in the correct packet order

Data transmission occurs asynchronously - the transmit thread can start processing the next packet while the current packet is being transferred over the network.

4.4 The Packet Processing Program

4.4.1 Required Functionality

The packet processing program is required to perform four tasks:

1. Encrypt the packet data using the AES algorithm

    - Break packet up into AES blocks (16 bytes) and add padding to the final block, if necessary
    - Use AES algorithm to encrypt the blocks

- Reassemble the ciphertext into a block of memory

2. Hash the enciphered packet data using the SHA-1 algorithm
3. Construct the Encapsulating Security Payload header and footer
4. Encapsulate the enciphered packet within an IPsec tunnel mode packet

All three tasks are performed by a single thread on the packet processing microengines. Multiple packet processing threads are used in parallel to improve performance and take advantage of data-level parallelism.

4.4.2 Design of Packet Processing Program

In Section 3.7 on page 20, two alternative designs for packet processing microengine programs were presented and compared. The final design uses eight concurrent threads per Microengine to improve the performance of packet processing operations. The use of eight threads effectively greatly reduces

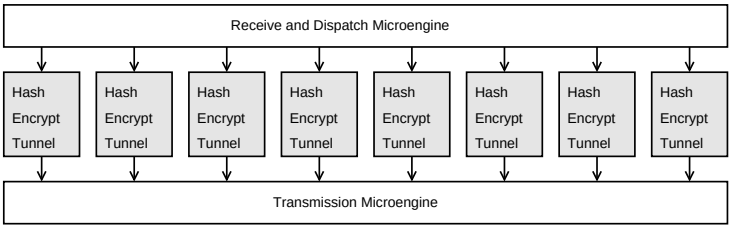


Figure 4.5: Design of Packet Processing Program

the effect of memory access latency and increases the number of Microengine clock cycles which are spent performing useful work, instead of waiting on memory access.

4.4.3 Encryption Operation

The first task required in the process of creating an Encapsulating Security Payload packet is the encryption of the original packet data. The ESP protocol (see Section C.1.3 on page C-3) originally specified two cryptographic algorithms, NULL and DES[16]. NULL is a placeholder in the protocol for no encryption, data is simply passed through unencrypted[17]. DES is the Defence Encryption Standard[18], a cryptographic algorithm which, due to it's small key size, is considered insufficiently secure for new IPsec systems.

The ESP standard was extended in 2003[19] to include the Advanced Encryption Standard in it's list of cryptographic algorithms. The AES algorithm (discussed fully in Section C.4 on page C-10) offers excellent security along with good performance in both software and hardware implementations. AES supports three different key sizes: 128bits, 192bits and 256bits. Support for 128bit keys is required for compliance with the standard, support for the other sizes is optional.

The most efficient implementation of the AES algorithm on 32 bit hardware (such as the IXP2400 Microengines) makes use of a set of four 1 Kilobyte look up tables for performing the majority of required operations. In this implementation, a single copy of these tables is stored in *scratch* memory and is shared between all the processing microengines.

Before the encrypt step the packet data is broken up into 16 byte blocks, as required by the AES algorithm. If the available data does not fit into an integral number of blocks, padding data is added to the final block to make the total size a multiple of 16 bytes. Bytes from the previous block are used as padding, to reduce the predictability of the bytes in the padded block.

Each block is then encrypted with the AES cipher algorithm operating in Cipher Block Chaining mode. In this mode, the ciphertext of the previous block is bitwise XORed with the plaintext of the current block before the block is encrypted. The first plaintext block of each packet is XORed with an Initialization Vector - a special block of data which is shared between the two ends of the IPsec connection, as part of the Security Association. A complete discussion of CBC mode can be found in Section C.4.6 on page C-15.

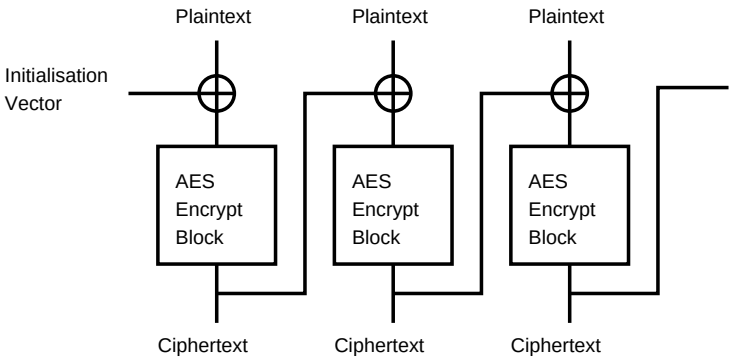


Figure 4.6: Cipher Block Chaining Encryption

4.4.4 Encapsulation of Packets

The encapsulation process requires the construction of an ESP header and footer, along with an outer IP header for encapsulation of the enciphered packet data. Figure 4.7 illustrates the packet structure that is formed with the constructed headers and footers.

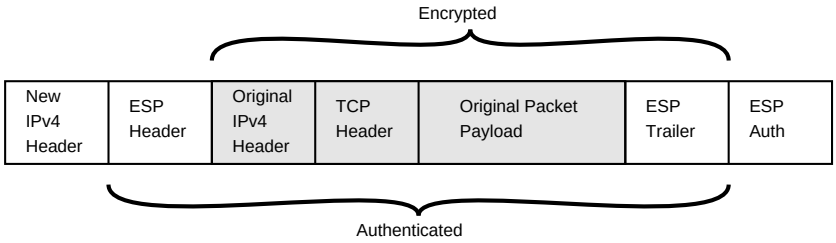


Figure 4.7: ESP Tunnel Mode (from [20])

Construction of the outer IP header is simple. The majority of the fields are filled in with constant values, or values derived from the Security Association.

Header Field	**Value**
Version	Set to 4 (only IPv4 is supported).
Header Length	Set to 5. The header length is 20 bytes as no IP options are supported.
TOS	Copied from the header of the inner packet.
Total Length	Calculated from the size of the encrypted packet, ESP header and IP header.
ID	Set to 0.
Flags	Copied from the header of the inner packet.
Fragment Offset	Set to 0. Fragmentation is not supported.
TTL	Set to the TTL of the inner packet header minus one.
Protocol	Set to 50 to indicate the ESP protocol.
Checksum	Calculated from the constructed packet.
Src Address	Set to the address of the gateway.
Dest Address	Copied from the header of the inner packet.

Construction of the ESP header requires very little computation. In a complete IPsec implementation, the Security Parameter Index (SPI) would be found from the Security Association Database and filled into the ESP header. In the minimal implementation we present this value is a constant - only one Security Association is supported. The Sequence Number field is

filled with the sequence number of the packet - a monotonically increasing variable which tracks the number of packets sent with the current security association.

The encrypted data is placed immediately after the ESP header, followed by the ESP footer. The ESP footer is constructed to indicate that the inner packet is IPv4 and the number of bytes of padding that were required.

4.4.5 Hashing Operation

Following the encryption process, groups of AES blocks repacked with padding as specified in [21] to create 64 byte SHA-1 blocks. The HMAC-SHA-1 (see Section C.3.2 on page C-8) algorithm is then used on these larger packets, along with the session key, to produce a 160 bit (20 byte) Message Authentication Code for the packet ciphertext. The constructed ESP header and footer are also packed into blocks and added to the data to be hashed.

The 20 byte message authentication code produced by HMAC-SHA-1 is truncated to 96 bits (12 bytes) to produce the HMAC-SHA1-96 message authentication code required by the ESP protocol. Following it's calculation, the MAC is copied to the end of the packet, after the ESP trailer.

4.5 Processing of Received IPsec Packets

The sections above discuss the method used to transform an IPv4 packet into a tunnel mode IPsec packet using the ESP protocol. The design of the process to transform an IPsec tunnel mode ESP packet into an IPv4 packet (decapsulation) is very similar and requires the same operations.

In the decapsulation case, packet reception and transmission operations are unchanged. The encryption operation is replaced with an AES-CBC decryption operation which is extremely similar to the decryption operation. The AES decryption operation uses different tables and a slightly modified key schedule, but requires the same number of operations as the encryption algorithm.

The hashing operation is performed identically in the decryption case — the entire 160 byte MAC is calculated, truncated to 12 bytes and compared with the MAC attached to the received packet. If the calculated and received MACs do not match, the packet is discarded. If the calculated and received MACs match, the decrypted inner IP packet is copied from the received packet and is transmitted over the network.

The decryption and decapsulation operations are very similar to the encryption and encapsulation operations and require the same amount of mem-

ory and CPU time. The design presented above for encapsulation and encryption would require very little modification to handle the decryption and decapsulation of IPsec packets.

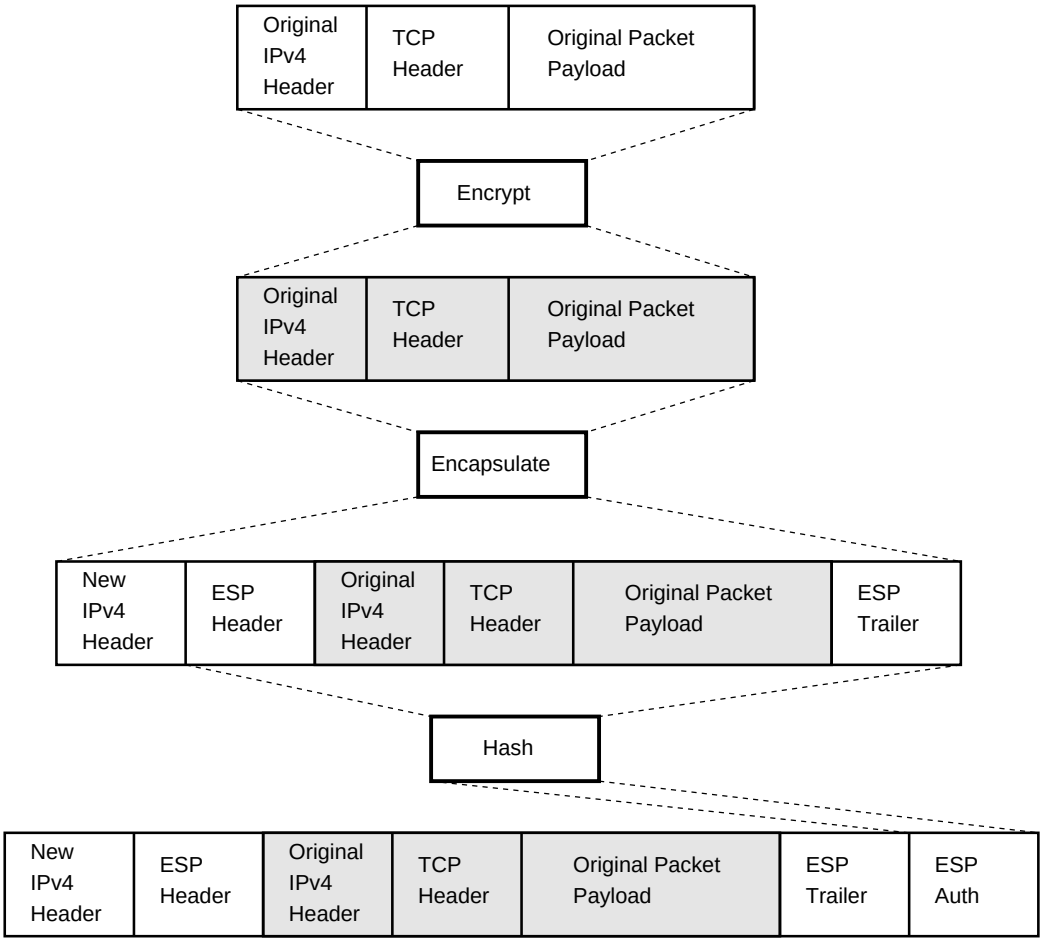


Figure 4.8: Overview of Packet Processing Operations

Chapter 5

Analysis of Software Design

5.1 Simulation Environment

In order to analyse the performance of the design detailed in Chapter 4, the system was implemented in Microengine C, a dialect of the C programming language supported by the Intel Developer Workbench development environment and compiler. The implementation is detailed in Appendix D.

Intel Developer Workbench provides a cycle-accurate[22] software simulator for the IXP2400 network processor which does not depend on the presence of the IXP2400 hardware. This simulator was used to analyse the performance of the design and i implementation. Use of a software simulator decreased the amount of time required for debugging of the implementation and setting up of simulations. Simulations were performed overnight and over weekends to gather a significant amount of data for processing.

Analysis of the design and implementation was performed by running the simulator with a variety of different packet streams, collecting the results of the simulation and analysing these results with MATLAB and Python programs.

5.1.1 Packet Streams

Two methods were used to generate streams of packet data for accurate simulation of a real-world deployment environment.

The first method used was packet generation DLL (external library), developed in C++ using Microsoft Visual Studio, which was plugged into the simulation environment. This external library used a statistical model to generate a packet stream with exponentially distributed interarrival times and packet sizes distributed according to the values typical of Internet traffic. The packet generation DLL can be tailored to generate streams at any

speed that is required.

The second method used for generating packet streams was the *Datastreams* packet generator built into the Developer Workbench environment. Using this tool, packets can be generated in many formats, including Ethernet TCP/IP, Ethernet IP and PPP TCP/IP. The Datastreams tool can generate packets with random sizes distributed linearly over an arbitrary range.

While the first method offered superior flexibility and configurability, the simulator performance penalty it caused was too large to allow large simulations to be run effectively. The majority of the simulations of the final design were run with packets generated by the built in *Datastreams* tool.

5.1.2 Distribution of Packet Sizes on the Internet

The distribution of packet sizes on the internet has two strong modes. The first mode is between 40 and 64 bytes per packet - made up by control packets, ACK packets and data packets for interactive applications such as Telnet, SSH and Voice over IP. The second mode is around 1500 bytes per packet, the maximum transmission unit (MTU) of Ethernet. Applications which transfer bulk data, such as HTTP and FTP generate the majority of large packets on the Internet.

