1-1-2007

Implementing IPsec using the Five-layer security framework and FPGAs.

James Wiebe
University of Windsor

Implementing IPsec using the Five-Layer Security Framework and FPGAs

by

James Wiebe

A Thesis
Submitted to the Faculty of Graduate Studies
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

2007

Your file Votre référence
ISBN: 978-0-494-35010-2
Our file Notre référence
ISBN: 978-0-494-35010-2

ABSTRACT

A VHDL implementation of 128-bit AES on a Xilinx Virtex-4 FPGA (lowest speed grade) and ML403 development board is developed from a Verilog design that adheres to the FIPS-197 standard, adding innovative features: automatic start of transform, CBC mode, key permutation value readout and store, and output of each intermediate state value. Core processing rate achieves 640 Mbps; 27 Mbps is achieved in practice, via peripheral register access. A non-linear, cryptographically secure LFSR-CASR pseudo-random number generator with a cycle length of 2^{80}-2^{43}-2^{37}+1 is translated into C and C++ from Verilog and evaluated. A C design and implementation of IPsec, based on the Five-layer security framework, using these primitives, is presented. The rate of IPsec packet processing achieved is 2 Mbps, determined by direct pulse measurement. A PC-based GUI drives the IPsec implementation and serves it policies, with a framework for flexibly choosing services, mechanisms and primitives using the SMIB.

Index Terms: IPsec, Virtex-4, FPGA, AES, pseudo-random number generator, Software Design, Cryptography

DEDICATION

To my mother, for a staggering amount of love, that is so great, that it is as difficult to comprehend as the most involved scientific theory.

ACKNOWLEDGEMENTS

I would like to thank the members of my thesis committee, Dr. Shervin Erfani, Dr. Huapeng Wu, and Dr. Arunita Jaekel, for their support, advice and encouragement – and patience in arranging the times of my seminars! My advisor, Dr. Erfani, also provided funding and printed and read all of my published and presented work done during the time of my Master's project work; as well, he helped me burn my seminar CDs on a last-minute basis. Dr. Wu also kindly printed a copy of my thesis for me.

I gratefully acknowledge the assistance of the department technologists, Mr. Frank Cicchello and Mr. Don Tersigni, for support in ordering and providing equipment and tools, and setting up presentations. The University of Windsor ECE (Electrical and Computer Engineering) department secretary, Andria Turner, was absolutely wonderful in chairing my defence, in providing other support during the time of my defence that I needed due to an extensive power failure that occurred that day, and in providing much other support during the time of my Master's work. Also, Dr. Roberto Muscedere provided equipment. Liviu Danaila, and George Granata, the FAEs (Field Applications Engineers) employed by the ML403 development board vendor, Nu Horizons, were invaluable. Liviu was the local FAE, and helped me considerably. Jennie, the Nu Horizons sales representative, made sure that I received the ML403 board by keeping track of the order. The following Xilinx technical support personnel were helpful with "webcases": Jonney Zhao, James Broadhead, Enda Behan, Yolanda Xu, Ricky Su and Zhaojin ("Michael") Ye. "KJ" on the comp.arch.fpga Usenet "newsgroup" helped me set simulation timing values so that the "post-map" simulation of my AES encryptor succeeded.

Some of my initial training was done on the Xilinx "Microblaze" boards using Xilinx training "lab" exercises from Xilinx and also as hosted by Dr. Paul Chow on his website at the University of Toronto [UTXILT]. The various tutorial documents provided by Xilinx were very useful – see the "Books, General Papers and other Resources" sub-section in the References section: the reference codes beginning with "XIL", particularly the "EDK 8.2 PowerPC Tutorial in Virtex-4" [XILML403T].

Thanks are due to my fellow students, Nima Bayan, Fang Chen, Amir Yazdanshenas, Raymond Lee, Kevin Banović, Ian Anderson, Mohammed Tarique and Wenying Zheng, for technical assistance and advice and encouragement. Dr. Mohammed Khalid also provided advice and support, and his course, "Reconfigurable Computing," provided some useful background. Dr. Xiang Chen provided advice and encouragement that helped lead me to pursue this area of study.

This work is largely based on the Five-Layer framework for designing security systems, patented by Dr. S. Erfani. The AES implementation presented is based on the design by Rudolf Usselmann on the "Open Cores" website [USS2002], [OPENCORES]. The module hierarchy figure, used in section 3.1.7., "AES Design Done in this Work," was modified from Figures 6 and 7 in [USS2002]. The pseudo-random number generator implementation presented is a translation from the Verilog design by Javier Villar on the "Open Cores" website [VILL2005]. This material found on the "Open Cores" website is in the public domain. The serial communication package used with MSVC++ V 6.0 (Microsoft Visual C++ Version 6.0) is from "The Code Project" website, is by Ramon de Klein, and is used under the terms of the LGPL (Lesser GNU Public License) [KLE2003].

The "E-Business Security" figure used in section 2.3.1., "Other Management Proposals", is redrawn from [TRC2003] with permission from Elsevier. The "C-ISCAP" figure used in section 2.3.1. is reproduced (redrawn) with kind permission of Springer Science and Business Media ([PAR2002], Figure 1, pg. 383, © Springer-Verlag Berlin Heidelberg 2002). The figure illustrating the Xilinx system architecture in section 3.1.4., "Architecture Provided by Xilinx" ([XILUT2003] pg 21), was published by Xilinx in order that it "may be used in any form that would benefit the professor and students," [XILTMAT]. The figures illustrating AES in section 3.1.1.1., "Overview of AES", are from [FIPS197] and are in the public domain.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AB.1. Acronyms

AB.1.1. General

AKA – Also Known As
BC – British Columbia – the Western-most Canadian province.
DST – Daylight Saving Time
DUANWKWYM – Don't Use Acronyms; Nobody Will Know What You Mean!
IBM – International Business Machines: a very large and famous company
ID – Identity
NA – Not Applicable
PC – Personal Computer
SW – SoftWare
US – United States
USA – United States of America

AB.1.2. Technical

3DES – Triple-DES – see DES
ACK – Acknowledgement: a non-printing ASCII control character
ACM – Association for Computing Machinery
AES – Advanced Encryption Standard – Rijndael, as of Fall 2000.
AH – Authentication Header – authenticating the packet contents in IPsec
ANSI: American National Standards Institute
ASCII – ANSI Standard Code for Information Interchange
ASIC – Application-Specific Integrated Circuit
b – bit(s) – single binary digit(s)
B – Byte(s) – fundamentally defined as a group of eight bits, i.e., an "octet" of bits. Also, the "B" programming language, which stands for Bell labs, where it was invented.
BDM – Background Debug Mode – see PCIV
BE – Big-Endian (see LE)
BIOS – Basic Input-Output System
BITS – Bump-In-The-Stack – the addition of processing by insertion and integration into existing layered protocols. Compare BITW.
BITW – Bump-In-The-Wire – the addition of processing devices placed downstream in the dataflow from the originator. Compare BITS.
BRAM – Block RAM
bps – bits per second. Compare Bps. Usually about ten times the latter due to parity and other bits included with every byte.
Bps – Bytes per second. Compare bps.
BSB – Base System Builder – a "wizard" in Xilinx EDK
BSD – Berkeley Software Distribution – various versions of open-source Unix.
C – a programming language; successor to B

C++ – an object-oriented programming language; successor to C
CA – Certificate Authority
CASR – Cellular Automata Shift Register – see LFSR
CBC – Cipher-Block Chaining – a cipher feedback mode in which the output of the previous encrypted block is XORed with the next block of plaintext before that is encrypted. See ECB.
C-ISCAP – Controlled Internet Secure Connectivity Assurance Platform – author-specific; see [PAR2002].
CLI – Command-Line Interface – compare GUI
CMOS – Complementary Metal-Oxide Semiconductor
COPS-PR – Common Object Policy Service for PRovisioning – RFC 3084
CPLD – Complex Programmable Logic Device
CPU – Central Processing Unit
CSPRNG – Cryptographically-Secure PRNG
DCM – Digital Clock Manager
DH – Diffie-Hellman
dDoS – Distributed DoS
DES – Pronounced "Dez" (short "e") – (The) Data Encryption Standard, US National Bureau of Standards, FIPS Publication 46, January 1977
DMA – Direct Memory Access
DNS – Domain Name Service – the protocol used on the Internet, for the WWW, for translation between IP addresses and website names.
DOI – Domain Of Interpretation
DoS – Denial of Service
DOS – Disk Operating System (owned and developed by Microsoft, Inc.)
DSP – Digital Signal Processing
E – Electronic, as in "E-business," "email," etc.
ECB – Electronic Code Book – a block cipher mode that uses no feedback. Each block is encrypted independently of the others. Pipelining or parallel encryption is possible, but is cryptographically, weaker, since identical blocks of plaintext will result in identical blocks of ciphertext. See CBC.
ECC – Elliptic Curve Cryptography
ECE – Electrical and Computer Engineering – still a fairly new amalgamation as of the 2000s
EDK – Embedded Development Kit – Xilinx software – see ISE
EEPROM – Electrically-Erasable PROM – often pronounced "E-squared PROM"
ESP – Encapsulating Security Protocol – encrypting the packet contents in IPsec
ETX – End-of-Text: a non-printing ASCII control character
FAE –Field Applications Engineer
FIPS – Federal Information Processing Standard – a US (United States) establishment
FPGA – Field-Programmable Gate Array
FSL – Fast Simplex Link
FTP – File Transfer Protocol
GF – Galois Field
GDB – GNU DeBug – an open-source debugger for embedded systems
GNU – GNU is Not Unix – A Unix-like operating system and collection of programs

GPIO – General Purpose – or Parallel – IO
GPL –GNU Public License
GUI – Graphical User Interface – compare CLI
HAS-160 – a cryptographic hash function designed for use with the Korean KCDSA digital signature algorithm [WIKIP].
HDL – Hardware Design Language, such as VHDL or Verilog
HLS – High-Level Synthesis
HMAC – Hashed Message Authentication Code
HO – High-Order – see LO
HOB – High-Order Byte
HTTP –HyperText Transfer Protocol
IC – Integrated Circuit
IDE – Integrated Development Environment
IEEE – Institute of Electrical and Electronics Engineers
IETF – Internet Engineering Task Force – see References – Websites.
IIC – Inter-IC bus
IKE – Internet Key Exchange
IO – Input-Output
IP – Internet Protocol; also Intellectual Property (used to refer to HDL implementations)
IPIC – IP InterConnect, Xilinx – IP: Intellectual Property
IPIF – IP InterFace, Xilinx – IP: Intellectual Property
IPsec, IPSec – IP Security
IPSP – Internet Protocol Security Policy
IPSPE – unknown: see IPSP, and [KEN1994]; possibly IPSP Extended.
IPv4 – IP version 4 – see IP
IS – Information Systems
ISAKMP – Internet Security Association and Key Management Protocol [STA2003], pg. 202
ISE – Integrated Software Environment (Xilinx software – see EDK); also Internet Security Evaluation system [PAR2002].
IV – Initial (or Initialization) Vector
JTAG – Joint Test Action Group, IEEE standard 1149.1
L – Layer, Laptop
LAN – Local Area Network
LCD – Liquid Crystal Display
LE – Little-Endian (see BE)
LED – Light-Emitting Diode
LFSR – Linear Feedback Shift Register – see CASR
LGPL – Lesser GNU Public License
LO – Low-Order – see HO
LSI – Low Scale Integration (or Large Scale Integration) – see MSI and VLSI
LUT – Look-Up Table
MAC – Message Authentication Code – see HMAC
MD5 – Message Digest algorithm 5 - one of a series of message digest algorithms designed by Professor Ronald Rivest of MIT.
MIB – Management Information Base

MITM – Man In The Middle, or Monkey (saboteur) In The Middle
MS – Microsoft (company) – see MSVC++V6
MSI – Medium Scale Integration – see LSI and VLSI
MSVC++V6 – MS Visual C++ Version 6.0
NAK – Negative Acknowledgement: a non-printing ASCII control character
NDP – Neighbour Discovery Protocol
NGM – New Group Mode, an IPsec key exchange utility mode.
NRE – Non-Recurring Engineering (costs)
OP – OutPut
OPB – On-Chip Peripheral Bus, Xilinx
OS – Operating System
OSI – Open Systems Interconnection
P – PC: Personal Computer, often intended to mean an "IBM" (now generic) PC.
PAR – Place And Route
PCB – Printed-Circuit Board
PCIV – the Xilinx "Parallel Cable IV" (pronounced "PC-four") cable. (Used for BDM)
PDA – Personal Digital Assistant
PGP – Pretty Good Privacy, a software package that provides confidentiality and authentication at the application layer [PGPI].
PIB – Policy Information Base – RFC 3159.
PLB – Processor Local Bus, Xilinx
PPC – Power PC, i.e., "Power Personal Computer", a microprocessor that can perform as Intel x86 as well as Motorola 68x CPUs.
PRNG – Pseudo-RNG – see CSPRNG
PROM – Programmable ROM – see EEPROM
PSTN – Public Switched Telephone Network
PU – frequency of occurrence Per Unit (per each one), as "percent" is per each hundred.
QoS – Quality of Service. Types of: "Hard": specific numerical guarantees are made and kept. "Soft": higher-priority data flows are given higher-priority access to the system.
RACE – Research and development in Advanced Communications technologies in Europe. Used in RIPEMD-160 (qv).
RADAR – RAdio Detection And Ranging
RAM – Random-Access Memory – see ROM, BRAM, SRAM
RC – ReConfigurable (hardware) such as FPGAs and CPLDs.
RFC – Request For Comment: an IETF document that can be informational, a proposed standard, or an IETF standard.
RIPEMD-160 – RACE (qv) Integrity Primitives Evaluation Message Digest – a 160-bit message digest algorithm (and cryptographic hash function) developed in Europe. [WIKIP]
RNG – Random-Number Generator – see PRNG, CSPRNG
ROM – Read-Only Memory – see PROM, EEPROM, RAM
RTP – Real-time Transport Protocol: i.e., a Transport Protocol designed for real-time communications such as streaming multimedia, or possibly telepresence.
S – Substitution, as in S-Box
SA – Security Association
SADB – Security Association Database

SHA-1 – Secure Hash Algorithm 1 – successor to MD5.
SLA – Service Level Agreement
SMIB – Security Management Information Database
SPI – Security Policy Index – uniquely identifies an SA
SPD or SPDB – Security Policy Database
SRAM – Static RAM
STX – Start-of-Text: a non-printing ASCII control character
SWAN – Secure WAN – see WAN
TCP – Transmission Control Protocol – the main Transport-level protocol used on the Internet, providing a guarantee of reliable delivery and correct packet ordering.
TTL – Time To Live – field of an Internet Protocol (IP) datagram
UART – Universal Asynchronous Receiver-Transmitter – an integrated device that can perform such communication protocols as RS232
UDP – User Datagram Protocol – a simple Transport-level protocol used on the Internet
UML – The Unified Modeling Language
VHDL – VHSIC HDL
VHSIC – Very High-Speed Integrated Circuit – see VHDL
VLIW – Very Long Instruction Word
VLSI – Very Large Scale Integration (see MSI and LSI)
VPN – Virtual Private Network
XMD – Xilinx Microprocessor Debug
XOR – Exclusive-OR – a common operation in cryptography, since it is its own inverse
WAN – Wide-Area Network
WEP – a standard for protecting 802.1X communications, "Wired Equivalent Privacy" – so-called; it is not really very strong.
WWW – The World-Wide Web, AKA, in some circles, as "The World-Wide Wait".
XST – Xilinx Synthesis Tool

AB.2. Abbreviations

AB.2.1. General

approx. – approximately
demo. – demonstration
dept. - department
Fig. – Figure, i.e., illustration.
hun. – hundred
inc. – incoming
out. – outgoing
rev. – revision (often synonymous with "ver.", but sometimes used to denote sub-version levels)
sub. – substitute, or substitution; "sub" is also a complete word or prefix meaning "below", "under" or "smaller".
thou. – thousand
v., ver. – version (often synonymous with "rev.")

AB.2.2. Technical

app.	– application
dec.	- decryption
enc.	– encryption
Flash	– Flash EEPROM
hex.	– hexadecimal
Hz	– Hertz – cycles, or any repetitive occurrence, per second. The repetitive occurrence is simply a dimensionless count, so this unit has a dimension only of inverse time.
Rcon	– Round constant
slv_regs	– slave registers, of a component in an embedded system
std_logic	– standard logic

Chapter I

INTRODUCTION

1.1. Motivation

To begin with the virtually self-evident, companies need communications. In any economy beyond that of the 1700s (in the USA), electronic communications are needed. Indeed, these are valuable to individuals for personal use, as well. Telegraph, telephone, telex and facsimile systems performed all the services of electronic communication systems in their day, for decades, since the 1800s, and it was not until the 1990s that the Internet began to rise to predominance, replacing and supplementing those systems with the mass ability to transmit documents. The facsimile machine has been largely rendered redundant by email and the WWW (World Wide Web), and telephony is in the process of becoming an Internet application at the time of this writing, although it is not now known, of course, if the entire legacy PSTN (Public Switched Telephone Network) will be replaced by the Internet and if so, how long that will require.

Partly due to its public nature, and partly due to economies of scale, the Internet is extremely economical to use, which gives a reason for companies and individuals to make great use of it. However, since it is public, it is necessary to secure its use for general privacy purposes. Today, generally only companies have the resources to perform mass securing of Internet services, although personal software packages such as PGP (Pretty Good Privacy) are available for individual use – PGP performs security services at the user, or application layer [PGPI]. For mass use, companies use encryption to implement VPNs (Virtual Private Networks), using the Internet as their own private communication network. VPNs are technically available to individuals, since Microsoft Windows XP, for one, is equipped to perform IPsec (Internet Protocol security), but its setup still requires technical expertise and cooperation that are mostly beyond the abilities of unorganized individuals. Notably, the "Free S/WAN" movement and software package has attempted to provide IPsec to individuals; so far without success [FSWAN].

Use of a public network gives some of the strongest possible reasons to adopt security measures against attacks that threaten, although managers of private networks are well advised to adopt additional security measures beyond the physical. Some attacks and their countermeasures are as follows:

- Masquerade, unauthorized access, repudiation: need an authentication/access control service
- Message modification, replay: need an integrity service
- Spying: need a confidentiality service
- Denial of service (DoS)/unauthorized access: need an availability/access control service

These imply the existence of five basic services: authentication, integrity, confidentiality, access control and availability. Two others are anti-replay (a form of integrity, given that the time a message is sent should be counted as part of its makeup), and non-repudiation (which consists of authentication plus audit logging or message retention).

The combination of "IP" (Internet Protocol) and "Security" creates the abbreviation "IPsec". Although it was originally intended to secure all IP transactions, it found its most natural application in VPNs [FER1999]. IPsec can be thought of as an adaptation of the general concept of VPNs to the Internet [PER2000].

1.2. Overview of IPsec

The idea of IPsec dates back to 1994, and the most important four RFCs (Requests For Comment) were issued in 1998 [STA2003]. They are: RFC 2401, "Security Architecture for the Internet Protocol", RFC 2402, "IP Authentication Header", RFC 2406, "IP Encapsulating Security Payload (ESP)", and RFC 2408, "Internet Security Association and Key Management Protocol (ISAKMP)" – see the "RFCs" subsection in the References section – a full list of IPsec RFCs is available [IPS2005].

A brief overview of IPsec is presented here. A comprehensive and detailed description of IPsec has already been done in the working group to which this author belongs, in the ECE department at the University of Windsor, by a previous Master's candidate; please see [FAH2005].

It is desirable to make the security system transparent to the user, for if the user has to perform any operation to accomplish the security transform in addition to sending the message, the additional burden will likely be refused, done carelessly, or, with the best will in the world, absent-mindedly forgotten in the press of competing primary duties. The IPsec layer or sublayer is located below the IP layer, just above the link layer (see Figure 1 for the OSI model adapted to five layers for the Internet – OSI stands for "Open Systems Interconnection"), and thus is transparent to the user, who generally deals directly with only the Application layer.

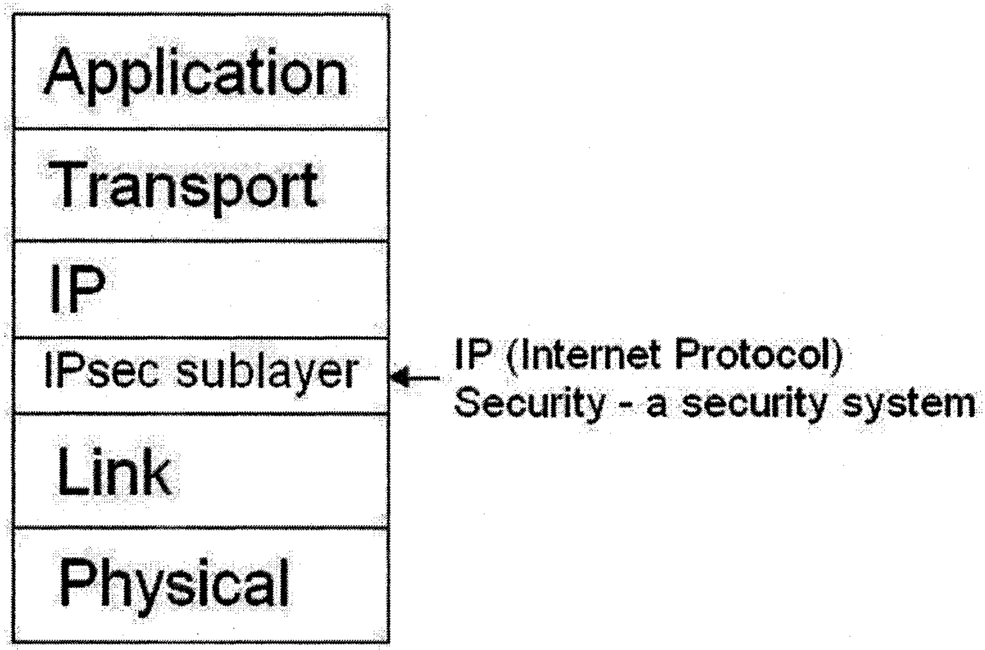


Figure 1. The OSI model adapted to five layers for the Internet – also showing IPsec

IPsec provides confidentiality, integrity, authentication, anti-replay services, and can be used for non-repudiation and access control, although common implementations of those two are typically weak (see the final paragraph in section 2.2.1. "Key Exchange"). It does not provide a formal availability service, and suffers from weakness in this area (see section 2.2.1., "Key Exchange").

1.2.1. Key Exchange

There are three different modes of key exchange in IPsec: *Main Mode*, *Aggressive Mode*, and *Quick Mode*. There are also some utility modes, such as NGM (New Group Mode), used to negotiate a new group for Diffie-Hellman Key Exchange, carried out under protection of ISAKMP (Internet Security Association and Key Management Protocol) phase 1. Main Mode and Aggressive Mode are alternate modes that can be used to establish the "Phase 1" SAs (Security Associations), whereas Quick Mode is an exchange that uses the protection of the Phase 1 SAs to establish the Phase 2 SAs that are the actual IPsec working SAs that protect the data packets ([ZHO2000] pg. 1606).

1.2.1.1. Main Mode

The initiator sends a cookie and a proposed phase 1 SA. The responder replies with a cookie and the accepted phase 1 SA. The initiator sends its Diffie-Hellman public key and a nonce, and the responder replies with its Diffie-Hellman public key and nonce. Both sides compute the Diffie-Hellman shared secret, or key. The initiator sends its signed certificate to establish its identity and the responder replies with its signed certificate ([AIE2002] Figure 3, pg. 55). Signing in this case means to encrypt a hash of the message using the shared secret – in general, signing means encryption, using a shared secret, or a private key in a public key cryptosystem, of the message itself or of a hash of the message. A total of six messages are sent.

1.2.1.2. Aggressive Mode

In Aggressive Mode, only three messages are exchanged. The initiator cookie, proposed Phase 1 SA, Diffie-Hellman public key and initiator ID (identity) are sent at once, and the the responder replies with its own cookie, the accepted Phase 1 SA, its Diffie-Hellman public key , its ID, as well as its signed certificate. The initiator then sends its signed certificate ([AIE2002] Figure 4, pg. 56).