The performance of the IPsec gateway will depend on the distribution of packets sent through it. This makes the packet size distribution of the traffic stream to be sent through the gateway critical to accurate evaluation of the performance of the design.

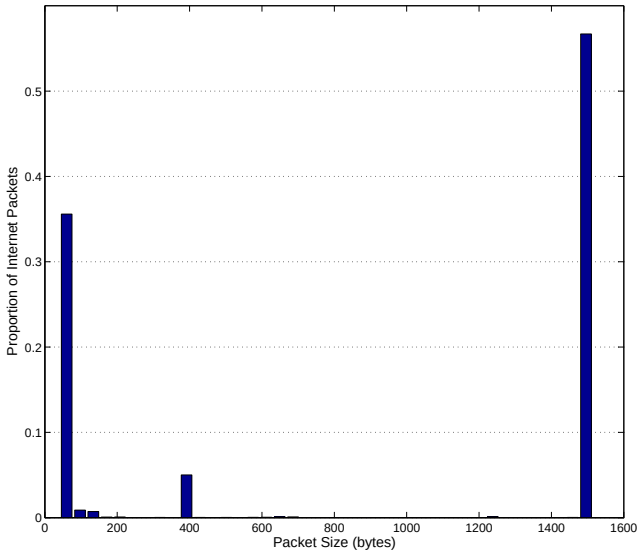
Measurement of Packet Sizes

The distribution of packet sizes in typical Internet traffic was found by collecting packet transfer statistics for a number of machines on the University of Cape Town campus. Statistics were collected with the `tcpdump`[1] program, an open source packet capture program which allows packet headers for all traffic flows to and from a machine to be captured.

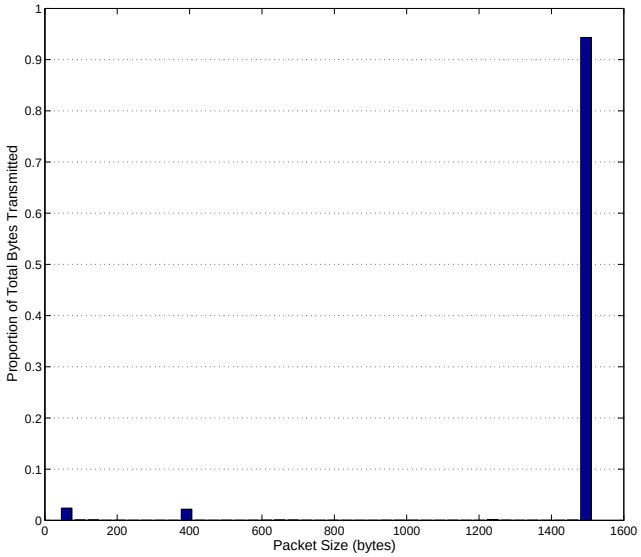
During the course of the analysis, 5.2 million packet headers were collected. A Python program was then used to extract the total packet size fields from these packets and write them into a text file. MATLAB was then used to analyse the packet distribution data and find the size distribution of this data.

The distribution of packet sizes obtained from this analysis is illustrated in Figure 5.1(a). While small packets make up a significant proportion of the

[1] `tcpdump` can be downloaded from *http://www.tcpdump.org*



(a) Distribution of Internet Packet Sizes



(b) Distribution of Packet Sized Weighted By Size

Figure 5.1: Results of Packet Size Investigation

total packet count (about 35% of total traffic), the total amount of data contributed by small packets is realtively insignificant. Figure 5.1(b) illustrates the total proportional contribution to internet traffic of packets by size.

The amount of time required to encrypt, hash and encapsulate an IP packet depends is proportional to the packet size, plus a small constant value. The proportional load caused by packets of different sizes on the IPsec gateway is difficult to calculate.

From the analysis, it was found that the mean packet size is approximately 903 bytes.

In order to validate these results, measurements of Internet packet size distributions taken elsewhere were compared to them. While distributions from all sources show the same strong bimodal structure, the exact distributions differ widely. It is intuitively obvious that packet size distributions from different sources would be different - they are entirely dependent on the usage of the network being measured.

Figure 5.2 illustrates the distribution of packets on an unknown tier one carrier, from Hank Nussbacher, based on 46×10^9 packets[2].

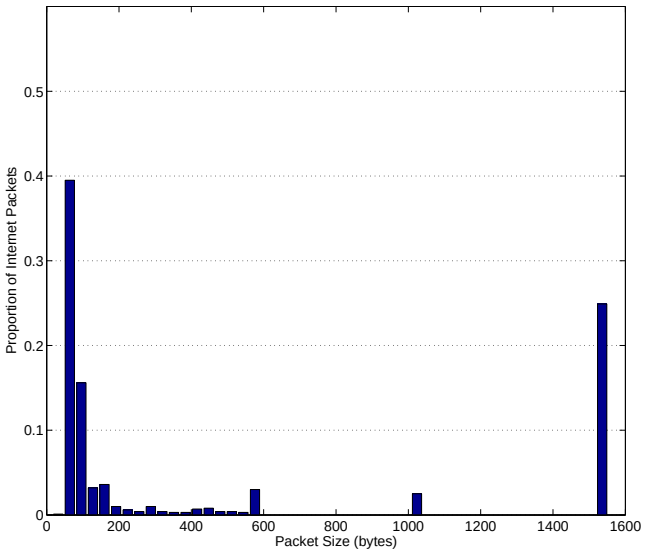


Figure 5.2: Distribution of Packet Sizes Weighted by Size

Nussbacher's data suggests an average packet size of approximately 500

[2]From a posting to the North American Network Operators Group mailing list by Hank Nussbacher, December 2003. *http://merit.edu/mail.archives/nanog/2003-12/msg00399.html*

bytes. While the distributions differ substantially, they are very similar in terms of overall form - a strong peak at between 32 and 64 bytes and a strong peak at around the Ethernet MTU of 1500 bytes. The ratio between the peaks and the number of packets of intermediate sizes depends on the applications used on the network being measured.

Distribution Used for Testing

Based on the measurements presented above, a simple packet size distribution was developed for testing of the IPsec gateway. This distribution simplified the distributions above to 47.5% packets linearly distributed between 32 and 96 bytes, 47.5% packets between 1400 and 1600 bytes and 5% packets between 450 and 550 bytes. The mean packet size of the distribution used was approximately 770 bytes.

5.1.3 Simulation Methodology

The system was tested with a variety of different packet loads on different speed simulated networks. For each simulation, the simulator was run until it was clear that the simulation had settled down to a constant behaviour.

The output of the packet simulator consists of two files - a file containing the packets received by the program and a file containing the packets transmitted from the program. Along with the packet data, the files contain timestamp information which indicates the times the packets were sent or received. A set of Python scripts was used to process the packet logs and extract data about the packets which were processes. The collected data was then analysed and graphed with MATLAB .

5.2 Voice Over IP

Voice Over IP protocols such as SIP, H.323 and Skype produce large numbers of small data packets containing compressed voice data. Interactive protocols and applications are extremely sensitive to packet delay - long delays will make conversation over VoIP networks uncomfortable and difficult.

In order to simulate Voice over IP traffic, a stream of packets was generated with sizes distributed uniformly between 75 bytes and 150 bytes. This packet distribution is similar to that generated by common voice over IP protocols. The widely used IP telephone application Skype, for example, generates UDP packets with a 67 byte payload or a total size of 95 bytes (67 bytes payload, 20 bytes IP header, 8 bytes UDP header)[23].

This simulation was performed with a simulated 1000Mbit per second Ethernet network on the receive and transmit side and an input packet flow which matched the demand of the processor.

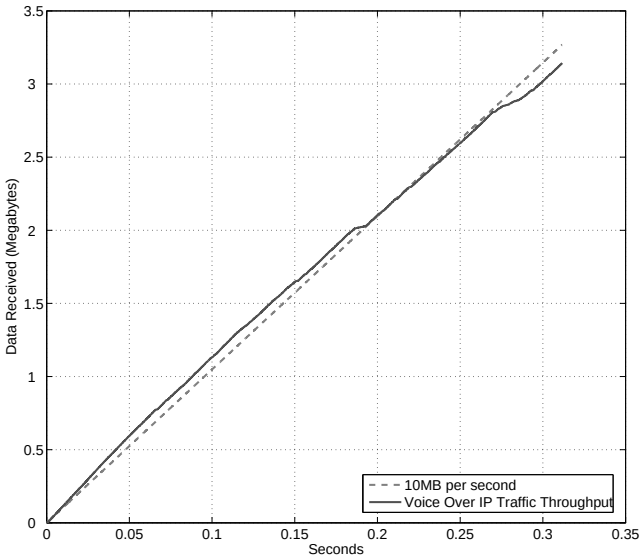


Figure 5.3: Transfer Rate for VoIP Packet Load

Voice Over IP Throughput

The Voice Over IP simulation was run for a simulated time of 310ms, during which time 27100 packets were processed, totalling 3.14MB. This corresponds to a total throughput of approximately 10 MB per second, or 80 $Mbit$ per second.

The Skype VoIP protocol uses between 3 and 16 kilobytes per second of bandwidth[23], depending on network availability and packet loss. Assuming a worst case usage of 16 KB per second per call, the throughput of the gateway will support approximately 615 simultaneous voice connections. The total data transferred by the processor while processing the VoIP packet load is presented in Figure 5.3, along with a reference line at 10 MB per second (80Mbit/s).

Voice Over IP Packet Delay

The average delay per packet introduced by the IPsec gateway was found to be 0.25 ms. As packets will pass through two IPsec gateways during their

transfer, the average packet delay caused by transmission over a VPN is approximately 0.5 *ms*. This delay is acceptable for interactive applications such as Voice over IP, SSH, multiplayer games and streaming video.

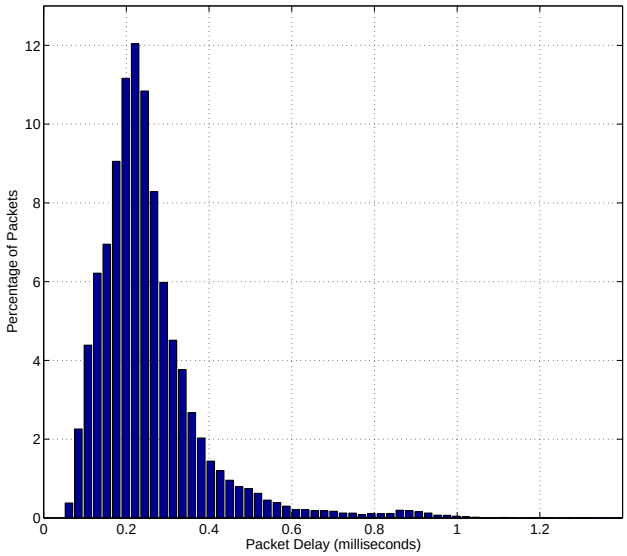


Figure 5.4: Per-Packet Delay for VoIP Packet Load

A histogram of packet delays, corrected for packet size, is presented in Figure 5.4. Ninety percent of packets have a delay greater than 0.12 *ms* and a delay less than 0.38 *ms*. The small ratio between delay and variation in delay (*jitter*) will cause small packets to be transmitted in a different order from the order in which they arrived. Modifications to the design could be made to prevent packets from being transmitted out-of-order.

The measured mean packet jitter for the VoIP simulation was 0.25 *ms* and maximum recorded jitter was 0.90 *ms* (using the method described in "RTP: A Transport For Real-Time Applications (RFC3550)"[24] by Schulzrinne et al.), a significant but acceptable amount of jitter.

5.3 Large Packet Simulation

Large packets make up the majority of the total number of bytes transferred over the Internet (see Figure 5.1(b) on page 38) and will make up the majority of processing time used by an IPsec gateway deployed on a general purpose network. Large packets are generated by applications and protocols including

HTTP (web browsing), FTP (bulk file transfer), peer-to-peer file sharing and SMTP. These applications transfer large amounts of data and are not sensitive to small (less than 500 ms) delays.

In order to test the IPsec gateway's performance with extremely large packets, a traffic stream was generated with random packet sizes distributed uniformly from 1500 to 2000 bytes. Using the simulation framework presented above, throughput and delay measurements were made for large packets.

Large Packet Throughput

The large packet simulation was run for a simulated time of 233 ms, during which time 2172 packets were received and processed, totalling 3.8 MB. The mean throughput (on the receive side) of the IPsec gateway was approximately 16 MB per second (128 $Mbit$ per second).

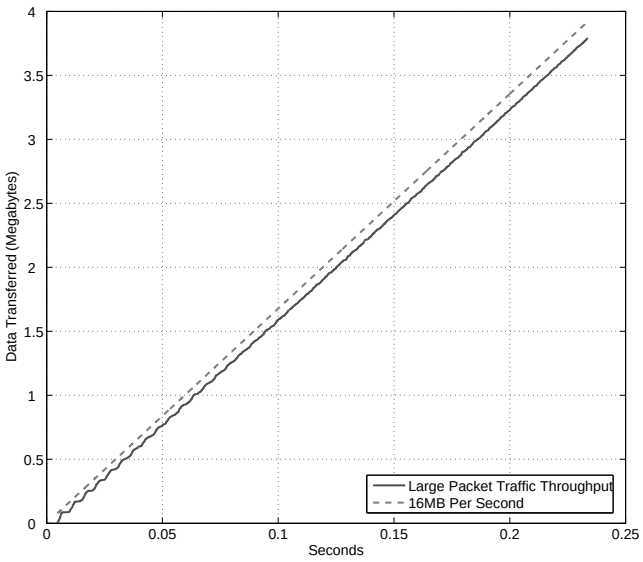


Figure 5.5: Transfer Rate for Large Packet Load

At 16 MB per second, the transfer of a 650 MB CD image would take 41 seconds and a 4.7 GB DVD image would take 300 seconds. This transfer rate is superior to the transfer rate of 100 $MBit$ per second Ethernet networks, which are currently widely deployed.

The total data received by the processor while processing the large packet load is graphed in Figure 5.5, with a reference line at 16 MB per second.

Note that despite early variations, the graph is extremely linear, suggesting that the behaviour of the system is constant.

Large Packet Delay

The mean delay per packet introduced by the IPsec gateway was found to be 1.9 *ms*, or 3.8 *ms* for a pair of gateways. This delay is acceptable for bulk data transfer applications and will not have a noticeable effect on application performance.

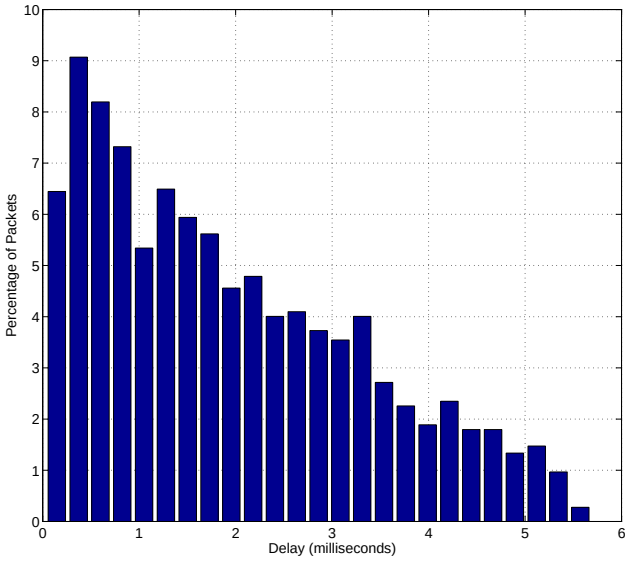


Figure 5.6: Per-Packet Delay for Large Packet Load

A histogram of delay introduced by the gateway is presented in Figure 5.6. Ninety percent of packets are delayed by between 0.37 *ms* and 4 *ms*. As in the VoIP simulation, large variations in delay are likely to cause out-of-order packet transmission, which could cause problems with some transport layer and application layer protocols.

5.4 Typical Internet Simulation

The third simulation was run with a packet mixture typical of real-world internet connections, as described in Section 5.1.2 on page 37. This simulation, unlike the Large Packet and VoIP simulations, is likely to be a good indication of real-world system performance.

As in the previous two simulations, the receive and transmit networks were limited to 1000 $Mbit$ per second and the speed of the packet stream was dynamically adjusted to keep the receive buffer full at all times or saturate the receive network.

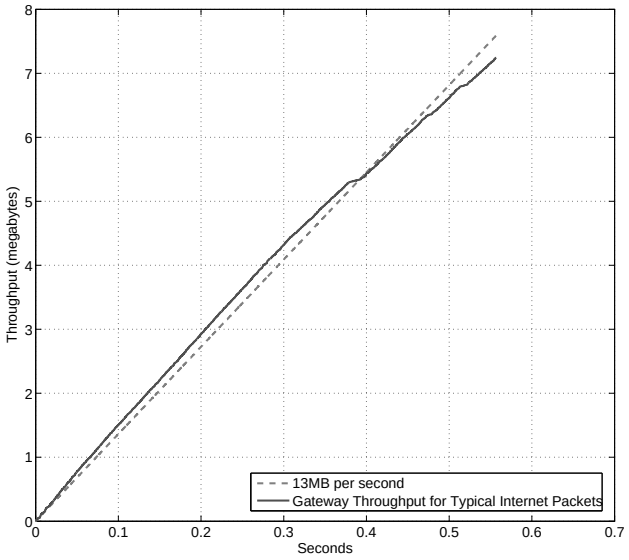


Figure 5.7: Transfer Rate for Typical Internet Packet Load

Typical Packet Throughput

The Typical packet simulation was was run for a simulated time of 557 ms. 7.24 MB of data was received in 8134 packets at a mean throughput of 13 MB per second (104 $Mbit$ per second).

This throughput is 23% less than the throughput recorded in the large packet simulation, but 20% greater than the throughput recorded in the VoIP simulation. Despite the reduction in throughput when compared to the Large Packet simulation, the transfer rate remains greater than 100 $Mbit$ per second.

Typical Packet Delay

The mean delay per packet for the internet packet load was 0.65 ms, or 1.3 ms for a pair of IPsec gateways. Figure 5.8 is the packet delay histogram for the simulation using a typical internet packet size distribution.

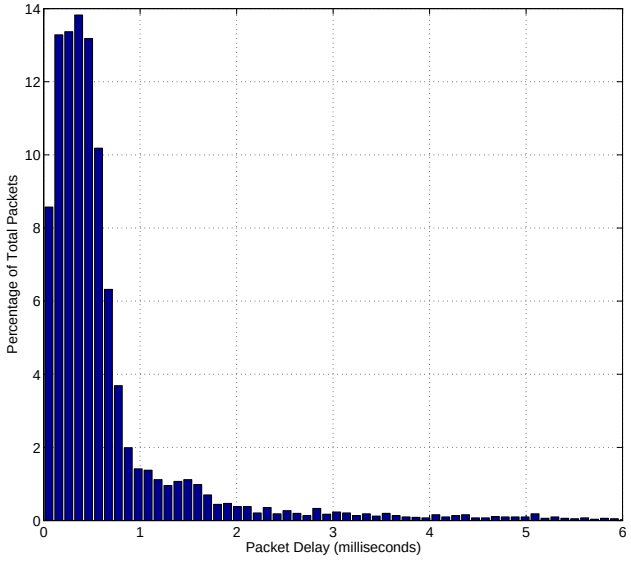


Figure 5.8: Per-Packet Delay for Typical Internet Packet Load

Ninety percent of packets experienced a delay of between 0.15 ms and 1.3 ms, while two thirds of packets experienced a delay of between 0.16 ms and 0.87 ms. As with the previous two simulations, the variation in packet delays could cause out-of-order delivery. High *jitter*, as generated by this gateway, can cause quality of service degradation in some interactive protocols. The calculated mean jitter for this simulation was 0.65 ms (using the method from [24]), a significant but not unacceptable amount of jitter. The variation in jitter with time is plotted in Figure 5.9, including a moving average with a window length of 300 samples.

5.5 Limited Network Speed Simulation

A fourth simulation was run with the the simulated network speed closer to the throughput of the gateway. The goal of this simulation was to analyse the effect that network speed had on the throughput and delay behaviour of the system.

In the previous three simulations, the simulated network was limited in speed to 1000 $Mbit$ per second, approximately ten times greater than the maximum throughput of the system. Two simulations were performed with network speeds closer to the transfer rate of the system. The first simulation

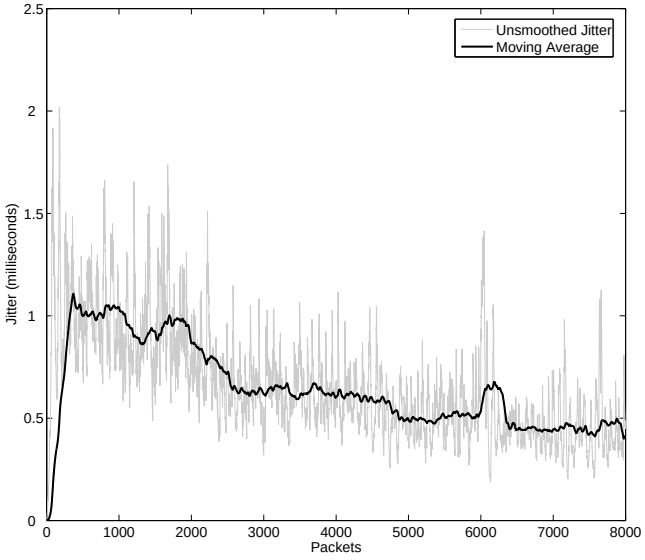


Figure 5.9: Per-Packet Jitter for Typical Internet Packet Load

was performed with 100 $Mbit$ per second limitations on both the ingress and egress networks. The second simulation was performed with a 80 $Mbit$ per second limit on the ingress network and a 100 $Mbit$ per second limit on the egress network.

Throughput

Both limited network speed simulations were run for a simulated time of 228 ms. During this time the first (100-100) simulation processed and transferred 1444 packets totalling 1.35 MB at a mean throughput rate of 5.9 MB per second (47 $Mbit$ per second). The second (80-100) simulation processed and transferred 1081 packets with a total size of 0.98 MB at a mean throughput rate of 4.3 MB per second (34 $Mbit$ per second).

Delay

The mean delay for both simulations was approximately 0.89 ms, or 1.78 ms for two gateways. The distribution of delays was very similar to that found in the typical internet packet simulation, with a mean increase of approximately 37%.

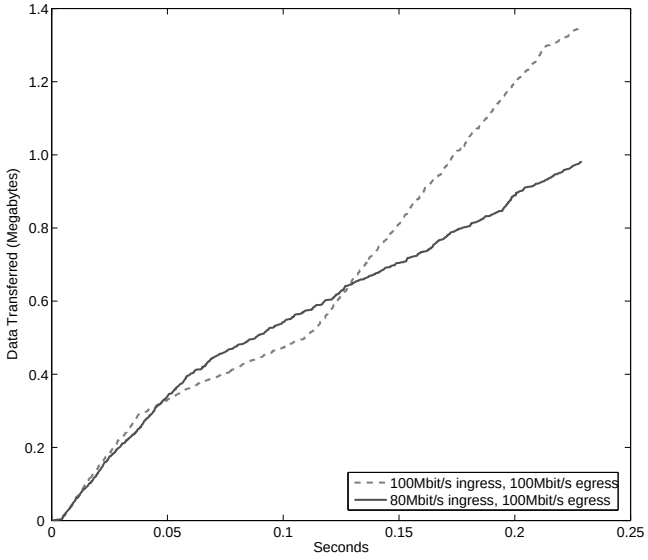


Figure 5.10: Transfer Rate for Limited Network Speed Simulation

5.6 Comparison With Other IPsec Implementations

Performance figures are not available for the majority of commercial IPsec implementations. Benchmarks have, however been carried out on a variety of open source implementations of IPsec. Linux Journal[3] measured the performance of the native IPsec implementation in the Linux 2.6 kernel. The maximum throughput was found to be 42 *Mbit* per second between two Pentium 4 2.2GHz machines. The FreeSWAN IPsec implementation for Linux claims 36 *Mbit* per second for a 733MHz Pentium III[4].

Dedicated embedded IPsec gateways offer excellent performance. Nokia claims a maximum throughput of 120 *Mbit* per second for it's 50i VPN gateway and 600 *Mbit* per second for it's 100i product. Cisco's ASA 5500 line ranges from 170 *Mbit* per second for the ASA5510 to 325 *Mbit* per second for the ASA5540. While both the ASA5540 and Nokia 100i offer functionality apart from IPsec processing, they are extremely expensive - the ASA5540 costs approximately US$18000 while the 100i retails at approximately US$10000.

[3]http://new.linuxjournal.com/article/7840

[4]http://www.freeswan.org/freeswan_trees/freeswan-1.99/doc/performance.html

While it is not necessarily valid to compare the performance of IPsec solutions without third party testing in a controlled environment on a standard packet load, the transfer rates presented here illustrate the difference in performance between software VPN implementations and dedicated hardware solutions.

5.7 Discussion of Results

The throughput of the gateway design was found to be dependent on packet size. This is unsurprising - the amount of time required to encrypt and hash a packet increases linearly with the size of the packet. The constant time required for packet encapsulation was found to cause system throughput on large packets to be superior to system throughput on loads dominated by small packets. Packet delay was considerably less on small packets. This was the expected result - smaller packets contain less data to process and hence processing time causes less per-packet delay. Per-packet delay for a given packet size was found to be approximated by the equation $delay = \frac{size}{1000} + 0.13$.

Per-packet delay results for all three simulations were found to be within acceptable bounds. The ITU-T SG9 recommends an end-to-end delay of 50 ms or less for high quality video streaming and 100 ms for CD-quality audio. The delay introduced by this gateway will not have a significant effect on end-to-end delay.

The significant performance degradation caused by limitation of media speeds is somewhat unexpected. The limitation of ingress speed appears to have prevented the system from keeping the processing threads busy, wasting processing resources and reducing throughput. The limitation on egress speed appears to have increased the amount of time required to transmit packets. This increase has lead to an increase on the amount of time processing threads waste waiting for other threads to unlock the packet transmission mutex. The implementation of a buffer between the processing units and the transmission unit would have been likely to decrease the performance effect of limited egress network speed. More simulations are required to fully understand the relationship between network speed and system performance.

The large variation in packet delay is a cause for concern - some protocols require low jitter and in-order packet delivery in order to function optimally. While the jitter observed in the simulations (0.65 ms for the typical simulation) is below the ITU-T recommended maximum for high-quality VoIP applications (5 ms), the gateway presented here will make a significant contribution to end-to-end jitter.

5.8 Conclusion

The reduction in Quality of Service caused by implementation of the VoIP gateway analysed above in an internet connection is likely to be acceptable for most applications. The throughput limitation imposed by the device places restrictions on bulk-data transfers over high speed networks, but is not a significant restriction for VoIP and other interactive traffic. Delay and jitter were found to be within acceptable bounds for both interactive and bulk-data transfer applications.

Improvements in the design of the system could be made to reduce jitter and prevent out-of-order packet delivery. The addition of a large transmit buffer to the design is also likely to reduce the effect of network speed limitation on throughput.

Chapter 6

Conclusion

6.1 Achievement of Project Goals

The aim of the project was to investigate, design and test the software for an implementation of a Virtual Private Networking gateway based on the IXP2400 Network Processor.

Following an analysis of current Virtual Private Networking protocols (Appendix B), the Internet Protocol Security protocol was chosen for implementation. IPsec is a large and complex protocol family and there was not sufficient time for the implementation of the entire protocol to be designed, programmed and tested. A subset of the IPsec protocol family - Encapsulating Security Payload operating in tunnel mode - was chosen for implementation as the operations performed by this protocol are the most performance critical in an IPsec implementation.