1.2.1.3. Quick Mode

See Figure 2. The initator sends its proposed Phase 2 SA, nonce, hash, optional DH public key, and optionally, client IDs. The responder replies with the accepted Phase 2 SA, nonce, hash, optional DH public key, and optionally, client IDs. As a handshake, the initiator sends a hash of the nonces ([ZHO2000] pg. 1608).

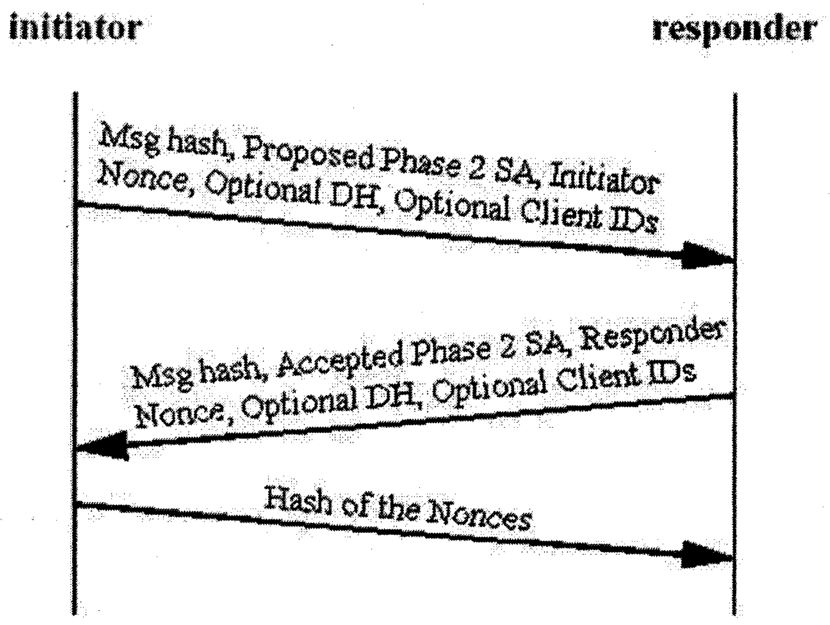

initiator
responder
Msg hash, Proposed Phase 2 SA, Initiator Nonce, Optional DH, Optional Client IDs
Msg hash, Accepted Phase 2 SA, Responder Nonce, Optional DH, Optional Client IDs
Hash of the Nonces

Figure 2. Quick mode

1.2.1.4. Key Exchange – Conclusions

It can be seen that these protocols are susceptible to a DoS (Denial of Service) attack. Since there is no burden of identification or computation placed upon the initiator, attackers can make the server perform computationally-expensive modular exponentiations in order to calculate the Diffie-Hellman shared secret. See section 2.2.1., “Key Exchange”, for a full discussion.

1.2.2. Security Policy Database and SMIB

An IPsec implementation requires an SPD (Security Policy Database) for the purpose of negotiating security associations. The SPD contains, in part, selectors to determine whether or not to process a packet, for a given policy of how to process the packet selected. The selectors can be set to “all”, as “wildcards”, and the sense of the selectors can be set as meaning either to select or not select the packets with the selected characteristics. Using an SPI (Security Policy Index) to uniquely identify a Security Association (SA), an SPD entry can refer to more than one SA, and a single SA can be derived from more than one SPD entry, in which case more than one SPD entry would have the same SPI – see Table 1 for an illustration.

From	To	Protocol	Port	Policy	SPI(s)
1.1.1.1	2.2.2.2	TCP	1000	ESP w. 3DES	1,3
1.1.1.1	2.2.2.2	*	*	ESP w.AES	2

Table 1. Key elements of a Security Policy Database (SPD)

In addition, it can be useful to define a Security Management Information Database (SMIB), containing the SPD and other information useful to running a security and communication system, such as the local address, clients served by the IPsec operating entity, or node, the functional modules available to the system to do key management and perform the other services noted before, and the parameters needed to control them. See [KEN1994] for a list of ideas.

1.2.3. Security Associations

A security association (SA) defines the agreement under which two entities will use IPsec to communicate, in a particular direction; i.e., two SAs are needed for bidirectional communication. The SA contains, at a minimum, the addresses of the communicating entities, the protocol and mode to be used, the SPI, and the algorithms to be used – see Table 2 for an illustration of a Security Association Database (SADB), which contains the information specifying the node’s SAs.

From	To	Protocol	Mode	SPI	Policy
2.2.2.2	1.1.1.1	ESP	Tunnel	10	64-bit DES
1.1.1.1	2.2.2.2	ESP	Transport	11	168-bit DES

Table 2. Key elements of a Security Association Database

1.2.4. Services

1.2.4.1. Authentication Header Protocol

The Authentication Header (AH) Protocol adds a header to IP packets, that contains a signed hash of the message, to transform the packet into an IPsec packet. This provides authentication and integrity services. The header also contains the SPI so that the SA can be identified, a sequence number for anti-replay purposes, and some other data, such as the type of header immediately following, and the "payload" length of the Authentication Header.

1.2.4.2. Encapsulating Security Payload Protocol

In the Encapsulating Security Payload (ESP) Protocol, each IP packet's contents are encrypted, providing confidentiality and encryption. The contents are first padded to make them a natural number multiple of the encryption block size, and to provide space for the padding length field itself, meaning that if the contents are already a multiple of the block size, padding must still be added. A header is added that contains the SPI and sequence number, and a variable-length authentication field is specified, following the payload data. If an Initial Vector (IV) is included with the ciphertext, it is usually not encrypted ([STA2003] pg. 183); in this work it was realized that if Cipher-Block Chaining (CBC) were to be used, and the IV is first encrypted without chaining, following which the IV is used, an attacker would be able to tell if the first block of data happened to be all zeroes, because then the encrypted IV and the first encrypted block of data would be identical (note that for the strongest possible security, security algorithms are generally made thoroughly public in order to receive the most possible scrutiny). If chaining were to be done from the encrypted IV, the encrypted IV would itself be the IV, meaning that time would have been wasted in a needless transform done to the IV.

1.2.4.3. Anti-Replay Service

Each IPsec packet contains a 32-bit sequence number to prevent replay attacks. If a packet with an identical sequence number is received, it is discarded. Also, if a packet with a sequence number that is too old is received, it is discarded. Of course, if the packet authentication fails, the packet is discarded. The foregoing are events that should be logged for audit [STA2003]. A settable "window size" determines the lowest sequence number that will be accepted, from the highest sequence number so far received. When the sequence number overflows, the SA should be renegotiated, although whether that will be done is generally also negotiated.

1.2.5. Modes of Operation

Two modes of operation are defined for each of the AH and ESP protocol, *Transport Mode* and *Tunnel Mode*. In Transport Mode, the IP packet's header is modified as needed for retransmission and the IPsec header is inserted following. Any transform, such as ESP, is done only to the packet payload. In Tunnel Mode, a new IP header is appended to the beginning of the IP packet, following which the IPsec header is added, following which the entire IP packet is included, unchanged in AH protocol and transformed only, in ESP. This allows the original IP packet to continue on unchanged after its transmission as an IPsec packet, which is very useful when a gateway is employed, and for VPNs.

1.3. Previous Work

As noted after, (see section 2.3.2., "The Erfani Patent"), in previous work in the author's group in the ECE department at the University of Windsor ([FAH2005] section 4.2, pp. 84-94, and Chapter V), a five-layer framework for the design of a security system was introduced. The five layers are: the Policy, Management, Services, Mechanisms and Primitives layers (see section 2.3.2., after). These five layers were "fleshed out" into modules and several operating scenarios were described. These are: an IPsec session scenario, in which the security system is used to establish an SA and send a packet via ESP Transport mode ([FAH2005] section 5.1.1, pp. 100-105), a comparison between policies used for secure vs. very secure applications ([FAH2005] section 5.1.2, pg. 105), and a description of a combination of SAs in which IPsec AH protocol packets are tunnelled through an ESP SA between two routers ([FAH2005] section 5.1.3, pg. 106).

1.4. Problem Statement

Generally, the software approach to implementation is versatile, but resulting implementations are relatively slow. Use of "hardware", or dedicated integrated circuits (ICs) – or even LSI and MSI (Low Scale and Medium Scale Integration) ICs to perform a task results in implementations that work much faster, but versatility suffers.

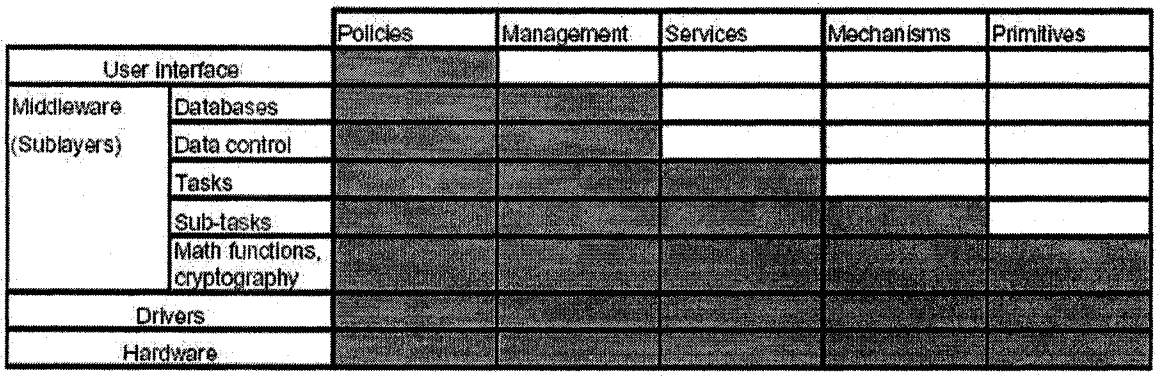
Cryptographic operations, such as encryption, decryption, hashing and random number generation are generally extremely computationally intensive, making hardware accelerators extremely desirable.

This leads to the question: How can the five-layer security architecture be used to implement IPSec in hardware, given that the overhead of using cryptography mandates hardware acceleration?

1.5. Motivation for General Layering

In implementation, breaking the task into implementation layers, from hardware, to software drivers, middleware and user-interface layers is useful to make it manageable and doable by a group of individuals or working groups, each of which does his own component. Modules can also be upgraded and replaced separately. Each layer uses the services of the layer below (except for the lowest layer – in perhaps a system-limited sense) to provide services to the layer above (except for the highest layer – again, in perhaps a narrow sense). This was done in the seven-layer OSI model which specifies the following layers, from top to bottom: Application, Presentation, Session, Transport, Network, Link and Physical. The seven were reduced to five for the Internet: Application, Transport, Network, Link and Physical. Note that in this modified OSI model, IPsec, if implemented via BITS (Bump-In-The-Stack), would fit at the *bottom* of the Network (Packet), or IP layer, since IPsec is applied after packetization just before sending to the Link layer – note that the diagrams in [FUM1998] (Figure 2, pg. 191), and in [HUN1998] (Figure 11, pg. 1118), depicting IPsec between the IP and TCP (Transmission Control Protocol) layers, are not right.

It is also useful to break the task down into conceptual, functional, or managerial levels, as in [ERF2003], which proposes five layers: Policy, Management, Services, Mechanisms and Primitives (Figure 3).



		Policies	Management	Services	Mechanisms	Primitives
User interface		▨				
Middleware (Sublayers)	Databases	▨	▨			
	Data control	▨	▨			
	Tasks	▨	▨	▨		
	Sub-tasks	▨	▨	▨	▨	
	Math functions, cryptography	▨	▨	▨	▨	▨
Drivers		▨	▨	▨	▨	▨
Hardware		▨	▨	▨	▨	▨

Figure 3. Functional vs. technical layers

As indicated in Figure 3, in a relation proposed in this work, a given functional layer requires presence at its and all lower implementation layers; the lower implementation levels have to contain "sub" functions to support the higher functionality. Policies require entering from the user interface, presence in databases, and control of the data. Management may require some entry from the user interface as well, but it at least is specified by the data entered into the databases. Services and Mechanisms are tasks and subtasks, and primitives require the low-level math functions and cryptography. Finally, in order for the electronic communications to proceed, the software must operate the hardware via drivers.