The design of a set of programs for the Intel IXP2400 Network Processor's Microengines was completed. The final design was chosen from the analysis of four candidate designs, which were described in detail. Following the selecting of the overall software designs, designs were presented for each of the constituent programs of the IPsec implementation. Two candidates for the design of the most critical component, the packet processor, were presented and analysed.

The final design was implemented in Microengine C, a dialect of the C programming language native to the Microengines of the IXP2400. An extensive performance analysis of the completed gateway was performed using a simulation environment.

6.2 Conclusions from Design and Testing

The multi-cored nature of the IXP2400 processor requires that applications be written with multiple processes, one per Microengine core. In order to take full advantage of the capabilities of each microengine, processes which perform extensive memory access need to be written with multiple threads.

Designing applications for this multi-process multi-threaded environment requires a different approach to designing for traditional uniprocessor systems. Design techniques need to be conscious of the requirement for multiple independent threads and aim to limit thread synchronisation. Despite the unusual nature of the hardware, traditional software design approaches, such as the finite state machine model, were found to be extremely useful in the design of individual threads.

Designing efficient applications for multi-cored network processors is extremely difficult. Simulation of trial designs and manual design space exploration are powerful tools that allow the designer to understand how design parameters affect system performance. Automated design space exploration is also likely to prove to be a useful tool for network processor software design.

The final system, an implementation of the Encapsulating Security Payload protocol, was found to have a maximum throughput of 104 $Mbit$ per second. Parameters relevant to Quality of Service, such as packet delay and jitter, were found to be within acceptable limits. The IXP2400 network processor is well suited to the implementation of IPsec.

6.3 Future Work

The scope for future work and extensions to the work presented above is extremely large. Pairing the design presented above with an implementation of the IKE protocol on the IXP2400's XScale control processor and adding support for Security Associations to the processing program would make the system a working IPsec implementation.

Beyond completion of the gateway, several other possibilities for future work exist.

- Perform profiling and hotspot optimisation of the packet processing code in order to reduce per-byte processing time.
- Investigate integration with a hardware AES implementation for improved processing speed.
- Implement the full IKE and ISAKMP protocols for Security Association establishment and maintenance.

- Implement the Authentication Header protocol and tunnel mode for AH and ESP.
- Perform interoperability tests with a reference IPsec implementation.
- Perform an automated exploration of the software design space for IPsec implementations.

Appendices

Appendix A

The Intel IXP2400 Network Processor

A.1 Overview

Along with the rise in broadband packet networks came the demand for processors which could process data flowing over these networks at full network speed. The first products designed to perform data processing tasks at line speed relied on specialised hardware (ASICs) which offered excellent performance at the cost of flexibility. ASIC based designs could not be modified if requirements changed and had to be replaced instead of being extended to perform new tasks. The reprogrammability of FPGAs made them ideal for performing some tasks, but throughput and memory limitations, along with high price, restricted their suitability for more complex network processing tasks.

In response to the growing requirements of packet processing and demand for more flexible hardware, network processors were developed. Network processors are programmable processors which can perform data processing tasks at line speeds. In most network processor designs, performance demands were met by including many processing engines on a single processor. Each processing engine is a very restricted RISC processor optimised for performing network processing tasks, including bit manipulations and checksums. Several processing engines are combined with other specialised hardware including medium access controllers (MACs), switch fabrics and high speed memory to make a complete network processor.

The Intel IXP2400 Network Processor, a second generation network processor, consists of eight Microengine v2 data processing engines, an Intel XScale control processor, SRAM and DRAM controllers, a medium interface

and a PCI controller.

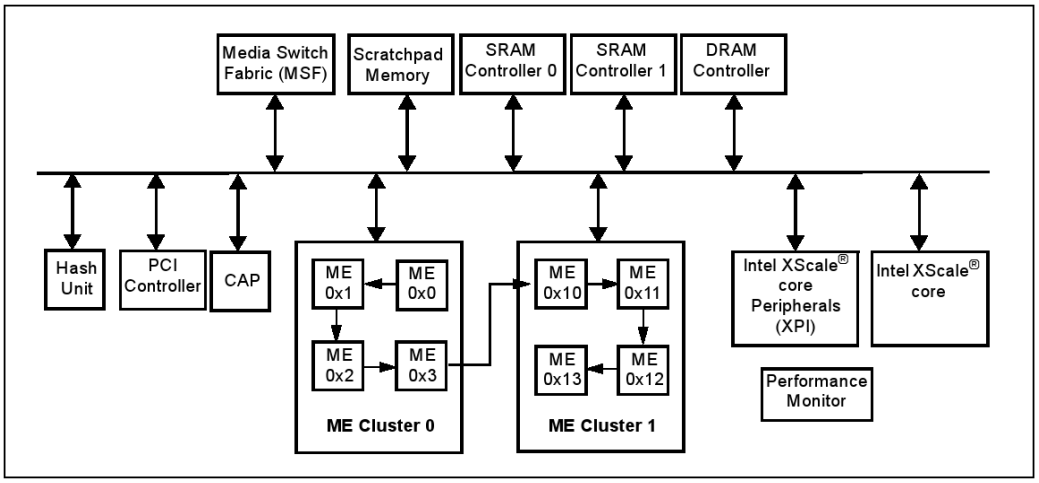


Figure A.1: Block Diagram of IXP2400 architecture (from [25])

A.2 Architecture of the IXP2400

A.2.1 Microengines

The IXP2400 network processor includes eight Microengine v2 network processing engines. These engines are very simple programmable RISC processors which perform the majority of data plane processing tasks on the IXP2400. The features of these microengines include:

- 256 General Purpose Registers
- 512 Transfer Registers (used for memory access)
- 128 Next Neighbour registers (used for transferring data to other microengines on the same processor)
- 16 entry Content Addressable Memory
- Zero overhead context switching for up to eight threads
- 4096 instruction control store
- Hardware signal handling of up to fifteen signals

Each microengine has access to all the other resources on the network processor, including the memory controllers and the media and switch fabric interface (MSF).

Contexts

Microengines have support for context switching, with zero effective overhead, between up to eight threads. Use of multiple threads decouples data processing times with comparatively long memory access times, allowing another thread to continue when one is waiting for data from the memory controllers. Context switching is handled by a non-preemptive round-robin hardware thread controller, implemented in hardware on each microengine. Each of the eight hardware contexts (threads) on each microengine can be

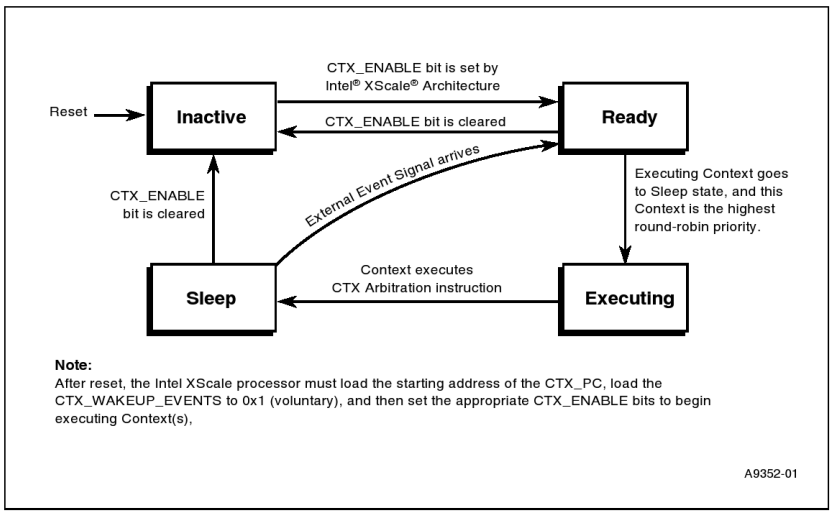


Figure A.2: Microengine Context State Transition Diagram (from [25])

in one of four states, *Inactive*, *Executing*, *Ready* and *Sleeping*. Figure A.2.1, from the IXP2400 Hardware Reference Manual, illustrates the possible transitions between these states.

Inactive The context is not required by the application and was disabled at compile time.

Executing The context is currently executing instructions from the control store. Executing contexts will continue to execute until they fetch an instruction which would cause them to block (such as a memory access). Only one context per Microengine can be the *executing* stage at a time.

Ready The context is ready to execute. When the current *executing* process switches to the sleep state, the hardware selects which of the *ready* contexts to run with a round robin arbitrator.

Sleep The context is waiting for an external event, such as the completion of a memory access operation.

Memory

Microengines have access to four different types of memory. Each of the memory types represents a tradeoff between expense, size, access latency and bandwidth.

Local On-die memory local to each microengine. It is extremely fast but very limited in size.

Scratch A small, fast memory shared by the eight microengines on the IXP2400. Scratch is large enough to contain small data structures and single packets.

SRAM Static Random Access Memory is medium speed and fairly large (depending on the system configuration).

DRAM Dynamic Random Access Memory is relatively slow but is likely to be the largest memory available due to it's low cost.

Table A.1: IXP2400 Types and Properties of Memory (from [12])

Memory Type	Size	Access Latency	Bus Width
Local	2560 bytes	3	4
Scratch	16K	60	4
SRAM	256M max.	150	4
DRAM	2G max.	300	8

Table A.1 lists the maximum sizes, bus widths and access latencies of the four memory types. Local memory on the IXP2400 microengines is similar in performance to L1 cache on modern PC CPUs. The L1 cache on the AMD Athlon family has an access latency of 3 clock cycles[1]. Local memory should not be confused with a cache, however. Data must be assigned to local memory manually and it does not offer the same features, such as set associativity, that a true CPU cache does. Local memory is suitable for small data structures which are accessed often but are not suitable for allocation to General Purpose Registers.

Scratch memory on the IXP2400 is shared between all eight microengines and the XScale core. It offers fairly low latency, but is severely limited in size.

[1]From "Calibrated Hardware Database" http://monetdb.cwi.nl/Calibrator/

Scratch is useful for shared memory communication between microengines and for storage of relatively small data structures. The IXP2400 also supports atomic access to scratch memory, using one of 16 possible rings. These rings make implementation of producer/consumer relationships between contexts or microengines simple and fast.

Static RAM is fairly expensive and thus it is unlikely to be used in large amounts in deployed systems. The IXP2400 supports up to 256MB of static RAM which can be used for medium speed access to large data structures and packet data.

Double Data Rate (DDR) Dynamic RAM, as used by the IXP2400, is cheap compared to other RAM types and is therefore likely to be present in large amounts. DRAM access is a fairly high latency operation (300 cycles if the controller is unloaded) which, combined with the lack of caches in the microengines, makes accessing data from DRAM slow compared to other types of RAM. In comparison, DRAM access latency on an AMD Athlon PC processor is 160ns (or 160 cycles on a 1GHz processor), which is effectively reduced (by as much as 80%) in practice by the use of a multi-level cache heirarchy[11]. In order to achieve good performance, techniques such as explicit caching and write/read combining must be used to reduce the effect of this high latency.

One of the largest challenges for the programmer of the IXP2400 network processor is effectively using these types of memory to achieve acceptable performance while keeping costs low.

Content Addressable Memory

The IXP2400 microengines each contain a 16 entry Content Addressable Memory. CAM is equivalent to a hardware implementation of an associative array, or dictionary. The CAM allows the microengine to perform an extremely fast lookup from a tag to an associated state. Content Addressable Memory hardware is extremely useful for many network processing applications, such as routing.

A.2.2 XScale Control Processor

The Intel XScale processor used as a control processor in the IXP2400 is a powerful RISC microprocessor, compliant with the ARM V5TE standard. The V5TE standard specifies a fairly limited instruction set without support for floating point operations. Focussing on instructions relevant to packet processing keeps costs down while not sacrificing performance.

This core is designed to run an embedded operating system (such as RTLinux) and oversee the operations of the microengines. It's responsibilities include start-time setup of the microengines and exception handling for most of the hardware devices. For example, an ECC error in the DRAM controller will send an interrupt to the XScale processor which will take responsibility for ensuring that the microengine which made the failed request continues with it's program successfully.

The XScale processor is also suitable for implementing non-performance critical but fairly complex protocols. The XScale core would, for example, be perfect for implementing the IKE and ISAKMP protocols in an IPsec implementation.

A.2.3 Media and Switch Fabric Controller

The IXP2400 communicates with the network (via a separate physical layer controller) or with other network processors on the same board through the Media and Switch Fabric Interface (MSF). The MSF supports a pair of independent 32bit wide buses with clock speeds of between 25MHz and 133MHz - one receive bus and one transmit bus. Each of theses buses can be configured independently, using different protocols and bus speeds, if necessary.

The MSF buses support:

- Configuration as four 8bit buses, two 16bit buses or one 32bit bus.
- POS-PHY (Packet over SONET) Levels 2 and 3.
  - Multi PHY master mode operation, with support for up to 32 slave ports.
  - Single PHY master mode operation.
- CSIX-L1 (Common Switch Interface) with a 32bit wide bus.
- UTOPIA (Universal Test and Operation Physical Interface for ATM) Levels 1, 2 and 3.
- CBus connection for connecting to other IXP series processors.

A.3 Conclusion

The IXP2400 is an extremely powerful and capable processor. It offers many features extremely useful for packet processing, along with a fast and efficient interface to network hardware.

The multi-cored nature of the IXP2400 is perfectly suited to exploiting the data level parallelism inherent in packet streams. Multiple packets can be processed in parallel, reducing delay and increasing system throughput.

Appendix B

Virtual Private Networking Protocols

B.1 Internet Protocol Security (IPsec)

B.1.1 Description of Protocol

IPsec[20] is a cryptographic-based protocol which provides authentication, encryption, replay protection and integrity to IP packets. In addition to these functions, IPsec also provides limited flow confidentiality. IPsec is an obligatory part of the IPv6[1] standard and an optional part of the IPv4 standard. This protocol provides its functions at the IP layer (layer 3) and provides security to all protocols which use IP as a network layer protocol.

Security services are provided by two security protocols: Encapsulating Security Payload (ESP) and Authentication Header (AH). AH[27] provides integrity and authenticity protection (but not confidentiality) for an entire IP packet, including the header. ESP[16] provides integrity, authenticity and confidentiality protection for the payload of an IP packet (not including the packet header). Each of these protocols can be used in one of two modes: transport mode or tunnel mode. Transport mode provides host-to-host security for upper layer protocols. Tunnel mode provides gateway-to-gateway (or portal-to-portal) security for IP packets.

The IPsec protocol suite also includes the IKE (Internet Key Exchange) [28] and ISAKMP (Internet Security Association and Key Management Protocol) [29] protocols. ISAKMP is a standard protocol for the creation and management of Security Associations (SA), authentication of hosts and generation of key material. IKE is a protocol for the exchange of authenticated

[1] Internet Protocol version 6, detailed in RFC2460[26]

key material between hosts and provides key exchange services to ISAKMP. IKE uses public key encryption techniques to provide a secure, authenticated channel for the transfer of key material.

B.1.2 Analysis of Protocol

Ferguson and Schneier

In "A Cryptographic Evaluation of IPsec"[30], Ferguson and Schneier found the IPsec is extremely complex and that its complexity has lead to a number of security weaknesses in the design of the protocol.

> "We have found serious security weaknesses in all major components of IPsec."[30]

The authors provide a number of recommendations for the modifications to IPsec which would improve the security of the protocol by removing redundant parts. One of the most important findings of the paper is that the ESP protocol and tunnel mode provide a superset of the functionality of the AH protocol and transport mode without any major shortcomings.

> "*Recommendation 1* Eliminate transport mode."
> "*Recommendation 2* Eliminate the AH protocol."[30]

Despite serious misgivings about the security of IPsec, the authors seem to believe that IPsec is the best VPN protocol currently available.

> "We are of two minds about IPsec. On the one hand, IPsec is far better than any IP security protocol that has come before: Microsoft PPTP, L2TP, etc. On the other hand, we do not believe that it will ever result in a secure operational system."[30]

> "We strongly discourage the use of IPsec in its current form for protection of any kind of valuable information, and hope that future iterations of the design will be improved. However, we even more strongly discourage any current alternatives, and recommend IPsec when the alternative is an insecure network. Such are the realities of the world."[30]

Bellovin

In "Problem Areas for IP Security Protocols"[31], Steven Bellovin lists a number of cryptographic and inherent weaknesses in IPsec. The attacks

presented range from chosen plain text and known plain text attacks against the cipher implementation to direct attacks against the ciphers themselves. This paper is extremely important as it identifies areas of IPsec which can be implemented in a manner which provides no security, yet complies completely with the published standard[20].

Bellovin notes that most of the attacks presented in the paper can be neutralised by ensuring that encryption is never used without integrity checking and that key material is not re-used between connections.

> "It is quite clear that encryption without integrity checking is all but useless. We strongly recommend that all systems mandate joint use of the two options"[31]

B.2 Point-to-Point Tunneling Protocol (PPTP)

B.2.1 Description of Protocol

The Point-to-Point Tunneling Protocol (PPTP)[32] is a VPN protocol which allows the tunneling of Point-to-Point Protocol (PPP) over an IP Network. PPTP provides a framework for endpoints to negotiate which authentication, encryption and compression algorithms will be used over the tunneled channel. Tunneling is provided at the packet level by encapsulating IP packets in PPP packets and encapsulating the PPP packets in GRE[33, 34] packets.

The most widely used implementation of PPTP is Microsoft's implementation which first shipped with Windows NT 4. It can use one of three authentication schemes: an unencrypted clear-text password, a hashed password or challenge-response using MSCHAP-v2 (Microsoft Challenge Handshake Authentication Protocol, version 2). If the third authentication option is used, optional packet encryption can be provided with MPPE (Microsoft Point-to-point Encryption).

Generic Routing Encapsulation (GRE)

> "[GRE is] a protocol for performing encapsulation of an arbitrary network layer protocol over another arbitrary network layer protocol."[33]

Generic Routing Encapsulation provides a simple protocol for encapsulation of packets at the network layer. Several standards specify the details of providing encapsulated GRE channels over a number of network layer protocols. PPTP builds on the IPv4 GRE standard detailed in "Generic Routing Encapsulation over IPv4 networks (RFC1702)"[34].

B.2.2 Analysis of Protocol

Schneier and Mudge

In "Cryptanalysis of Microsoft's Point-to-point Tunneling protocol" [35], Schneier and Mudge present a number of attacks against Microsoft's implementation of PPTP. The most critical of these is an attack against MS-CHAPv1, which was replaced with MS-CHAPv2 in response to this paper and others.

The paper also presents a number of security weaknesses in the MPPE encryption protocol. The most critical of these is that the protocol is keyed using an SHA hash of the user's password. As a result of this method of key generation, the encryption key only has the same amount of entropy as the user's password - making dictionary based and brute force key recovery attacks a possibility. The use of cryptographically secure key generation algorithms (such as those recommended by the IPsec specification) and key distribution algorithms (such as IKE) would have dramatically strengthened Microsoft PPTP.

> "Microsoft's PPTP implementation is fragile from an implementation perspective, and seriously flawed from a protocol perspective." [35]

Wagner, Schneier and Mudge

In response to [35], Microsoft improved their PPTP implementation by introducing MS-CHAPv2, a more secure authentication protocol. In "Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2)."[36], Wagner, Schneier and Mudge present a complete analysis of the new protocol. Weaknesses of the new protocol against version rollback attacks (forcing the server to use MS-CHAPv1), dictionary attacks and cryptographic attacks are discussed. The authors conclude that the changes to authentication and encryption in MS-CHAPv2 greatly strengthen the algorithm. The greatest concern about the new protocol is that key generation is still based on the user's password.

> "However, the fundamental weakness of the authentication and encryption protocol is that it is only as secure as the password chosen by the user."[36]

> "Our hope is that PPTP continues to see a decline in use as IPsec becomes more prevalent."[36]

B.3 Layer 2 Tunneling Protocol (L2TP)

B.3.1 Description of Protocol

Layer 2 Tunneling Protocol (L2TP)[37] is an extension and generalisation of the Point-to-point protocol (PPP)[38]. L2TP is a data-link layer protocol which transports client data over an existing packet based network. L2TP is an extremely flexible protocol, which is applicable to a large number of applications. Most commonly it is used to provide dial-in like connections to remote networks over the internet.

L2TP and IPsec

L2TP provides security for tunneled data using PPP encryption and authentication services. These services have a number of major drawbacks: they do not protect the control channel data and do not address integrity checking, key management and replay protection. While PPP authentication serves to authenticates hosts on a per-connection basis, it does not perform per-packet authentication.

In order to use L2TP as a secure VPN protocol, L2TP can be combined with a network layer security protocol - most typically IPsec. In "Securing L2TP Using IPsec (RFC3193)"[39], the authors describe a protocol for protecting L2TP tunnels with IPsec's ESP protocol in transport mode. The IKE protocol[28] is used for key generation and distribution. This protocol does not provide end-to-end security - it only encrypts the data for the time it is inside the L2TP tunnel. If end-to-end security were required, an IPsec or TLS connection could be used to secure communication between the end points.

B.4 Transport Layer Security

B.4.1 Protocol Description

Transport Layer Security (TLS)[40] and it's predecessor Secure Sockets Layer (SSL) are protocols designed to provide encryption and authentication to the connection between two applications. TLS provides security to the transport layer of the protocol stack - it uses a reliable transport layer protocol (such as TCP) and it's own services to provide encryption, authentication and reliability to application layer protocols. The TLS standard specifies two protocols - the *record protocol* and the *handshake protocol*.

The TLS *record protocol* provides connection privacy through use of symmetric cryptography and connection reliability using a keyed MAC. A variety of symmetric cryptography algorithms are supported and can be negotiated between client and server when the connection is established. RC4, RC2, DES and 3DES are supported, with a variety of key lengths.

The TLS *handshake protocol* uses an asymmetric encryption algorithm (RSA and DSS are supported) to provide optional authentication to one, or both, peers. In many real-world implementations only one of the peers, the server, is authenticated. Distribution of key matter to be used by the *record protocol* is also handled by the *handshake protocol*. The protocol ensures that the shared secret is distributed both securely (an eavesdropper cannot extract the secret) and reliably (an attacker cannot modify the key in transit) to both parties.

B.5 Selection of Protocol

B.5.1 Objective Protocol Selection

There are a number of important factors to consider in the choice of a VPN protocol for implementation on the IXP2400 network processor. These factors include: security, simplicity and efficiency. While these three factors are extremely important the most important single factor is industry acceptance - it makes little sense to implement a protocol which is not widely supported.

The technique chosen to choose the ideal protocol is to assign a score to each protocol in each of four categories - simplicity, security, efficiency and acceptance. Weights are then applied to each category and a final score is calculated for each protocol. This score will be a valuable tool in making the final selection of a protocol. The score in each category for each protocol is based, as far as possible, on reports of practical experiences of implementing each protocol in a production environment.

	Simplicity	*Security*	*Efficiency*	*Acceptance*	*Total*
IPsec	7	8	6	7	28
PPTP	8	3	6	8	25
L2TP+IPsec	5	9	3	7	24
TLS	4	6	5	6	21

Simplicity

The simplicity score is a measure of the simplicity of implementing each protocol as a VPN gateway protocol. The complexity of the protocols them-

selves, any support protocols needed and cryptographic algorithms used were taken into account.

IPsec Only ESP and tunnel mode were taken into account - ESP can provide all the functions of AH and transport mode is not applicable to VPN gateways.

L2TP In order to provide acceptable security, L2TP relies on a protocol such as IPsec, which greatly increases its complexity.

TLS While TLS can be implemented as a gateway-to-gateway VPN protocol, such an implementation is likely to be complex and fragile.

Security

The security score reflects the strength of the algorithms, measured by the number and severity of any public attacks. Security features provided by the algorithms, such as authentication, privacy, replay protection and flow confidentiality were also taken into account.

PPTP There are a number of published attacks against PPTP and some concerns about it's key generation methods.

L2TP L2TP provides flow confidentiality in addition to the security services provided by IPsec.

TLS While there is little doubt as to the security of the TLS and SSL 3.0 protocols, it is not clear whether they will provide security when deployed as a VPN protocol.

Efficiency

This score reflects the relative performance that can be expected from a network gateway implementing each of the protocols. Both processing time (added latency and reduced bandwidth) and added network overhead are taken into account.

IPsec IPsec's performance is entirely dependent on the protocols and encryption algorithms used, but has small overhead and excellent performance in some configurations.

TLS The throughput of SSL is lower than the throughput of IPsec on the same hardware [41].

L2TP L2TP's performance is dominated by the performance of the IPsec encryption used on the tunnel. Protocol overhead is larger than IPsec.

Acceptance

Acceptance reflects how widely the protocol is accepted as a gateway-to-gateway protocol by both industry and communications researchers.

IPsec Generally accepted as the most viable protocol for implementation in new products, IPsec has found wide acceptance in academia and industry. An IPsec implementation is available for Windows, Linux and many other operating systems.

PPTP The most widely spread and widely used VPN protocol, because of it's inclusion in Microsoft Windows. PPTP is widely deployed, used and implemented. Implementations are available for Windows, Linux, OSX and some other operating systems.

L2TP L2TP+IPsec is widely accepted and deployed, especially as a remote access tunneling protocol. Implementations are available for Windows, Linux and other operating systems.

TLS While TLS and SSL are possibly the most widely used of all the above protocols due to their use to provide security to web applications and their implementation in most web browsers, they are not widely used as gateway-to-gateway VPN protocols.

B.6 Conclusion

IPsec appears to be the most suitable Virtual Private Networking protocol for implementation in this project. Several factors are in IPsec's favour. IPsec is extremely widely implemented in and accepted in the communications industry and offers superior security to it's most direct competitor, PPTP. IPsec was also designed as a gateway-to-gateway virtual private networking protocol, which makes it more suitable than SSL and TLS. It is difficult to justify the implementation of the extra complexity of L2TP for use as a portal-to-portal VPN protocol.

Appendix C

Internet Protocol Security

C.1 The IPsec Protocol

Internet Protocol Security[20], or IPsec, was briefly introduced in Section B.1. In this section, IPsec for IPv4 will be more completely described, with emphasis on the Encapsulating Security Payload (ESP)[16] protocol and tunnel mode operation. Emphasis will also be placed on issues regarding the implementation of IPsec on a "Bump-in-the-wire".

C.1.1 Security Associations

The operation of IPsec is based on the concept of Security Associations. A Security Association (SA) is a connection between two hosts which defines security parameters for traffic being carried by the connection. Security Associations are simplex connections - for bi-directional traffic two Security Associations are required.

Security Associations are stored by the end points in a database. The IPsec standard [20] recommends that IPsec compliant devices keep two databases - a Security Association Database containing details of all the active security associations in use by the device and the Security Policy Database which stores general policies regarding the establishment of new Security Associations. Each network interface on an IPsec device needs a two sets of databases - one for inbound traffic and one for outbound traffic.

C.1.2 Transport Mode and Tunnel Mode

The IPsec protocols support two modes of operation.

Transport Mode

Transport mode applies the IPsec protocols to packets as they are sent over the network, providing protection to upper-level protocols and packet data, including the TCP header. Figure C.1 illustrates which parts of the original IP packet are protected using the ESP protocol in transport mode. In this figure, sections of the packet containing data from the original packet have a hatched background. As can be clearly seen from this diagram, some data which may be interesting to an attacker, including the IP header of the original packet, is not encrypted or authenticated by the ESP protocol.

Encryption and authentication are performed on all upper level protocol headers of the IP packet and it's entire data payload. For applications that demand authentication and encryption of packet data but do not require stream anonymity or enhanced replay protection, the reduced overhead of transport mode when compared to tunnel mode makes this mode attractive for end-to-end implementations. Transport mode is not suitable for gateway-to-gateway or bump-in-the-wire IPsec implementations.