It might theoretically be possible to be more efficient with an "ad hoc" unlayered design, but a sufficiently complex system would not then be understandable and improvements or bug fixes would eventually become impossible, at some level of complexity. Disadvantages of layering include the necessity of passing data down through the multiple layers, which can slow down processing. Strict separation of layers can prevent a successful system design if passing needed data is disallowed on the grounds that it appears to "belong" only to one certain layer. A layer may duplicate functionality present in another layer, perhaps due to insufficient communication and planning between the respective working groups. An example of this latter inefficiency is error checking and recovery, which is often done at the link layer as well as on "an end-to-end basis" ([KUR2000], 3rd ed., pg. 47).

1.6. Motivation to use FPGAs

FPGAs (Field-Programmable Gate Arrays) contain thousands of blocks of identical generic logic which can be configured via programming like a static RAM (Random Access Memory) to operate in an extremely wide range of different behaviours. They are cost-effective in production and testing since this can be done in-house with affordable equipment, and devices are provided by manufacturers that make available many resources (block RAM, clock dividers, etc.) in a structured way.

FPGAs offer some of the performance levels of hardware and also some of the versatility of software, since they can be reconfigured for different functionality, even during runtime. Configuration files, or "bit files", for programming the FPGA, can be stored in memory, such as Flash EEPROM (Electrically-Erasable Programmable Read-Only Memory), or "Flash", and recalled at will. Stored configurations, such as encryption schemes, can be compressed for storage [DAN2000]. Field upgrades for such things as bug fixes and new standards are possible, even using pin-compatible devices [CHE2002].

FPGAs offer lower non-recurring engineering (NRE) costs compared to custom or semi-custom ICs, such as ASICs (Application-Specific ICs), since they come ready-made, and need only be configured. On the other hand, they incur higher per-unit (PU) costs, such that if sales of more than 100,000 units occur, it would be more economical to spend the NRE costs to produce an ASIC ([KHA2006] pg. 8, [OGA2004]).

Software running on a general-purpose processor can typically produce AES (Advanced Encryption Standard – Rijndael is the name of the specific algorithm adopted) throughputs of low tens of Mbps (i.e., 30) [DAN2000], whereas FPGAs can achieve up to 176 Mbps implementing DES (the Data Encryption Standard) and up to Gbps rates implementing AES; for example, speeds of 964 Mbps ([WOL2004] pp. 550, 554) and 1.197 Gbps [LUJ2005] have been reported (see section 2.4., "Implementations of IPsec," after). An ASIC processor achieved 2.29 Gbits/s of AES throughput in a 0.18μm CMOS (Complementary Metal-Oxide Semiconductor) standard-cell technology in 2002 [SCH2002].

1.7. Embedded Systems

An embedded system is a computerized module or component containing built-in software, usually on an IC chip, which is not changed in its normal course of operation. As part of its computerization, it also contains a computer processor, but that is not a defining characteristic, since non-embedded systems such as PCs (Personal Computers) also contain processors. For example, in this work, the normal operation of the system does not include loading and running the firmware, i.e., once the bit file, which contains both the FPGA configuration (the "soft hardware") and the ("firm") software, has been loaded to the board. Although this work involved loading the ML403 board on a regular basis, every single time it was used, this can be seen as engineering development work, not regular user operation, for which the bit file could be stored in the Flash EEPROM (Electrically-Erasable Programmable Read-Only Memory) and automatically loaded at bootup. The ML403 board can also be loaded by a user via a "flash card", in which case its status as an embedded system would be greatly mitigated. For another example, a PC is not an embedded system, because the user operates it by loading programs to its RAM IC chip or chips from its hard disk, such as by clicking with a "mouse", in its normal course of operation. Internally, however, a PC contains an embedded system, the BIOS (the Basic Input-Output System), which historically could not be very easily changed by the user at all (PCs first appeared in the early 1980s, and somewhat earlier, too, in a general sense, before the IBM PC appeared on the market). Today, the BIOS chip can be a Flash EEPROM, which can be changed by the user via a special procedure, not in its normal course of operation. Another example of an embedded system might be a PDA (Personal Digital Assistant), able to perform many applications. Another might be the anti-lock braking module in a car. Yet another might be a coffee maker, in which the software would most likely be present in a PROM, soldered directly to the PCB (Printed-Circuit Board), to keep manufacturing costs low. Another might be a washing machine, in which the PROM might conceivably be placed in a socket, for possible warranty repairs. Another might be a "set-top box", capable of having its software in its Flash EEPROM updated "over the air" by the service provider. Clearly, the more easily the software can be changed, and the more frequently it actually is, the less "embedded" the system is.

1.8. Objectives

This work has the following objectives: (1) to produce a block-level design of an IPSec processor, (2) implementing each layer of the Five-Layer paradigm, and (3) implement at least a key portion of it, using FPGA hardware accelerators; (4) to avoid pitfalls – note that the implementation of a security system can detract from its maximum theoretical strength as planned in corresponding standards – standards do not specify implementation details; and (5) to compare the performance results achieved to those of others.

1.9. Thesis Organization

This thesis is organized as follows: Chapter 2 presents a review of the literature pertaining to IPsec and its implementations. A brief history of and background to IPsec are presented and its applicability is discussed. IPsec key exchange, as a noteable point of weakness in IPsec, is treated. An overview of high-level management schemes for IPsec is presented, including the patent by S. Erfani, which is the background for this work. Software and hardware implementations of IPsec and its primitives are surveyed, the latter in FPGAs and ASICs, and an example application is presented. High Level Synthesis is discussed, as well as random number generators. Chapter 3 presents the design of an AES hardware accelerator in VHDL (Very High-Speed Integrated Circuits Hardware Design Language), as ported, or translated, from Verilog and implemented on a Xilinx Virtex-4 FPGA using the Xilinx ML403 development board, the design of test software for it, the design of a CSPRNG (Cryptographically-Secure Pseudo-Random Number Generator) in C (the programming language), as ported from Verilog, the design of an implementation of a portion of an IPsec implementation in C using the novel security design framework proposed by S. Erfani (see [ERF2003] and [FAH2005]), the design of two demonstration GUIs (Graphical User Interfaces) using MSVC++V6 (MicroSoft Visual C++ *ver. 6*), and the design of a CLI (Command-Line Interface) suitable for performance-testing of the IPsec implementation. The test methodologies are also presented. In Chapter 4, the results acquired from testing the AES implementation, the CSPRNG, and the IPsec implementation are presented and analyzed. Lastly, Chapter 5 presents conclusions and discusses areas for future work. In each of Chapters 3-5, section 1 contains the AES implementation discussion, section 2 contains the CSPRNG discussion and section 3 contains the IPsec implementation discussion.

CHAPTER II

REVIEW OF LITERATURE

2.1. Introduction

This chapter presents an overview of IPsec implementation and management in a variety of different areas: industry white papers, FPGA papers, papers on ASICs, papers on implementation of primitives such as AES and Random Number Generators (RNGs), papers on High-Level Synthesis (HLS), papers on IKE (Internet Key Exchange), and system-wide, or "high-level" papers. The Erfani patent [ERF2003], which is the paradigm for the present research, is presented. It is shown that the state of the art in the literature contemplates system-wide approaches to IPsec, but there is still room for improvement in terms of explicit recognition of all layers of an IPsec system for the purpose of managing its design and implementation (see the author's overview paper [WIE2006]).

2.1.1. History of IPsec

IPsec refers to the "Secure IP" set of proposals published by the IETF (the Internet Engineering Task Force) as RFCs [IETF]. The formal standards process in the IETF began in 1992 (compare 1994 as stated in [STA2003] before, in section 1.2.) with the publication of the first draft charter for the IPSEC working group [DUN2001], and as of April 29, 2005, there were 31 RFCs listed in the IPsec Charter [ITEF-IPSEC].

IPsec has now been in existence for so long that the pace of technological change has obsoleted part of it – the original, or "single" DES (Data Encryption Standard) specified only a 56-bit key and can now be broken by an exhaustive search attack in a few days using publicly-published techniques. The FreeS/WAN [FSWAN] organization has disallowed "56-bit" DES on the grounds that it is now too weak (even though that level of security would prevent real-time monitoring of transmissions and could allow the continuing accumulation of ciphertext faster than it could be cracked), which technically places them in violation of the standard. In Oct 2, 2000,The US National Bureau of Standards officially adopted one of the proposals, Rijndael, which was submitted in the competition to provide the Advanced Encryption Standard (AES), replacing DES [JAR2003], [REJ2003], but the movement to replace DES as the minimal encryption standard with 3DES did not succeed. However, DES expired as a standard in 1998 [ELB2000].

Another debate was over simplex vs. duplex data flows. Since data might need to be transmitted in only one direction, it was decided to base IPsec on simplex connections; hence Security Associations (SAs) are one-way [DUN2001].

2.1.2. Government Politics

The US government's reaction to new encryption technologies was one of the strongest: it classified cryptographic hardware and software as "munitions" and forbade its export. Furthermore, US nationals were forbidden from even providing any technical assistance whatsoever to the development or maintenance of cryptographic products that would be available in other countries. This caused severe problems for the development of IPsec in that most of the IPsec working group members were US citizens and could only work on the standard, not provide any technical examples or do any testing. Implementations of IPsec had to be developed with the input of US citizens entirely forbidden in order to keep US government regulations from preventing their distribution. This resulted in the slowing of design and deployment of IPsec-compliant systems [DUN2001].

2.1.3. The Standards Process – Outcome

Input to the standards process came from hardware vendors, who wanted "bump-in-the-wire" (BITW – compare BITS) devices to tunnel IP packets through hardware encryption systems. Adding this capability to the standards increased their extent [DUN2001].

The standards produced are very complex. This is an inescapable consequence of a committee process; a much more streamlined standard would be developed by having a competition and awarding a large monetary prize to the winner, which would save everyone money overall due to the never-ending costs of dealing with the permanent excessive complexity that resulted from the committee process. The competition approach was seemingly successfully used to select Rijndael as the AES. Excessive complexity invites misunderstanding, resulting in implementation and user mistakes that leave security holes. Industry, government and academia were each involved in IPsec, and the results show in the multiple options specified. One harsh but useful critique of IPsec stated that although IPsec is the best security option in this area, it is not possible for the authors to determine whether or not IPsec is secure [FER1999].

2.1.4. Applicability of IPsec

IPsec is really only useful for implementing VPNs (Virtual Private Networks). The following are some areas in which IPsec was tried and either found unworkable or workable with difficulties ([ARK2005] pp. 242-246).

2.1.4.1. Neighbour Discovery Protocol (NDP)

There is a basic logical flaw in attempting to use IPsec for NDP: a "chicken-and-egg" problem. In order to exchange keys with the neighbours, they have to be discovered. In order to discover them securely, keys would have to be exchanged with them. Solving this and other problems that were involved, caused additional thorny problems, inducing the IETF to abandon IPsec for use in NDP.

2.1.4.2. IP Mobility

There are some basic concerns with using IPsec and Mobile IP. IP addresses can change rapidly, and new IPsec tunnels have to be set up, which could cause so much overhead that any actual user communication would not have any time to run. The implementation approach to IPsec – relying on IP addresses, which is not a correct approach (see the final paragraph in section 2.2.1., "Key Exchange") – has to be changed to mitigate this. Another problem is how the mobile node could continue to set up Mobile IP tunnels to the host node if the host node is behind a firewall or gateway and the mobile node travels away from the LAN. These problems are not insurmountable and IPsec is still used, running it over the Mobile IP tunnels.

Aside from these, there is the question of using IPsec to secure the binding updates in which the mobile node informs the host of its new IP address. A global authentication infrastructure would be required for this, which does not exist. Also, such an infrastructure would have to track all IP addresses assigned to users and provide this information in a secure way, which would be impractical, to say the least. Instead of using IPsec, a different set of mechanisms was adopted which use the routing infrastructure to assist in authorization of the mobile node.

2.1.4.3. Network Management Protocols

Although IPsec could provide security for all management traffic in a network, it itself does not provide means with which to differentiate nodes in order to provide them with different privileges, since it was not designed for that, but rather to identify different SAs between different users. These protocols would have to add their own user authentication mechanisms at the application layer.