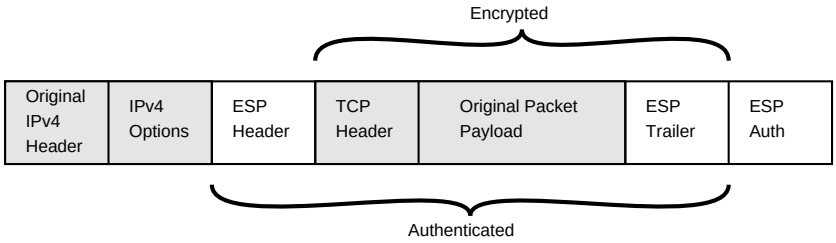


Figure C.1: ESP Transport Mode (from [20])

Tunnel Mode

IPsec operating in tunnel mode encapsulates the contents and headers of the IP packets it is protecting within an outer IP packet. This encapsulation for tunnelling is similar to the operation of other IP-over-IP tunneling protocols, such as GRE. The ESP protocol operating in tunnel mode offers confidentiality and authentication protection for the entire IP packet, including all headers.

Tunnel mode ESP is illustrated in figure C.2. The entire inner IP packet (marked with a grey background in the figure) is protected. This affords greater stream anonymity and replay protection that ESP operating in transport mode.

Tunnel mode is suitable for both end-to-end and gateway-to-gateway implementations and is more simple to implement without knowledge of the

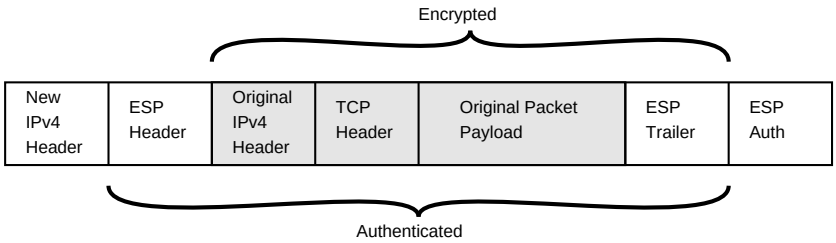


Figure C.2: ESP Tunnel Mode (from [20])

state of either the source or destination. These properties make tunnel mode ideal for bump-in-the-wire (IPsec gateway) implementations.

C.1.3 Encapsulating Security Payload

Described in the paper "IP Encapsulating Security Payload (ESP)" by Kent and Atkinson, is a header designed for use with IPsec, which provides a set of security services to IP traffic. ESP can be applied to both IPv4 and IPv6 traffic with few modifications. The security services provided by ESP include authentication, confidentiality, integrity, sequence integrity and traffic flow confidentiality, depending on the details of the Security Association (SA) between the hosts using the protocol. Users can select to use either confidentiality (and associated services), authentication (and associated services) or both - but not neither.

The ESP Header

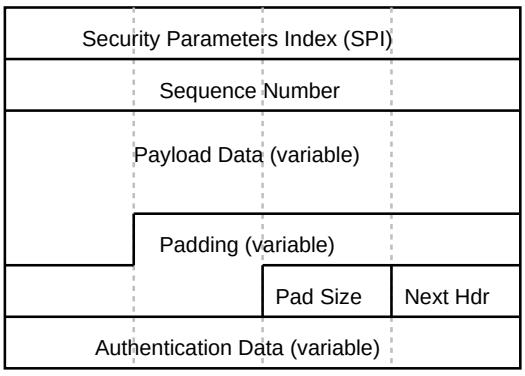


Figure C.3: The ESP Header (from [20])

The ESP header consists of seven fields, as illustrated in figure C.3:

Security Parameters Index The SPI uniquely identifies the security association used for transmission of the packet.

Sequence Number The sequence number is a counter value which must be increased by the sender for each packet which is sent. If the sequence number counter were to roll over, a new Security Association must be negotiated.

Payload Data In tunnel mode, this field contains the entire encapsulated datagram in encrypted form. Initialization vectors for the ciphers used are also transferred in this section.

Padding Payload data must be padded prior to encryption to fit into an integral number of cipher blocks.

Pad Size The length of the pad.

Next Header The protocol contained in the payload, uses one of the IANA assigned protocol values.

Authentication Data A cryptographic hash of the entire ESP header.

C.1.4 Internet Key Exchange

Security Associations (SAs) between IPsec hosts are established, maintained and removed using the *Internet Key Exchange* (IKE) protocol[28]. The IKE protocol, in turn, depends on the *Internet Security Association and Key Management Protocol* (ISAKMP) which is an extremely complex framework for the definition of security association negotiation protocols and a partial Security Association establishment and maintenance protocol.

Security Association between two IPsec hosts occurs in two phases. The first phase is the establishment of a ISAKMP Security Association between the hosts. The seconds phase uses this ISAKMP SA to negotiate a pair of IPsec security associations. Negotiation of a pair of Security Associations requires the two hosts (or gateways) to agree on which authentication (SHA-1, MD5, etc) and encryption algorithms (DES, NULL, AES, etc) will be allowed, encryption parameters such as key length and Initialization Vector, and how long the SA will last before it is re-negotiated.

Diffie-Hellman Key Exchange

The IKE protocol uses an authenticated version of the cryptographic technique known as Diffie-Hellman Key Exchange for the exchange of a pair of

encryption keys. Diffie-Hellman allows to hosts to exchange encryption keys (or any other information) securely, without any prior knowledge or shared information, even if the network between them is insecure.

A simplification of the Diffie-Hellman protocol proceeds as follows:

1. Alice and Bob swap, over the open network, a large primes p and a generator g
2. Alice chooses a random natural number, A_a as her private key
3. Bob chooses a random natural number, B_a as his private key
4. Bob calculates $B_b = g^{B_a}$ modulo p and sends it to Alice, over the open network
5. Alice calculates $A_b = g^{A_a}$ modulo p and sends it to Bob, over the open network
6. Bob calculates $k = (A_b)^{B_a}$ modulo p and Alice calculates $k = (B_b)^{A_a}$ modulo p
7. Alice and Bob now have the same value k

The Diffie-Hellman protocol is described more completely in "Diffie-Hellman Key Agreement Method" by Eric Rescorla, RFC2631[42].

C.2 Overhead of IPsec

In this section we will consider the transfer size overhead of implementing IPsec on a typical packet network, such as an Ethernet. We will consider IPsec for IPv4 in both tunnel mode and transport mode with the Encapsulating Security Payload (ESP) protocol.

Overhead of ESP

The base size of the ESP header is 10 bytes, including the SPI, Sequence Number, Pad Size and Next Header fields. In addition to these fields, in a typical implementation such as the one we present, a 12 byte (96 bit) HMAC-SHA-1-96 authentication block will be present. The payload also needs to be padded to a multiple of the cipher's block length. The AES cipher used in the IPsec implementation we present uses a block length of 16 bytes - therefore a maximum of 15 bytes of padding will be needed.

The total size of the ESP header is between 22 bytes and 37 bytes, depending on the amount of padding needed.

Tunnel Mode Overhead

In addition to the ESP header, a packet protected using IPsec's tunnel mode will require an IP packet header. The minimum size of the IPv4 header, if no options are used, is 20 bytes. In transport mode a second IP header is not required, so no extra overhead will be introduced.

C.2.1 Overhead calculations

Protocols such as FTP and HTTP generate large packets. If we assume these packets are all 1500 bytes (the Ethernet MTU) long after encapsulation, the the overheads will be as follows. For tunnel mode:

$$1 - \frac{S_{avg}}{S_{tunnel}} = 1 - \frac{1443}{1443 + 57} = 3.8\%$$

For transport mode:

$$1 - \frac{S_{avg}}{S_{transport}} = 1 - \frac{1463}{1463 + 37} = 2.5\%$$

In neither case does the ESP and IPsec overhead make a significant contribution to the packet size.

For protocols and applications which generate small packets, such as VOIP and Telnet, the bandwidth overhead of IPsec can be significant. Assuming an average packet size of 64 bytes, and worst case padding the overheads will be as follows. For tunnel mode:

$$1 - \frac{S_{avg}}{S_{tunnel}} = 1 - \frac{64}{64 + 57} = 47\%$$

For transport mode:

$$1 - \frac{S_{avg}}{S_{transport}} = 1 - \frac{64}{64 + 37} = 37\%$$

In both these cases, the IPsec packet size overhead could be extremely significant. Implementation if ESP in tunnel mode could reduce the amount of VoIP traffic a network can transfer by as much as 50%. If other IPsec related services are used - such as UDP encapsulation of IPsec packets for purposes of NAT and firewall traversal, this overhead will be increased even further.

C.3 Secure Hash Algorithm (SHA-1)

C.3.1 Background

A hash function is a function which maps a large input space onto a considerably smaller output space. Cryptographic hash functions are functions which perform this mapping in a way which them useful for implementing message authentication and other security operations. There are three important measures of the security of a cryptographic hash function (from [43]):

Collision resistance It is computationally infeasible to find x, y, x != y such that H(x) = H(y).

Preimage resistance Given an output value y, it is computationally infeasible to find x such that H(x) = y.

Second preimage resistance Given an input x', it is computationally infeasible to find x such that H(x) = H(x').

Simply stated, *Collision Resistance* a measure of the difficulty of finding two pieces of data which have the same hash value. *Preimage Resistance* is the difficulty of extracting the original data from the hashed value and *Second Preimage Resistance* is a measure of the difficulty of finding data which has the same hash value as another, given, piece of data.

The most important properties hash function used to provide authentication in a protocol such as ESP is Second Preimage Resistance. This is because the hash is used to indicate that the message contained in each packet has not been altered. If it were possible to find a message with a different meaning to the one contained in the packet but with the same hash, the hash would not be effective in ensuring that message data has not been altered.

There are many hash functions which are used widely in industry. Possibly the most well known and widely spread is MD5. The second most widely well known and implemented is SHA-1, which has seen wide adoption following growing concerns about the security of MD5[44]. Other hash functions, such as RIPEMD-160 and the other members of the SHA family - SHA-224, SHA-256, SHA-384 and SHA-512 - are also widely used in real-world implementations.

The selection of SHA-1 as the cryptographic hash function primitive to use in the implementation of IPsec was one that was carefully considered. Despite recent breakthroughs in finding collisions in SHA-1 hashes (see section C.3.4 on page C-9), it is still considered secure enough[43] for use as

a hash function to be combined with HMAC. SHA-1 offers superior performance to the rest of the SHA family and RIPEMD-160, while being simpler to implement. MD5 is attractive because of it's excellent performance and simplicity, but recent attacks have shown that it's collision resistance is extremely weak[44]. Although the WHIRLPOOL hash function (which shares a common ancestor, the Square block cipher, with AES) is believed to be secure, is fairly simple and can be implemented extremely efficiently on most hardware it is not suitable because its lack of wide adoption would effectively limit interoperability.

C.3.2 The Keyed-Hash Message Authentication Code (HMAC)

IPsec uses the HMAC[45] algorithm, along with a cryptographic hash algorithm, to provide message authentication codes for use with the ESP and AH protocols. The HMAC algorithm is a keyed function which makes use of a block of key data and a cryptographic hash function to generate key-dependent digests of a specified length.

The definition of HMAC

Given a cryptographic hash function H with block size B, a block of data D and a block of key data K, HMAC defines a simple procedure for calculating a message authentication code. The algorithm proceeds as follows:

1. Define ipad to be the value 0x36 repeated B times and opad to be the value 0x5C repeated B times
2. Pad the key data K with 0x0 bytes to a total length of B bytes
3. XOR the padded key data with the ipad and append the data block, D, to the result
4. Calculate the hash value I of the data and key block
5. XOR the padded key data with the opad and append the hash value I to the result
6. Calculate the hash value R of the key and I block

Thus, HMAC calculates the value $R = H(K \oplus opad|H(K \oplus ipad|D))$ where K is the padded key data, $\oplus$ is the XOR (bitwise exclusive OR) operator and | is the concatenation operator.

Truncation

In order to use the output of HMAC with a protocol such as ESP, the output must be truncated. RFC2404[46] specifies the use of HMAC-SHA-1-96 with ESP and AH. HMAC-SHA-1-96 means that the MAC expected is the first 96 bits (12 bytes) of the output of HMAC using SHA-1 as a cryptographic primitive. This truncation is only performed on the final output of the HMAC function - it is not performed on the output of the inner hash function. For example, if the 160 bit output of HMAC-SHA-1 is 0x4c1a03424b55e07fe7f27be1d58bb9324a9a5a04 then the 96 bit output of HMAC-SHA-1-96 would be 0x4c1a03424b55e07fe7f27be1.

C.3.3 The definition of SHA-1

A complete explanation of the SHA-1 algorithm is presented in FIPS-180-1 and FIPS-180-2[21] and a similar explanation and sample code can be found in RFC3174[47]. Complete C code, written by the author, for an implementation of HMAC-SHA-1 is on the Compact Disc attached to this document.

SHA-1 functions by taking the data to be hashed, padding it to fit into a number of 64 byte blocks and repeatedly perturbing each byte of the block. The inner loop which performs the perturbations is fairly simple, but still provides a good basis for a hash function. In C, the loop appears as follows:

```
for (t = 0; t < 80; t++)
{
  T = rotl5(a)+ft(b, c, d, t)+e+kt(t)+sched.data[t];
  e = d;
  d = c;
  c = rotl30(b);
  b = a;
  a = T;
}
```

Where *rotl* is a bitwise left rotation, *ft* is a bitwise mixing function and *kt* is a constant.

C.3.4 Security of SHA-1

In August 2005, researchers Xiaoyun Wang, Yiqun Yin and Hongbo Yu from Shandong University announced a new attack against SHA-1[48], which can produce two distinct messages with the same hash with complexity 2^{63} (approx. 10^{19}), instead of the 2^{80} (approx. 10^{24}) which would be required for a

brute force search. This attack makes finding collisions in SHA-1 practical, which severely reduces it's usefulness as a cryptographic hash function.