2.1.4.4. Streaming Multimedia

Streaming Multimedia uses RTP (Real-time Transport Protocol), which changes port numbers dynamically. This would prevent use of IPsec implementations that rely on stable IP addresses, upper layer protocol identifiers, and port numbers to locate the policies and SAs to use.

2.2. Operational Aspects of IPsec

2.2.1. Key Exchange

The Key Exchange protocol has a number of weaknesses which were the subject of several investigations.

The Key Exchange protocol is rather susceptible to a Denial of Service attack due to the acceptance of Diffie-Hellman (DH) values; the initiator (or client) can have the responder (or server) doing modular exponentiation for nothing. Even though cookies are used, a Distributed Denial of Service attack can always be mounted. Also, since ISAKMP uses a date and time stamp as a responder cookie, these must be left behind in the responder in order to track initiators, meaning that the responder can be clogged with these, giving rise to a so-called "cookie crumb" attack. Instead, as in the Photuris protocol [RFC2522], the responder cookie should be regenerable from sender information and one local secret ([SIM1999] pg. 3, [RFC2522] pg. 18). There is no resource-limitation feature in ISAKMP, as in Photuris – an initiator can collect ISAKMP responses in a "cookie jar" and then send them all rapidly as key exchange messages ([SIM1999] pg. 4). A saboteur, or "Monkey In The Middle" (MITM) can simulate the initiator to the responder and vice-versa, sending each of them different DH keys so that they waste resources computing a non-matching "shared secret" and fail to discover the attack until later verification fails ([SIM1999] pg. 4) (Note that this is not to be confused with the "Man In The Middle" attack, in which the attacker maintains the illusion, to both parties, that they are each secretly communicating with the other, in order to breach the confidentiality of the communication). Aggressive Mode eliminates the initial cookie exchange, thereby reducing its utility as a counter against DoS attacks. It does not provide identity protection, but it is intended for mobile users, who most need it, due to the ease of eavesdropping on wireless links ([SIM1999] pg. 5). Quick Mode opens the door to a DoS-Replay attack in which an attacker simply replays the Quick Mode packets and the responder uses all of its resources decrypting the packets only to find that the nonces used are the same ([SIM1999] pg. 6). Additional flaws noted in [SIM1999], (pp. 6-8) include the overly general IKE/ISAKMP framework that relies on a Domain Of Interpretation (DOI), requiring further negotiations to agree on specifications, the addition of modes and options which defeats scalability and simplicity, inadequate and inconsistent error messaging, unpadded ID field sizes that indicate the types of contents such as IP addresses, and unauthenticated fields that could be used as Trojan-Horse channels.

However, there seems to be a mistaken diagnosis of a possible "Man In The Middle" attack in [ZHO2000], pg. 1609. Its analysis is that a MITM attack is made possible because the final hash sent by the responder in Main Mode is done using the initiator's suggested SA. An attacker can pose as the initiator in the SA exchange and choose one of several SA offers for the responder and a different one for the initiator. A check of the final hash that is received by the initiator, done by the initator, using the SA supplied to it, will verify the final hash sent. However, the final hash sent from the initiator to the responder should fail its check due to different SAs in use without the MITM any longer, and so should the final hash sent from responder to initiator. This problem seems to be the same, then, as the "Monkey In The Middle" vulnerability noted before.

Also noted in [ZHO2000], pg. 1610, is the possibility of an active attack in which the identity of a correspondent can be learned. Since no authentication is done until initial SAs are set up, an attacker could pose as a responder and learn the identities of any initiator when the SA is set up and the initiator sends its identity.

Several papers provided suggestions to improve key exchange by suggesting new and different protocols, as discussed in the following section.

[AIE2002] suggested a pair of protocols, called JFKi and JFKr, for "Just Fast Keying", "initiator" and "responder", respectively; the former was designed to provide identity protection for the initiator in the key exchange and the latter to provide it for the responder. Applications would be an anonymous client contacting a public server, vs. peer-to-peer. These protocols combat DoS attacks against the responder by not requiring the responder to perform modular computation until the initiator has first done so, and established round-trip communication. This basic idea is also the idea of [CHO2003] (pp. 332-333).

Identity protection is provided to the initiator in JFKi because after doing the key calculation, the initiator sends its identity encrypted. In JFKr, the responder sends its identity encrypted after receiving the initiator's identity. Active identity protection is not possible for both initiator and responder, as noted ([AIE2002] pg. 52), due to the DoS protection for the responder – the initiator has to send its ID first, because the responder can't be allowed to go ahead with modular computation until the initiator has taken on that burden first. Thus the initiator will be subject to an active ID attack in using the JFKr protocol, but not in the JFKi protocol.

It seems that the possibility of a "Man In The Middle Attack" was forgotten; to combat that, public keys exchanged should be signed by a CA (Certificate Authority) at the time that one side sends its identity.

Another proposal ([CHO2003] pg. 329), involves "client puzzles" in which the server requires a client to solve a computationally-intensive puzzle before the responder will create state or do its own computations. The server sends a hash containing its nonce, to the client, along with a partial solution to the hash. The client has to do a certain amount of computation to find the nonce and it has to return the correct nonce before the server will authenticate it, while the server only has to store the nonce for each client. The client's workload increases rapidly and linearly with the number of requests it makes, whereas the line representing linear increase of storage and work at the server has a very low slope when shown on a graph ([LEI2000] pg. 7). The server could vary the difficulty of its puzzles in direct relation (or more) to its load.

It is to be hoped that the debate process within the IETF will adopt these and/or other suggestions for improving the present easily-attackable state of IKE/ISAKMP.

Finally, related to key exchange, an example of the way that implementation can cause security holes is in the practice of treating an IP address as being authenticated by the IPsec AH protocol, since that is what gateways or firewalls can examine for filtering purposes. Actually, the AH authenticates the packets as coming from a user who knows the key. ([FER1999] pg. 5) This means that a different user could use the trusted IP address, set up SAs, and be trusted as a different party. This kind of masquerade is precisely what authentication is supposed to prevent. Since IP addresses are so easily forged, identification must depend upon the possession of secret knowledge, not upon IP addresses ([SIM1999] pg. 6). A general statement of this problem is that somehow binding has to be achieved between entities that are outside of the protocol and their purported identities within the protocol. A related problem is that of different protocols using different names for the same entity [ROE2001].

2.3. Management and Architecture

2.3.1. Other Management Proposals

[GUT2004] has proposed and developed an approach to dealing with the complexity involved in configuring real-world security systems, in order to prevent oversights that cause security holes. The method takes four steps: (1) modeling, (2) expressing security goals, (3) deriving algorithms and (4) implementing. Modeling expresses the security system in mathematical terms which allows it to be processed by algorithms to check for missed areas whose validity in turn can be verified. Thus a security system can be checked for correctness in the design phase before the expense of implementation is incurred.

[TRC2003] has pointed out the paradigm of low-level to high-level interactions and that at each level, the needs of technology, the organization and government mandates must be taken into account (Figure 4).

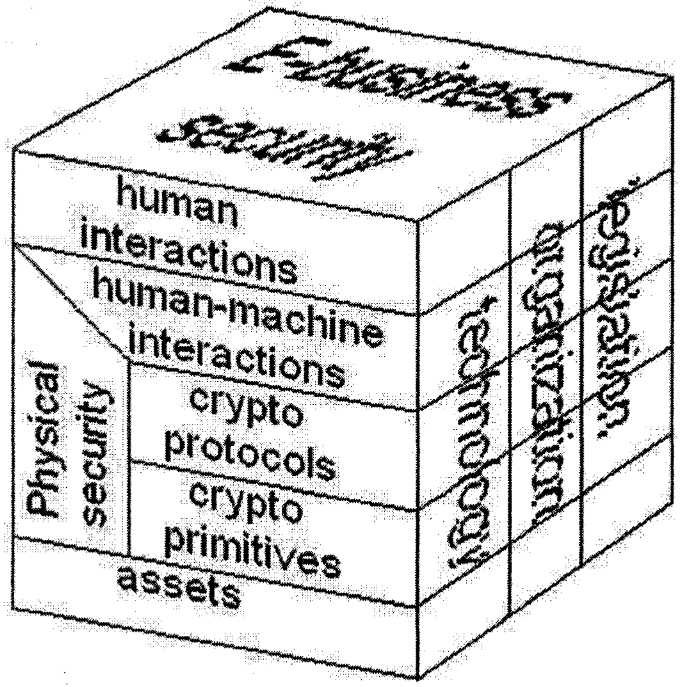


Figure 4. Aspects of E-business security technical layers

"Approaches related to security of IS (Information Systems) are to be linked within appropriate methodology to achieve optimal and balanced solutions for an enterprise." ([TRC2003] pg. 359). The security of E-business should be designed along with the E-business and not added in as an afterthought. Unfortunately IPsec is an afterthought to IPv4.

[DUF2002] has proposed a three-level architecture for security management for distributed multimedia services, arranged in three layers: service, middleware and network ([DUF2002] Figure 1, pg. 364) – note that the unlabeled ellipses represent additional services and managers according to their layer (The ACM – Association for Computing Machinery – did not grant permission to reproduce this figure).

Note that functional concepts at a certain high level require implementation at its level and at all lower levels; for example the policy rules need handling here at the Middleware level as well as the Network level.

[PAR2002] has proposed a "C-ISCAP" (Controlled Internet Secure Connectivity Assurance Platform), "which is an internet information security system based on IPsec." (Figure 5).

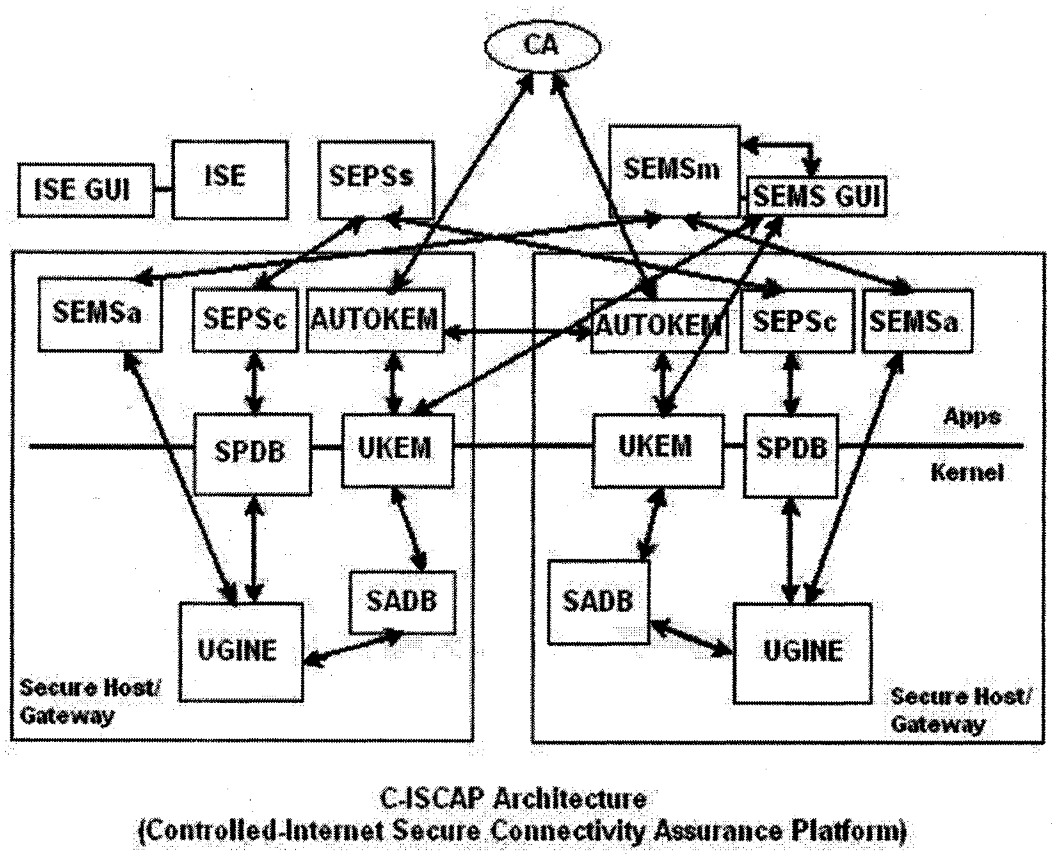


Figure 5. Another implementation-oriented layered architecture for IPsec management. Reproduced with kind permission of Springer Science and Business Media.