These attacks are not currently relevant to the use of SHA-1 as a hash function in the context of IPsec. Security researchers currently believe that the use of SHA-1 with HMAC is still safe, as the addition of key material makes the presented attacks impractical. In "Deploying a New Hash Algorithm"[43], Bellovin and Rescorla write:

> "Furthermore, HMACs are probably safe, since the unknown component - the key - of the inner hash function makes it impossible to generate a collision at that stage; this in turn helps protect the outer hash."

C.4 Advanced Encryption Standard (AES)

C.4.1 Background

In September 1997, the National Institutes of Standards and Technology (NIST), from the USA, made a call for the submission of encryption algorithms to a competition to find the replacement for the Defence Encryption Standard (DES). The purpose of the AES competition was to replace DES with a fast (in hardware and software on a number of platforms) and secure symmetric block cipher which could be used in both commercial and government applications. Fifteen different ciphers were submitted to the competition — CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, RC6, Rijndael, SAFER+, Serpent and Twofish. Each of these algorithms was presented at the first AES conference in August 1998.

At the second AES conference, held in March 1999, papers were presented detailing the cryptanalysis of many of the ciphers presented in the first round. Following this conference, a shortlist of five ciphers was announced: MARS, RC6, Rijndael, Serpent and Twofish. In December 2000, the Rijndael cipher was selected, with some minor modification, as the AES.

AES, as standardised in FIPS 197[49], is symmetric block cipher based on a substitution permutation network. It has a block size of 16 bytes (128 bits) and supports key sizes of 128, 192 and 256 bits.

C.4.2 Use of AES with IPsec

The original IPsec specification [20, 16] specified two encryption algorithms for use with ESP. These algorithms are the Defense Encryption Standard (DES) which has severely limited key size and is not believed to be secure

enough for sensitive applications and the NULL algorithm which is a placeholder to represent no encryption. Following the completion of the AES certification process, the use of the AES algorithm with ESP was standardised in "The AES-CBC Cipher Algorithm and Its Use with IPsec (RFC3602)" by Frankel, Glenn and Kelly.

AES has several advantages over DES. AES supports longer keysizes than either DES or triple DES, making it more secure against brute force attacks and is believed to be a more secure cipher. AES is also significantly faster and more efficient than DES in software and hardware implementations.

C.4.3 Elements of The AES Algorithm

Finite Field Arithmetic

All byte-level arithmetic operations in AES are done with bytes representing elements in the Galois Field GF(2^8) with the reducing polynomial $m(x) = x^8+x^4+x^3+x+1$, the first irreducible polynomial of degree 8. Explanations of the algorithms for arithmetic operations in this space can be found in "The Design of Rijndael" by Daemen and Rijmen[50] along with a justification of the choice of this field for the cipher.

The *SubBytes* Transformation

Each byte in the 16 byte AES state block is replaced with the corresponding byte in an 8bit S-box. The values of the S-box are calculated using an affine transformation of the multiplicative inverse for the number in AES's finite field. For each byte x in the box, the multiplicative inverse y is found, such that $x \times y = 1$ in the finite field, then the bits of y, $[y_0 \dots y_7]$, are transformed using the following affine transformation:

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The inverse *SubBytes* transform is similar to the forward transform, with a change of S-Box. The S-Box used for the inverse transform is the inverse of the one used for the forward *SubBytes* transform.

The *ShiftRow* Transformation

In this transformation, each row of the AES state is shifted over by a number of bytes. The first row is not shifted, the second row is shifted left one byte, the third left two bytes and the fourth left three bytes. Refer to Figure C.4 for a graphical representation of this transformation. The *ShiftRow* transformation ensures good mixing between the columns of the state.

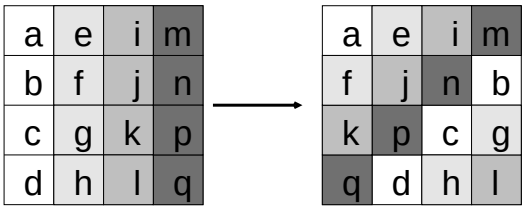


Figure C.4: The AES Shift Rows Transform

The inverse of the *ShiftRow* transformation is trivial - the first row is not shifted, the second shifted right by one element and so on.

The *MixColumns* Transformation

For the *MixColumns* transformation, each column of the AES state treated as polynomials in GF(8). The transform multiplies each column polynomial by $c(x)$, modulo $x^4 + 1$ where $c(x) = 3x^3 + x^2 + x + 2$. This transformation is also representable as the matrix multiplication:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

The inverse transform is similar - a multiplication with $d(x)$ modulo $x^4 + 1$ where $d(x) = 11x^3 + 13x^2 + 9x + 14$

The *AddRoundKey* Transformation

In this transformation each byte of the round key (derived from the key schedule, see Section C.4.4 on page C-13) is bitwise exclusive ORed (XOR) with the corresponding byte of the AES state. Due to the symmetry of the XOR function ($y \oplus (x \oplus y) == x$) the *AddRoundKey* transformation is it's own inverse.

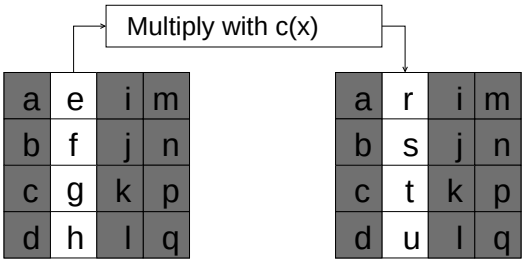


Figure C.5: The AES MixColumns Transform

C.4.4 AES Key Schedule

The *Key Schedule* is an algorithm for deriving the round keys from the cipher key. The cipher key needs to be expanded to $128(R+1)$ bits of round keys where R is the number of cipher rounds. The first four 32 bit words of the cipher key are copied directly to the expanded key. A simple algorithm is followed for the rest of the expansion, illustrated by this pseudocode:

```
def gen_key(key):
  exp_key[0:4] = key[0:4]
  for i in range(4, 44):
    w = exp_key[i-1]
    if (i&0x3) == 0:
      w=rol(w)
      w=bytesub(w)
      w=w^rcon((i-4)/4)
    w = w^exp_key[i-4]
    exp_key.append(w)
  return exp_key
```

The *SubByte* function is equivalent to the forward *SubByte* transformation discussed above, executed on each byte of a 32 bit word. The *RotByte* function takes a 32 bit word and performs a logical left rotate by 8 bits. *Rcon* is defined as:

$$Rcon[i] = \left(\begin{cases} 1 & \text{for } i = 1 \\ 2Rcon[i-1] & \text{for } i > 1 \end{cases}\right) \ll 24$$

with multiplication taking place in AES's finite field and $\ll 24$ representing a bitwise left shift by twenty four bits. The Rcon function can be implemented with a simple lookup table.

```
const unsigned char rcon_data[10] = {0x01, 0x02, 0x04,
   0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36};
rcon[i] = rcon_data[(i-4)>>2] << 24
```

C.4.5 The AES Algorithm

AES Forward Transform (Encryption)

The AES encryption operation (referred to as the Forward Transform in some literature) consists of ten rounds for 128bit keys, twelve rounds for 192bit keys and fourteen rounds for 256bit keys. The number of rounds is increased with the key length to ensure that indirect cryptanalytic attacks (attacks that do not involve key guessing) do not become relatively easier to perform than brute force attacks as key lengths increase.

Each round of the cipher involves the *Sub Bytes* transformation, the *Shift Rows* transformation, the *Shift Columns* transformation and mixing of key material, in that order. A pseudocode representation of the encryption function for a 128bit key (176 bytes of key data) appears as follows:

```
def encipher(state, key):
  add_round_key(state, key[0:4])
  for round in range(1, 10):
    sub_bytes(state)
    state = shift_rows(state)
    mix_columns(state)
    add_round_key(state, key[4*round:(4*round+4)])
  sub_bytes(state)
  state = shift_rows(state)
  add_round_key(state, key[40:44])
  return state
```

Each round of the forward transform can be simplified to a number of lookups into a set of four 1024 byte tables. This approach offers superior performance to the naive implementation described above, especially on 32bit hardware (such as the Intel IXP2400). The process of generating the required lookup tables and a proof that the lookup table approach is equivalent to the naive algorithm is presented in Section 5 of [50].

The AES Reverse Transformation (Decryption)

Two versions of the AES decryption algorithm are presented in the AES documentation [49, 50]. The first of these uses the same key schedule as the forward transform, but the reverse order of operations. While this decryption algorithm is conceptually simple (it "unwraps" the operations of the forward transform), it cannot be optimised to use a series of lookup tables on 32bit hardware.

A more suitable decryption algorithm for optimisation on 32bit hardware is referred to as the "Equivalent Inverse Cipher". This algorithm uses the same order of operations as the forward transform algorithm (using the inverse of each operation), but uses a modified key schedule. Following the expansion of they key, all but the first four and last four 32bit words of the expanded key are processed with the *Inverse Mix Column* operation, described above. While the added complexity does make the key schedule slower, this loss of performance can easily be made up for (in all but the most key-agile and memory limited environments) by the fact that it allows lookup table optimisations on 32 bit hardware.

A naive pseudocode implementation of the equivalent inverse cipher for 128bit keys is:

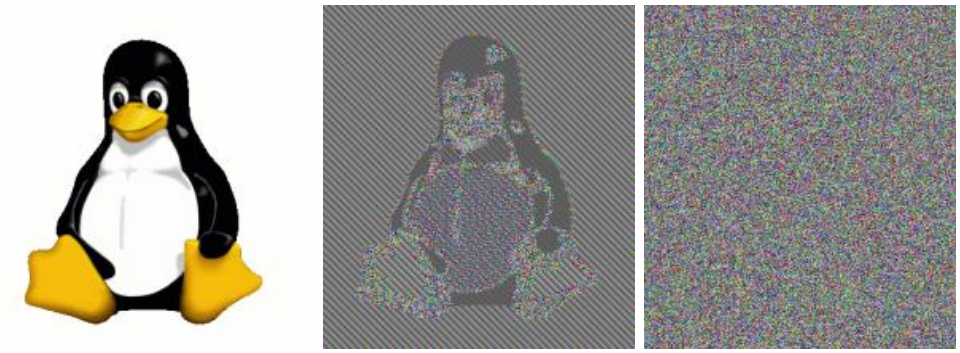
```
def decipher(state, key):
  add_round_key(state, key[40:44])
  for round in range(10, 1):
    inv_sub_bytes(state)
    state = inv_shift_rows(state)
    inv_mix_columns(state)
    add_round_key(state, key[4*round:(4*round+4)])
  sub_bytes(state)
  state = inv_shift_rows(state)
  inv_add_round_key(state, key[0:4])
  return state
```

C.4.6 Modes of Operation

Two IETF standards specify the use of AES with IPsec. The first standard, "The AES-CBC Cipher Algorithm and Its Use with IPsec"[19] specifies the use of AES in Cipher Block Chaining mode while the second standard, "Using AES Counter Mode with IPsec" specifies the use of AES in Counter Mode. Five modes of operation are commonly used with block ciphers - Electronic Code Book (ECB), Counter Mode, Cipher Block Chaining (CBC), Output Feedback (OFB) and Cipher Feedback(CFB).[51, 52, 53]

The most obvious usage mode for a block cipher is one where each block of the input is enciphered independently to produce a single block of the ciphertext — a mode known as Electronic Code Book (ECB). This mode of operation, while simple, does not provide adequate security for production use. This is because ECB mode does not hide statistical patterns in the plaintext, vastly simplifying the task of cryptanalysis. An example of pat-

terns remaining after encryption in ECB mode is presented in Figure C.6[1].



(a) Original Image

(b) Image Encrypted in ECB Mode

(c) Image Encrypted in CBC Mode

Figure C.6: Data Patterns Evident in Electronic Code Book Ciphertext (from [52])

The most common alternative to ECB mode is *Cipher Block Chaining* (CBC). In this mode of operation, the ciphertext of the previous block is bitwise exclusive ORed (XOR) with the plaintext of the current block before the block is encrypted. The first block of plaintext is XORed with an Initialisation Vector (IV) which is known by the party decrypting the data stream. CBC mode effectively hides patterns in the plaintext.

In counter mode, the block cipher is used to encrypt a counter (a monotonically increasing value in most implementations), which produces a value which is then XORed with the plaintext to provide the ciphertext. In this mode of operation, a block cipher can be used as a stream cipher. In counter mode, unlike CBC mode, multiple blocks of plaintext can be encrypted in parallel, which can increase the performance of the cipher on parallel architectures.

C.4.7 Security of the Advanced Encryption Standard

There have been several attacks published on reduced round variants of the AES algorithm which perform better than a brute force search of the key space. The best attack, currently, can break seven (out of ten) rounds of the AES algorithm for 128 bit keys. This attack is presented in the paper "Improved Cryptanalysis of Rijndael" by Ferguson, et al[54]. It uses a chosen

[1]Tux image copyright Larry Ewing (lewing@isc.tamu.edu), used with permission.

plaintext attack which can break seven rounds of the cipher with a complexity of between 2^{120} and 2^{172}, depending on key length and the amount of memory used.

The same attack can break eight rounds of the cipher for 192 and 256bit keys. Another attack, using "related keys", is presented in the same paper and can break nine rounds of 256bit AES with a time complexity of 2^{224}. The authors of the paper do not appear to believe that these attacks can be extended to the full number of rounds of AES.

> "Our results have no practical significance for anyone using the full Rijndael."[54]

Investigations of the mathematical properties of AES (especially it's S-Boxes) have recently revealed that these boxes do not provide the amount of nonlinearity to the cipher that the original authors claimed. In "Cryptanalysis of Rijndael S-Box and Improvement", Jing-mei, et al[55]. present an improved version of the S-Box that they claim will increase the amount of nonlinearity in the cipher and imply that this will increase the security of the AES algorithm. Whether these findings, and other similar findings, will lead to successful attacks against the current AES algorithm is unclear.