Here, ISE stands for "Internet Security Evaluation System", which evaluates system safety and attempts to proactively identify threats. SEPS is the security policy database, SEMS is the Security Management System, "AUTOKEM" is the automatic key exchange mechanism, using a CA (Certificate Authority) to prevent "Man In The Middle" attacks, UKEM is the "Universal Key Management System", SPDB is the Security Policy Database, SADB is the Security Association Database and "UGINE" is the "Universal IPsec Engine".

In a follow-up paper, [KIM2002], it is proposed to use multiple secure IKE sessions in parallel when one C-ISCAP system needs to communicate with more than one other, by using different Diffie-Hellman random numbers in each signature to keep track of the different sessions. A "chicken and egg" dilemma, somewhat similar to the one mentioned before, would result if the policies for creating IPsec associations were to be distributed using IPsec. To improve communication between the UGINE and the SEPS, two separate stacks, one for each communication direction, are proposed.

A design at the Mechanisms and Primitives level was provided by [FER2005] in proposing a multi-accelerator. Each accelerator was provided with its own work queue and a scheduler distributed the work among the accelerators and the CPU (Central Processing Unit). A scheduling algorithm was developed that controlled this distribution of IPsec packet processing. Soft QoS (Quality of Service) could be supported in that higher-priority bit streams would be provided with a higher-priority access to the scheduler.

[LIM2003] proposed a system of policy distribution using a four-layer architecture of management, processing, consumer and target, with the policy data base serving the upper three layers. A policy server defines, stores, and configures policies for the ultimate target systems and the policies are distributed to the targets using IETF standard protocols: COPS-PR (Common Object Policy Service for PRovisioning) or SNMP; the Policy Information Base (PIB) standard is proposed in RFC 3159. The usefulness of this can be appreciated if a large company has tens or hundreds or more IPsec installations to be configured throughout a country or large region, in similar or different ways; automating the configurations helps to prevent human error.

[GAB2004] proposed an "Active Networks" architecture that contains policy, service, management as well as lower modules. This architecture is active in the same sense as the other architectures in this section in that it contains a policy layer and controlling and reactive elements. A "commodity" PC was used, containing an Intel P4 2.2 GHz CPU with 512 Mbit RAM running Red Hat Linux 8.0; 396 packets per second could be processed when the user credential (such as an X.509 certificate) is only contained in the packet(s) and 1190 packets per second (three times the rate) could be processed when the user credential was already contained in the node. However, the size of packets used was not given.

[DON2004] proposes a Secure Name Service (SNS) to enhance availability between cooperating extranets. SNS only answers queries from trusted network domains, and returns a "secure handle" to a service, rather than an IP address as does DNS. "This SH (Secure Handle) is mapped to the real IP address of the host in the SNS framework by SGs (Security Gateways), and the IP address is only known to the SNS server and associated SGs." ([DON2004] pg. 549).

As can be seen from the foregoing, the state of the art in the literature contemplates some systematic approaches to IPsec. What seems to be needed here is a unifying paradigm.

2.3.2. The Erfani Patent

[ERF2003] [USPTO] "outlined a comprehensive system and method for managing security in an electronic network," composed of five functional layers: policy, management, service, mechanisms and primitives (see Figure 6 and the top row of Figure 3). Just as the layered OSI model is a model optimized for the design of communication systems, this five-layer model is optimized for the design of security systems.

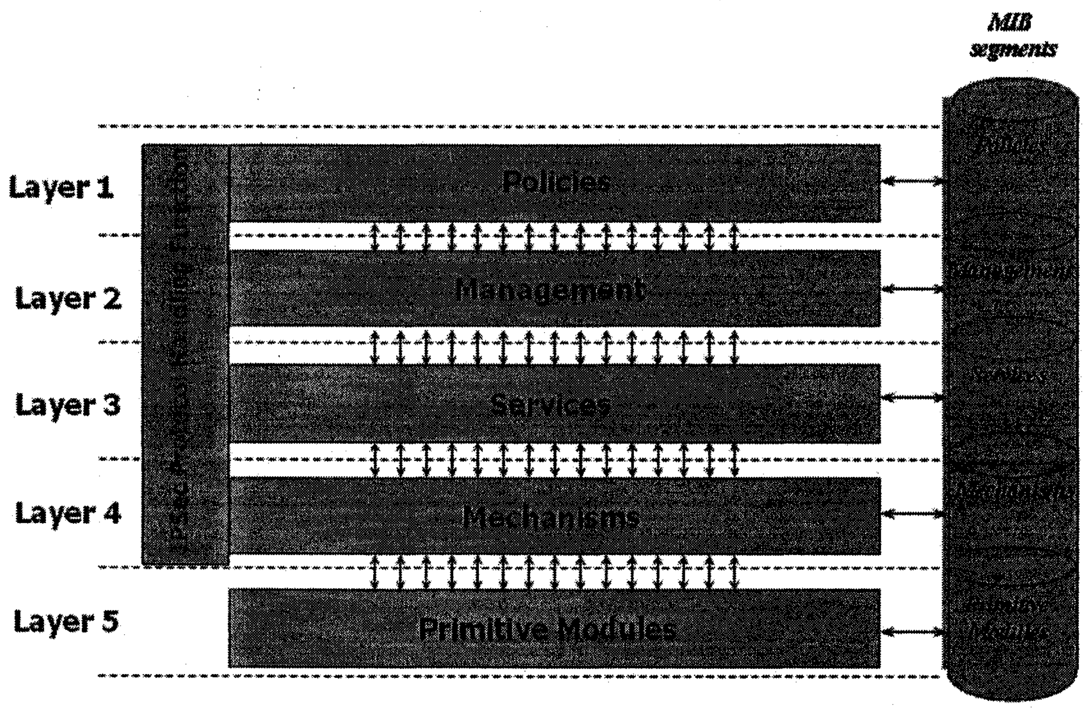


Figure 6. The five-layer security framework, applied to IPsec

In previous work in the author's group in the ECE department at the University of Windsor ([FAH2005], Section 4.2, pp. 84-94, and Chapter V), as noted before, these five layers have been "fleshed out" into modules and several operating scenarios have been described. Modules in the policy layer include: Prevention and detection of IPsec security violations, Network-wide IPsec implementation policy, and Disaster recovery. Modules in the management layer include: Policy control and management of security services, Event logging, IPsec services monitoring, User interface, Interoperability and Recovery and backup. Modules in the services layer include: Access Control, Integrity, Authentication, Confidentiality, Privacy and Rejection of replayed packets. Modules at the mechanisms layer are further subdivided into the following groups: Encryption, Message authentication, Key management, Certificates and Digital signatures. Finally, modules at the primitives layer are further subdivided into the following groups: Prime number generation, Modular arithmetic, Encryption, Hashing and Elliptic Curve Cryptography (ECC), although the latter is not yet used in IPsec. Additional modules at the policy level are SLA (Service Level Agreement) and User Information. Additional modules at the service level include Availability, although this service is not formally specified by the IPsec RFCs and is weak in IPsec (see Section 2.2.1., "Key Exchange").

2.4. Implementations of IPsec

2.4.1. Software Implementations

It was found in [NAY2005] that the Free S/WAN implementation incurred greater performance degradation than 802.1X due to its end-to-end security with double authentication, a stronger encryption method as well as better key management and tunneling. One of their results was that in using DES for FTP (File Transfer Protocol), degradation of performance was worse than the degradation for HTTP (HyperText Transfer Protocol) in going from 802.1X to Free S/WAN.

In [KER1997], a software implementation of IPsec was done on Linux and several different versions of BSD (Berkeley Software Distribution). It was found that encryption of packets "was a major bottleneck", resulting in a factor of ten decrease in throughput in a ping performance test. Authenticating packets caused no significant decrease in throughput in this test. Unfortunately their results for UDP (User Datagram Protocol) throughput and TCP transfer throughput were not reported in meaningful units – for example it was not possible to discern the meaning of 5000 units of throughput in terms of "cpu time". However, the factor of ten decrease in throughput using ESP was evident, and in these tests, the use of AH did make significant differences in throughput, reducing throughput by 30% and 50% in UDP transfer and 50% and 60% in TCP for MD5 and SHA-1, respectively.

In [KAN2004], an IPsec stack was developed for the Linux kernel 2.4 and 2.6 series. HMAC-SHA-1 and HMAC-MD5 (HMAC: Hashed Message Authentication Code) were implemented for authentication, NULL, DES-CBC, and 3DES-CBC were included for encryption. This work was submitted to the Linux kernel maintainers; it had the advantage of simplicity, but it differed in the Security Association and Policy Database (SADB and SPDB) cache lookup system used in IPv4 and IPv6, leading to it being declined for use with Linux. Throughput and other numerical results were not reported in this work.

2.4.2. Hardware Implementations

2.4.2.1. FPGA Implementations

[DAN2000], "An Adaptive Cryptographic Engine for IPSec Architectures", was the first to take advantage of compressibility of dynamic configurations. The AES finalists at the time, which were MARS, RC6, Rijndael, Serpent and Twofish, were implemented. Compared with software implementations, throughput speedup of 4 to 20 times was achieved while the key setup time was reduced 20 to 700 times.

[BEL2002], "GRIP: A Reconfigurable Architecture for Host-Based Gigabit-Rate Packet Processing", offloaded processor cycles onto a dedicated network interface, which allowed more bandwidth for the cryptography. Throughput of 50 Mbps were measured, possibly due to decryption failures of packets over 1500 bytes in size, and flow-control issues caused by the design of the header-processing logic.

In [CHE2002], "Implementation of an FPGA Based Accelerator for Virtual Private Networks," a 3DES core achieved 120Mbits/s in CBC (Cipher-Block Chaining) mode, three times as fast as a software implementation.

In [MCL2002], "A Single-chip IPSEC Cryptographic Processor," a single-chip IPsec cryptographic processor was implemented on a single XCV1000E Xilinx Virtex FPGA. Throughput results were 310 Mbps for AES and 78 Mbps for SHA-1.

In [KIM2004], "Design and Implementation of a Private and Public Key Crypto Processor and its Application to a Security System," AES: 390, 3DES: 267, SEED: 358 and KASUMI: 568 Mbps were achieved using an FPGA. Parts of the processor were later implemented in 0.5μm CMOS. To test and demonstrate the chip, a custom board providing real-time data security for a data storage device was developed. It encrypted all data going to the hard disk and decrypted all data leaving it.

In [LUJ2005], "IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application," IKE was done in the Power PC portion of the FPGA; the hardware invoked the software only when necessary. Throughput result were AES: 1197, SHA-1: 304 and MD5: 277 Mbps.

These all reach to the Services Layer – no higher.

2.4.2.2. FPGA Implementations of Primitives

2.4.2.2.1. AES

As noted before, Rijndael was chosen as AES in October 2000 [REJ2003].

In [JAR2003], a Finnish reference, a fully-unrolled implementation of Rijndael was done using a Xilinx Virtex-II FPGA, implementing the S-Boxes (Substitution Boxes) combinatorially. It was designed fully pipelined so that a new data-key pair can be input at every clock cycle. The design consists of eleven separate blocks. Throughput results were 17.8 Gbps for an individual block, but overall throughput in a cipher feedback mode such as CBC, was not reported.

In [STN2003], a reference from Belgium, another fully-unrolled implementation of Rijndael was done. Using the Xilinx Virtex E FPGA LUTs (Look-Up Tables), throughput of 1,563 Mbps was achieved, and using the RAM to implement the S-Boxes, throughput of 11,776 Mbps was achieved.

A final-round contender for AES, Serpent, was implemented on a Xilinx Virtex XCV1000 FPGA in [ELB2000]. Four different architectures were implemented: Iterative Looping, Iterative Looping with Partial Loop Unrolling, Full Loop Unrolling and Full (32-stage) pipelining. Throughputs achieved were 61.92 Mbps, 444.16 Mbps, 312.32

Mbps and 4.86 Gbps, respectively, although the final one was in ECB (Electronic Code Book) mode only. Software could process Serpent at a rate of 26.90 Mbps of throughput.

2.4.2.2.2. Hashes

In [KAN2002], SHA-1, HAS-160, and MD-5 were implemented on one chip, an Altera EP20K FPGA. Combining SHA-1 with HAS-160 reduced the required logic elements by 27%. Throughput results depend on the speed grade of the device; grade 3 was used. Results were 114, 160 and 142 Mbps, respectively.

In [ZIB2003], a Chinese reference, the SHA-1 algorithm was implemented on an Altera EP1K FPGA and a maximum throughput of 268.99 Mbps was achieved.

In [KHA2005], a reference from the University of Victoria, BC, Canada, the similarities between MD5, SHA-1 and RIPEMD-160 (since they are based on an earlier hash function, MD4) were used to design one chip to perform all three; a LUT (Look-Up Table) design on a Xilinx Virtex II FPGA. Simulation only was reported; in that, projected throughput was 145.72, 116.94, and 116.94 Mbps, respectively.

In [DEE2001], a reference from the Memorial University of Newfoundland, Canada, two implementations of MD5 using iteration and full-loop unrolling were done on a Xilinx Virtex V1000 FPGA with a clock rate of up to 200 MHz. Throughput was 165 and 354 Mbps, respectively.

2.4.2.3. ASIC (Application-Specific Integrated Circuit) Implementations

In [WUL2001], "CryptoManiac: A Fast Flexible Architecture for Secure Communication," 0.25 μm standard-cell technology was used to implement the "CryptoManiac" processor, a 32-bit VLIW (Very Long Instruction Word) dedicated cryptographic processor which contains four functional units each with an adder, a 1kB S-Box cache, two logical units for instruction combining, a rotator, and two multipliers.

A specialized instruction set optimized for running cryptographic algorithms was provided. One key innovation was combining arithmetic and logical operations within a single cycle, since the latter type of operation often follows the former in cryptographic processing, allowing the processor clock cycle to be better used. The best results for Rijndael (AES) that were achieved was about a 64 Mbps encryption rate, superior for a software implementation.

2.4.2.4. ASIC Implementations of Primitives

In [WAN2004], an ASIC implementation of SHA-1 and MD5 was done using 0.25 μm CMOS technology; their innovations were reduced hardware complexity in reducing the number of multiplexers and hardware sharing by using common hardware for both algorithms. Throughput results were 417 and 520 Mbps for SHA-1 and MD5, and about 94 and 117 Mbps when digital signing of these hashes was required.

In [REJ2003], two ASIC implementations of Rijndael were done using 0.13 μm CMOS technology; in one, only one lookup table was used to implement the S-Box used for all rounds and access to it is pipelined between rounds. In the other, separate S-Boxes were implemented in order to use them concurrently. Both implementations achieved 2.56 Gbps of throughput in feedback modes.

2.4.3. Conclusion of the "Implementations" Section

In conclusion and summary, the heavy overhead incurred by encryption mandates hardware acceleration. Software running on a general-purpose processor can typically produce AES (Advanced Encryption Standard), i.e., Rijndael, throughputs of low tens of Mbps (i.e., 30) [DAN2000]. 70.5 Mbps using Visual C++ was achieved, as reported in [MRO2000]. FPGAs (Field-Programmable Gate Arrays) can achieve up to 176 Mbits/s implementing DES (Data Encryption Standard) and up to 964 Mbits/s implementing AES [WOL2004]. An ASIC processor achieved 2.29 Gbits/s of AES throughput in a 0.18μm CMOS standard-cell technology in 2002 [SCH2002], [VER2003].

None of these implementations used a comprehensive functional architecture such a the present, proposed, five-layer framework.

2.5. An IPsec Application

[GOD2002] used IPsec to secure a wireless gateway. The Microsoft Windows 2000 implementation – i.e., a software implementation – of IPsec was used; but the WEP (so-called "Wired Equivalent Privacy") implementation was not specified. A Buffalo WLI PCM-11 wireless network interface PC card was used. Throughput was 604 kBps unencrypted, 458 kBps using 40-bit WEP, 355 kBps using IPsec with DES and MD5 and 209 kBps using IPsec with 3DES and SHA. Multiply by 10 to get speeds in bits per second (bps), which are believable in terms of the roughly 30 Mbps maximum throughput possible using software implementations of IPsec.

2.6. High-Level Synthesis for Hardware Implementations

UML (the Unified Modeling Language) [JAC1998] was considered as the design language for the FPGA portion of this project, but was rejected as not suitable because it was object-oriented, too high-level and tools to program FPGAs were not available.

Simulink, a software package by Mathworks Inc., was considered. In a user report, "The engineers at SELEX generated a specification for what they wanted the FPGA to do using Simulink and used Xilinx System Generator for DSP to program the FPGA to match the Simulink model," [MAT2006].This was found to be aimed at DSP (Digital Signal Processing) and required the basically manual step of replacing the Simulink standard blocks with Xilinx standard blocks.

According to a tutorial on HLS (High-Level Synthesis), there are many unanswered questions when it comes to using this technique in a complete context. "Much work needs to be done before synthesis becomes really practical," ([MCF1988] pg. 335). According to [COM2002], HLS tends to produce larger and slower designs than structural description can produce. Its descriptions can leave many aspects of the circuit unspecified. Also, optimizing for the bit width of operands cannot be done.

In [LIH2005], the rather heroic measure of inserting a timing and netlist control guidance stage between the place and route steps of physical synthesis for ASICs had to be done. It is reported that there are very few algorithms that have been proposed to make the HLS tool aware of the layout information, so that the resulting physical design can be improved.

Investigating SystemC, a HLS language, "The performance of this simulation kernel is not to be compared with that of commercial VHDL/Verilog simulators at the present," ([WIKIP], SystemC). Also, the size and complexity of SystemC models becomes too large to be practical in modern design projects, and new tools are being researched to deal with the complexity [GEN2006].

HLS is an open area of research (Dr. M. Khalid, in personal conversation, May 2006).

As a result of these investigations, the Xilinx ISE (Integrated Software Environment) and EDK (Embedded Development Kit) software packages were chosen, due to availability, as well as availability of compatible hardware development boards such as the Xilinx "Microblaze" (or "Multimedia") boards, which are equipped with Xilinx Virtex II FPGAs. HDL (Hardware Description Language) programming in VHDL or Verilog was chosen, rather than attempting to use HLS.

2.7. Random Number Generators

Since an RNG (Random Number Generator) was incorporated in this work, some background on PRNGs (Pseudo-RNGs) and CSPRNGs (Cryptographically Secure PRNGs) was investigated.

It is desirable to have an RNG to generate keys and IVs (Initial Vectors), for if the users were entrusted with this task, they would skip it as a time-consuming burden, leaving them set to all zeroes, or provide short and simple values, or naturally provide predictable values. IVs are often sent in the clear and only need to be different, since their purpose is only to vary the ciphertext. However, it should not be possible to predict the future values or determine the previous values used for keys, in case some become known to an attacker. An RNG which has the property of difficulty of determining past or future values from current values is known as *cryptographically secure*. For testing, it is useful to use a PRNG for its repeatable output. For actual use, the PRNG is *seeded* with an initial value taken from randomly-occurring values, such as the time of day, the value of a free-running counter or the time delays determined between user activity. Many such values are often combined together, often using the XOR (Exclusive-OR) operation, for the greatest possible unpredictability.

To make an exhaustive search – involving trying all possible values (somewhat inaccurately known as "brute force") – attack impractical, a long sequence, known as the "period" or "cycle length", before the PRNG repeats, is important. A well-designed PRNG has a cycle length of 2^e, where e is the number of bits in the state; the "state" is the core "word" on which the PRNG operates and which provides the source of the bits of the output number) [LEC1998]. The number of bits in the state is also known as the "linear complexity", of a linear PRNG, of course [WAL2007]. A good PRNG has a cycle length of over 2^{200} [LEC1998]. The "Mersenne Twister" algorithm has a period of $(2^{19,937})-1$ ([WIKIP], "Pseudorandom_number_generator").

Linear PRNGs, such as LFSR (Linear Feedback Shift Register) types suffer from predictability [HP12006]. For example, an LFSR was developed that operated at high speed, low power and high precision and was useful in general communication systems, RADAR (RAdio Detection And Ranging) signal simulation and processing environments where random numbers exhibiting more than one type of statistical distribution were needed [WEI2004], but would not be completely useful for cryptography. In order to be cryptographically secure, the RNG should perform well in strict statistical tests. One such is the “DIEHARD” series of tests, developed by George Marsaglia at the Florida State University Department of Statistics ([SOT1999], pg. 2), [MAR1995], which consists of fifteen different tests (see also [WIKIP], “Diehard_tests”, for an intuitive description of each). "The higher the entropy in a series of numbers is, the more difficult it is to predict a given number on the basis of the preceding numbers in the series," [HAA1999]. "True random numbers are independent from each other and therefore unpredictable but they are rarely employed," [KAR2000]. For more comments on randomness required for cryptography, see [RFC1750].

AES itself makes a fine CSPRNG with an enormous period [HP12006]. It could be used to encrypt the value of a counter beginning at some seed, using CBC or some other mode, which would give a period of 2^b, where b is the cipher block length (128 to 256, for AES) ([HEL2003] pg. 324); it could also be used to repeatedly encrypt its own output (as in CBC encrypting blocks of all zeroes), but the period cannot be guaranteed using this method ([HEL2003] pg. 324). The foregoing could be done, of course, starting with some seed IV. AES also has tremendous non-linearity included in its design [FIPS197].

CHAPTER III

DESIGN AND METHODOLOGY

3.1. Design of an FPGA AES Hardware Accelerator

3.1.1. Introduction

The AES cipher was chosen in October 2000 [REJ2003], to replace DES (Data Encryption Standard) which is now too computationally intensive to use in obtaining good security, given that it has to be run three times to obtain an effective key length that is sufficiently secure.

3.1.1.1. Overview of AES

AES is a ten-round substitution-permutation cipher [FIPS197]; it carries the 128 (or 192 or 256)-bit plaintext value through "rounds", i.e., repetitions of the four processing steps, which are: XOR with the "key permutation" or "key expand" value for that round (i.e., the round key), substitute (sub) bytes using an "S-box", shift rows and mix columns, to convert it into the ciphertext. The value being carried is known as the *state*. The inverse cipher does each round in reverse order, meaning that an inverse S-box, and an inverse mix columns function are required. The same key expand values are applied, in reverse order. A former name of the algorithm chosen as the AES, Rijndael, was "Square", as can be seen after. Rather than attempting a repetition of the complete details of AES, an idea, or the "flavour" is presented here and the reader is referred to the standard for the complete details [FIPS197].

The difference between Rijndael and AES is that Rijndael is defined for block and key sizes of every increment of 32 bits from 128 to 256 bits, inclusive, whereas AES has a fixed block size of 128 bits and key sizes of only 128, 192 and 256 bits ([WIKIP], "Advanced_Encryption_Standard"). The 128-bit key size definition was chosen to implement for this project, for simplicity.

The following listing shows the pseudo-code for the AES encryption cipher [FIPS197]:

```
Cipher(byte in[4*N_b], byte out[4*N_b], word key[4*N_b])
      byte state[4*N_b], w[4*N_b, N_r+1], state[4*N_b], integer round
      state = in
      ComputeRoundKey(key, w[k_x, 1])
      AddRoundKey(state, w[k_x, 1])
      for round = 1 step 1 to N_r-1
            ComputeRoundKey(key, w[k_x, round+1])
            SubBytes(state)
            ShiftRows(state)
            MixColumns(state)
            AddRoundKey(state, w[k_x, round+1)
      end for
      ComputeRoundKey(key, w[k_x, N_r+1])
      SubBytes(state)
      ShiftRows(state)
      AddRoundKey(state, w[k_x, N_r+1)
      out = state
```

Note: N_b: number of blocks, N_r: number of rounds, w: key expand array, k_x: key permutation, 4: 4 bytes, i.e., each block is 4 bytes, or 32 bits.

and the inverse, or decryption cipher:

```
InvCipher(byte in[4*N_b], byte out[4*N_b], word key[4*N_b])
      byte state[4*N_b], w[4*N_b, N_r+1]
      state = in
      ComputeRoundKey(w[k_x, 1])
      AddRoundKey(state, w[k_x, 1])
      for round = N_r-1 step -1 downto 1
            InvShiftRows(state)
            InvSubBytes(state)
            ComputeRoundKey(w[k_x, round+1])
            AddRoundKey(state, w[k_x, round+1)
            InvMixColumns(state)
      end for
      InvShiftRows(state)
      InvSubBytes(state)
      ComputeRoundKey(w[k_x, N_r+1])
      AddRoundKey(state, w[k_x, N_r+1)
      out = state
```

Note that, in spite of the pseudo-code provided by the FIPS (Federal Information Processing Standard), in decryption the entire set of round keys must be computed before decryption can begin, since the last must be used first. For AES-128, N_b=4 and N_r=10. *w* is chosen to represent the array of key expand values in [FIPS197].