Based on a study conducted by the National Security Agency (NSA) of the United States of America, the Center for National Security Studies (CNSS) has accepted the AES for protection of Classified, Secret and Top Secret information in that country. This is significant as AES is the first a published algorithm has been accepted for use with Top Secret information in the United States.

> "(6) The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths." CNSS Policy 15 [56]

The Advanced Encryption Standard is rapidly becoming the industry standard for deployment in a wide variety of systems, including Virtual private networks. Current research indicates that this cipher is strong enough to ensure that data encrypted with it will remain private for the foreseeable future.

Appendix D

Implementation of Design

D.1 Microengine C

The design presented in Chapter 4 was implemented in Microengine C[57], a dialect of the C programming language supported by the Intel Developer Workbench 4.0 IDE and compiler.

While Microengine C is a complete language, it does not support many of the features expected by C programmers on general purpose microprocessors. Features which are unsupported include:

- A call stack or local data stack. Variables local to functions are allocated to memory or registers at compile time. Memory and registers allocated to `automatic` variables (local variables which are not `static`) can be used by multiple functions, as long as their scopes do not overlap.
  - "Function calls are implemented by loading a register with the return address and jumping to the function."[57].
- Misaligned or byte aligned access. Pointers to external memory must point to a four byte (for SRAM) or eight byte (for DRAM) aligned address. Byte aligned access can be performed with compiler intrinsics.
- The standard C library. A very minimal subset of the C library is provided.
- Floating point arithmetic. The IXP2400 microengines do not have a floating point unit and the compiler does not provide floating point emulation support.

- Recursive functions. Due the the lack of a call stack, recursive functions are not supported by the compiler.
- Function pointers. Pointers to functions and pointers to types which, when dereferenced, would become pointers to functions are not supported.
- The address operator (`&`) can not be applied to variables which are stored in registers.

The development tools for the IXP2400 network processor support two languages - Microengine C and Microengine Assembler. Microengine assembler is a rich, high level assembly language exposing a very limited instruction set with high per-instruction functionality. Programming the IXP2400 Microengines in assembler would have allowed better use of the advanced features of the Microengines which are not fully exposed by compiler intrinsics.

Despite this advantage of assembler, Microengine C was chosen for the implementation of the IPsec gateway design. This choice was made based on several advantages of C over assembly language. A high level language such as C allows code to be developed more quickly - the programmer does not have to be concerned with the underlying instruction set. C code is also easier to read and debug than assembler code — this makes the code more valuable to others who wish to analyse or extend it.

The use of a high level language simplified the design of the software as details such as memory addresses and register allocation did not have to be included in the design. The Intel Microengine C compiler supports extensive compile-time optimisation of C code. It is likely that an inexperienced assembler programmer would be unable to produce code which runs faster than compiler produced code.

D.2 Details of Implementation

D.2.1 Receive Program

The receive program was broken up into four source files. Interfacing with the MSF and packet receive functionality was placed in *rx.c*. The multiple producer-consumer ring buffer code was included in *mcprb.c* while the main program functionality was split into *test.c* and *packetprocess.c*.

File	Description	Lines
rx.c	MSF Receive Interface	154
mcprb.c	Multiple Consumer Ring Buffer	168
packetprocess.c	Packet Processor Arbiter	137
test.c	Main Program	122
	Total	581

The total number of lines of Microengine C code in the receive program is 581. In order to simplify the code in *packetprocess.c*, this file was written to be modified by the C preprocessor and included six times — one per processing microengine. Due to this, the total number of lines that are compiled for the receive program is 1266.

D.2.2 The Packet Processing Program

The C preprocessor was used to produce six similar packet processing programs with different addresses and variable names. The majority of packet processing functionality was placed in *hash_engine_template.c* which is customised for each ME and included in *hash_engine[123456].c* to form the main program of the packet processor.

File	Description	Lines
sha1.c	SHA-1 and HMAC implementation	388
aes_fast.c	AES implementation	289
hash_engine_template.c	Template for Packet Processor	119
hash_engine1.c	Main Program	107
	Total	903

The SHA-1 and HMAC implementations was placed in *sha1.c* and *hmac-sha1.c*. The AES implementation was placed in *aes_fast.c*, and automatically generated AES lookup tables were placed in *aes_table.h*. Support programs, written for a PC, were used to automatically generate lookup tables for the AES implementation.

D.2.3 The Transmit Program

The transmit program is considerably simpler than the receive and packet process programs. All the functionality was included in a single source file, *tx_engine.c*.

File	Description	Lines
tx_engine.c	Main Program	286
	Total	286

D.3 Support Programs

The support programs required for development and testing of the project were written in the Python programming language.

AES Table Generation

A program was written to calculate the lookup tables required by the fast AES implementation. The method for calculating these lookup tables is detailed in [50]. This program calculates the tables and exports them in the form of a C header file for convenient inclusion in Microengine C code.

Log Processor

The logs produced by the Developer Workbench simulator are not conveniently formatted for the collection of packet processing statistics. The log processor program takes the receive and transmit log files produced by the simulator, pairs the received and transmitted packets and output the data in a comma separated value text file.

The .csv files produced by this program can be imported into a spreadsheet program (such as OpenOffice Calc) or MATLAB which can then be used to analyse and graph the data.

TCPDump Processor

Packet size statistics were collected by running the *tcpdump*[1] program on a number of workstations. A processing program was written to analyse these logs and output packet size statistics and data files for import into a spreadsheet or MATLAB .

D.4 Conclusion

The implementation, as completed, consists of 1770 lines of Microengine C code. In the process of comparing alternative designs, a considerably larger number of lines of code were written, tested and discarded.

[1] tcpdump is an open-source packet capture program which can be found at *http://www.tcpdump.org/*

Program	**Lines of Code**	**Percentage**
Receive	581	33%
Process	903	51%
Transmit	286	16%
Total	1770	

The complete source code for the implementation can be found on the included CD (see Appendix E) along with the code of all the support and analysis programs, reference implementations of AES and SHA-1 and other code.

Appendix E

Project CD

E.1 Directory Structure

CD Root	
├─ Programs	executables
│ ├─ Gateway	IPsec Gateway Microengine C code
│ ├─ Support	Support Programs
│ └─ Reference	Reference code for AES and SHA
├─ Literature	Selected Papers and Resources
│ ├─ AES and SHA	The AES and SHA Algorithms
│ ├─ IETF RFCs	RFCs and Standards from the IETF
│ ├─ IPsec	IPsec related papers
│ ├─ IXP2400	Papers related to network processors
│ └─ VPNs	Papers related to VPN protocols
├─ Report	The Project Report
│ ├─ Sources	Project report LaTeX and bibTeX source
│ └─ Diagrams	Diagrams included in the report
└─ Simulations	Simulation Data

Tools Used

The following tools were used for the completion of the project:

- Intel Developer Workbench 3.51 and 4.0
- Microsoft Visual C++ 6.0
- GNU Emacs
- The GNU Compiler Collection

The following software and tools were used for preparation of the report

- Kile, KDE Integrated Latex Environment 1.8.1
- GNU Emacs 21
- LaTeX 2_ε and pdfLaTeX
- Mathworks MATLAB R14
- GNU Image Manipulation Program 2.2
- Python 2.3
- Dia 0.94

Bibliography

[1] C. Falk, "The Ethics of Cryptography," Master's thesis, Purdue University, May 2005.

[2] Wikipedia, "Categorical imperative — wikipedia, the free encyclopedia," 2005. [Online; accessed 4 October 2005].

[3] D. Ermann, M. Williams, and C. Gutierrez, *Computers, Ethics and Society*. Oxford University Press, first ed., 1990.

[4] C. Xenakis and L. Merakos, "IPsec-based end-to-end VPN deployment over UMTS," *Elsevier Computer Communications*, May 2004.

[5] A. Godber and P. Dasgupta, "Secure wireless gateway." Arizona State University, 2002.

[6] A. Keromytis, J. Ioannidis, and J. Smith, "Implementing ipsec." Aug. 1997.

[7] "Microsoft windows 2000 internet protocol security (ipsec) review," tech. rep., Network Associates, Inc., Oct. 2003.

[8] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application specific dataflow architectures," in *Application-Specific Systems, Architectures and Processors, 1997*, July 1997.

[9] S. Rajagopal, J. Cavallaro, and S. Rixner, "Design space exploration for real-time embedded stream processors," *IEEE Micro*, August 2004. Accepted.

[10] "Ipsec forwarding application level benchmark implementation agreement," tech. rep., Network Processing Forum, July 2003.

[11] A. Tanenbaum, *Structured Computer Organization*. Prentice Hall, fourth ed., 1999.

[12] R. Yavatkar, *Network Processors: Building Block for Programmable Networks*. Intel.

[13] R. Szymanek, *Constraint-Driven Design Space Exploration for Memory Dominated Embedded Systems*. PhD thesis, Lund University, July 2004.

[14] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw Hill, sixth ed., 2004.

[15] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, Oct. 1974.

[16] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)." RFC 2406 (Proposed Standard), Nov. 1998.

[17] R. Glenn and S. Kent, "The NULL Encryption Algorithm and Its Use With IPsec." RFC 2410 (Proposed Standard), Nov. 1998.

[18] C. Madson and N. Doraswamy, "The ESP DES-CBC Cipher Algorithm With Explicit IV." RFC 2405 (Proposed Standard), Nov. 1998.

[19] S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec." RFC 3602 (Proposed Standard), Sept. 2003.

[20] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol." RFC 2401 (Proposed Standard), Nov. 1998. Updated by RFC 3168.

[21] N. I. of Standards and Technology, "Secure Hash Standard (FIPS-180-2)," Aug. 2002.

[22] Intel, *Intel IXP2400 Network Processor Development Tools User's Guide*, Mar. 2004.

[23] S. Baset and H. Schulzrinne, "An analysis of the skype peer-to-peer internet telephony protocol," Sept. 2004.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications." RFC 3550 (Standard), July 2003.

[25] Intel, *IXP2400 Network Processor Hardware Reference Manual*, Nov. 2003.

[26] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460 (Draft Standard), Dec. 1998.

[27] S. Kent and R. Atkinson, "IP Authentication Header." RFC 2402 (Proposed Standard), Nov. 1998.

[28] D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)." RFC 2409 (Proposed Standard), Nov. 1998. Updated by RFC 4109.

[29] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)." RFC 2408 (Proposed Standard), Nov. 1998.

[30] N. Ferguson and B. Schneier, "A Cryptographic Evaluation of IPSec." Counterpane Internet Security Systems, Inc, Feb. 1999.

[31] S. M. Bellovin, "Problem Areas for IP Security Protocols," in *Proceedings of the Sixth Usenix Unix Security Symposium*, July 1996.

[32] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn, "Point-to-Point Tunneling Protocol." RFC 2637 (Informational), July 1999.

[33] S. Hanks, T. Li, D. Farinacci, and P. Traina, "Generic Routing Encapsulation (GRE)." RFC 1701 (Informational), Oct. 1994.

[34] S. Hanks, T. Li, D. Farinacci, and P. Traina, "Generic Routing Encapsulation over IPv4 networks." RFC 1702 (Informational), Oct. 1994.

[35] B. Schneier and Mudge, "Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP)." Counterpane Internet Security Systems, Inc, 1998.

[36] B. Schneier, D. Wagner, and Mudge, "Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2)." Counterpane Internet Security Systems, Inc, Oct. 1999.

[37] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter, "Layer Two Tunneling Protocol "L2TP"." RFC 2661 (Proposed Standard), Aug. 1999.

[38] W. Simpson, "The Point-to-Point Protocol (PPP)." RFC 1661 (Standard), July 1994. Updated by RFC 2153.

[39] B. Patel, B. Aboba, W. Dixon, G. Zorn, and S. Booth, "Securing L2TP using IPsec." RFC 3193 (Proposed Standard), Nov. 2001.

[40] T. Dierks and C. Allen, "The TLS Protocol Version 1.0." RFC 2246 (Proposed Standard), Jan. 1999. Updated by RFC 3546.

[41] A. Alshamsi and T. Saito, "A technical comparison of IPsec and SSL." 2004.

[42] E. Rescorla, "Diffie-Hellman Key Agreement Method." RFC 2631 (Proposed Standard), June 1999.

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

[44] V. Klima, "Finding MD5 Collisions - a Toy For a Notebook." Cryptology ePrint Archive, Report 2005/075, 2005.

[45] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication." RFC 2104 (Informational), Feb. 1997.

[46] C. Madson and R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH." RFC 2404 (Proposed Standard), Nov. 1998.

[47] D. Eastlake 3rd and P. Jones, "US Secure Hash Algorithm 1 (SHA1)." RFC 3174 (Informational), Sept. 2001.

[48] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1." Aug. 2005.

[49] N. I. of Standards and Technology, "Advanced Encryption Standard (FIPS-197)," Nov. 2001.

[50] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[51] M. Dworkin, "Recommendation for block cipher modes of operation," in *NIST Special Publications 800*, vol. SP800, National Institute of Standards and Technology, 2001.

[52] Wikipedia, "Cipher block chaining — wikipedia, the free encyclopedia," 2005. [Online; accessed 10 October 2005].

[53] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code*. Wiley, Oct. 1995.

[54] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting, "Improved cryptanalysis of rijndael." Counterpane Internet Security Systems, Inc, 2000.

[55] L. Jing-mei, W. Bao-dian, C. Xiang-gao, and W. Xin-mei, "Cryptanalysis of Rijndael S-Box and Improvement." July 2005.

[56] CNSS, "National policy on the use of the Advanced Encryption Standard (AES) to protect National Security Systems and National Security Information," June 2003.

[57] Intel, *Intel IXP2400/IXP2800 Network Processors Microengine C Language Support Reference Manual*, Nov. 2003.

[58] Intel, *Intel IXP2400 Network Processor Programmer's Reference Manual*, Nov. 2003.