3.1.1.1.1. Round Keys

The round keys, or "key expand values", are permutations of the key. The first "permutation" is the key itself. In each round, this is XORed with the "sub-bytes" (substituted bytes) of the current key permutation value rotated left by 8 bits, using the same S-box as do the rounds, i.e., "key XOR (sub(rot(key)))". Also, the high-order byte of each 32-bit portion of the 128-bit block is XORed with the output of a function of the round step in which the HOB (High-Order Byte) is determined from Table 3. This function is called "Rcon" (Round Constant), where Rcon(step) = $[2^{(\text{step}-1)}\{00\}\ \{00\}\{00\}]$ in GF(2^8) [FIPS197]. (GF: Galois Field).

Round	01	02	03	04	05	06	07	08	09	0A
Rcon (HOB)	01	02	04	08	10	20	40	80	1B	36

Table 3. The AES round constant (hex)

Figure 7 [FIPS197] illustrates how the round key is added in each round. Each 32-bit block of the round key is XORed column-wise to a matrix formed from the bytes of the state.

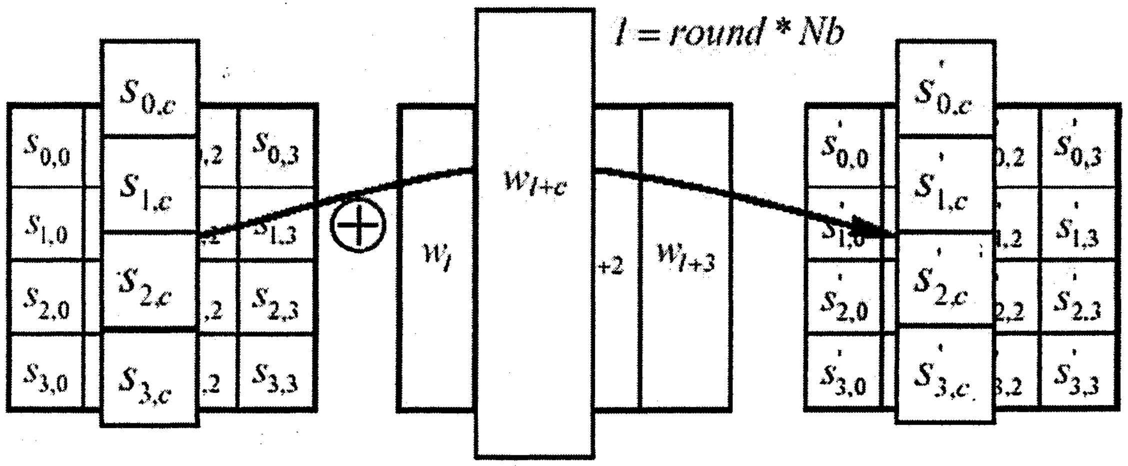


Figure 7. AES add round key

3.1.1.1.2. Substitute ("Sub") Bytes

Figure 8 [FIPS197] illustrates the substitution process. Each byte is replaced, on an individual basis. This operation introduces non-linearity.

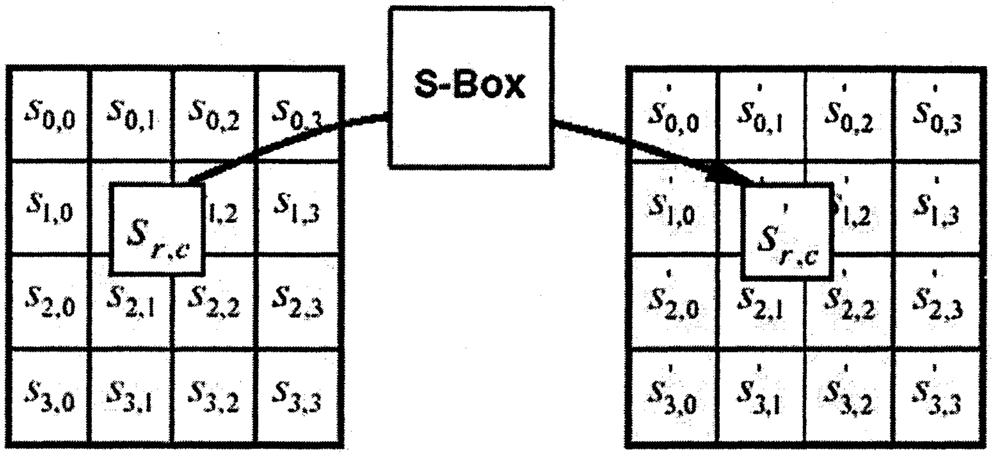
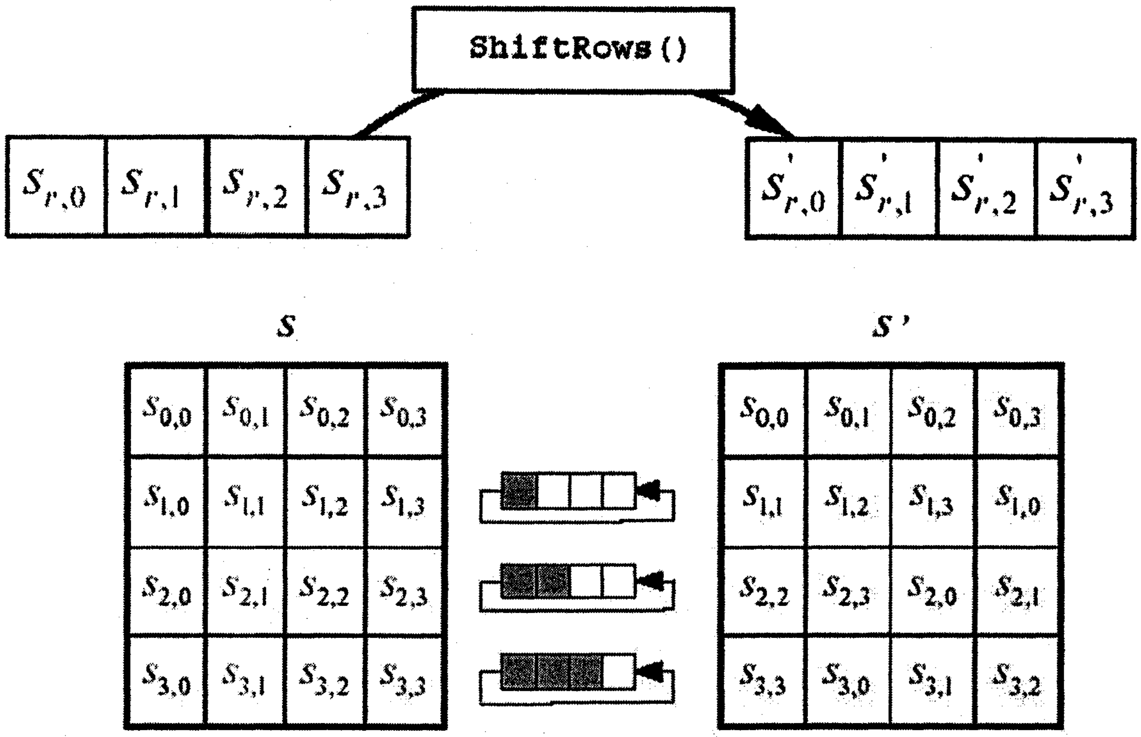

S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,2}$ $s_{1,3}$
$s_{r,c}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{r,c}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 8. AES "sub"-bytes

3.1.1.1.3. Shift Rows

Figure 9 [FIPS197] illustrates how each row is shifted, each by one successive byte extra.



ShiftRows()
$S_{r,0}$ $S_{r,1}$ $S_{r,2}$ $S_{r,3}$
$S'_{r,0}$ $S'_{r,1}$ $S'_{r,2}$ $S'_{r,3}$
S
S'
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,0}$
$S_{2,2}$ $S_{2,3}$ $S_{2,0}$ $S_{2,1}$
$S_{3,3}$ $S_{3,0}$ $S_{3,1}$ $S_{3,2}$

Figure 9. AES shift rows

3.1.1.1.4. Mix Columns

Figure 10 illustrates how the "Mix columns" operation is applied. The operation itself is a complex and staggered combinatoric operation done to each byte of each 32-bit block (column) in the matrix.

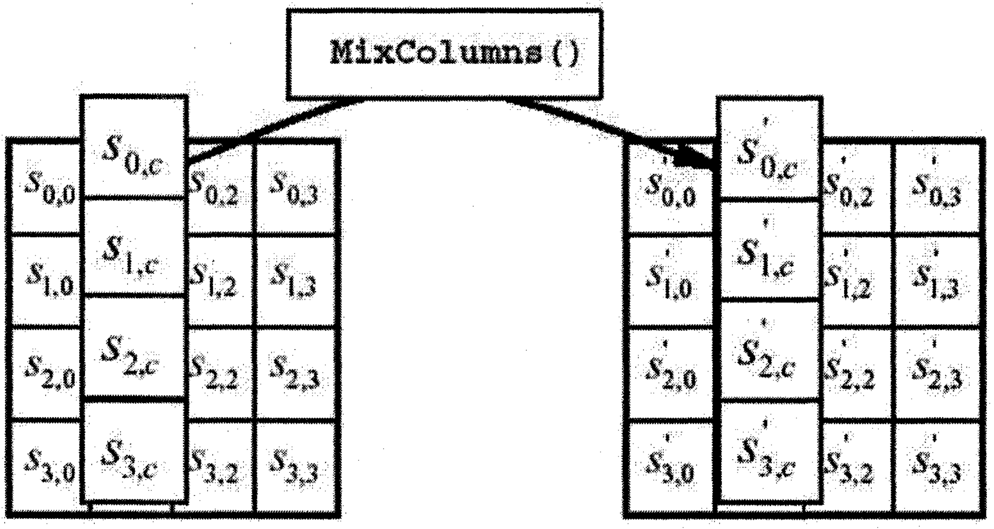

MixColumns()
$s_{0,0}$ $s_{0,c}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,c}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,c}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,c}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,c}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,c}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,c}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,c}$ $s'_{3,2}$ $s'_{3,3}$

Figure 10. AES mix columns application

3.1.2. Technology

As noted before, Xilinx was chosen for tool availability on the desktop PC (Personal Computer) used for this research, which had a 2.39 GHz motherboard containing a "Celeron" CPU and 256 MB of RAM. The Xilinx software packages used were version 8.2i of: the ISE (Integrated Software Environment – 8.2.03i, specifically) and the EDK (Embedded Development Kit, version 8.2.01i, specifically).

In the ISE individual modules are built – including even an entire project instance from the EDK –, simulated, and loaded as a "bit file" or "bitstream" to a target board, whereas in the EDK individual modules are put together , such as the Power PC (PPC), RS232/UART (Universal Asynchronous Receiver-Transmitter) module and any user IP (Intellectual Property) that was developed, and entire systems are built [XILQST], [XILIDT].

The Xilinx ML403 board, shown in Figure 11, an embedded system, was chosen for this research; the board includes a Xilinx Virtex-4 FPGA which in turn contains a hard-core PPC 405 – the XC4VFX12-FF668 [XILUG80] (a "hard-core" device is actual device with its design doped into the semiconductor, as opposed to a "soft-core" device, which is implemented by means of the FPGA fabric). The Virtex-4 was the next to most recent version in the Xilinx Virtex series; it is a general principle never to buy version one of anything (even the first of a new sub-version of an existing series). Moreover, the board had a reasonable price, of $495 US, as of June 2006.

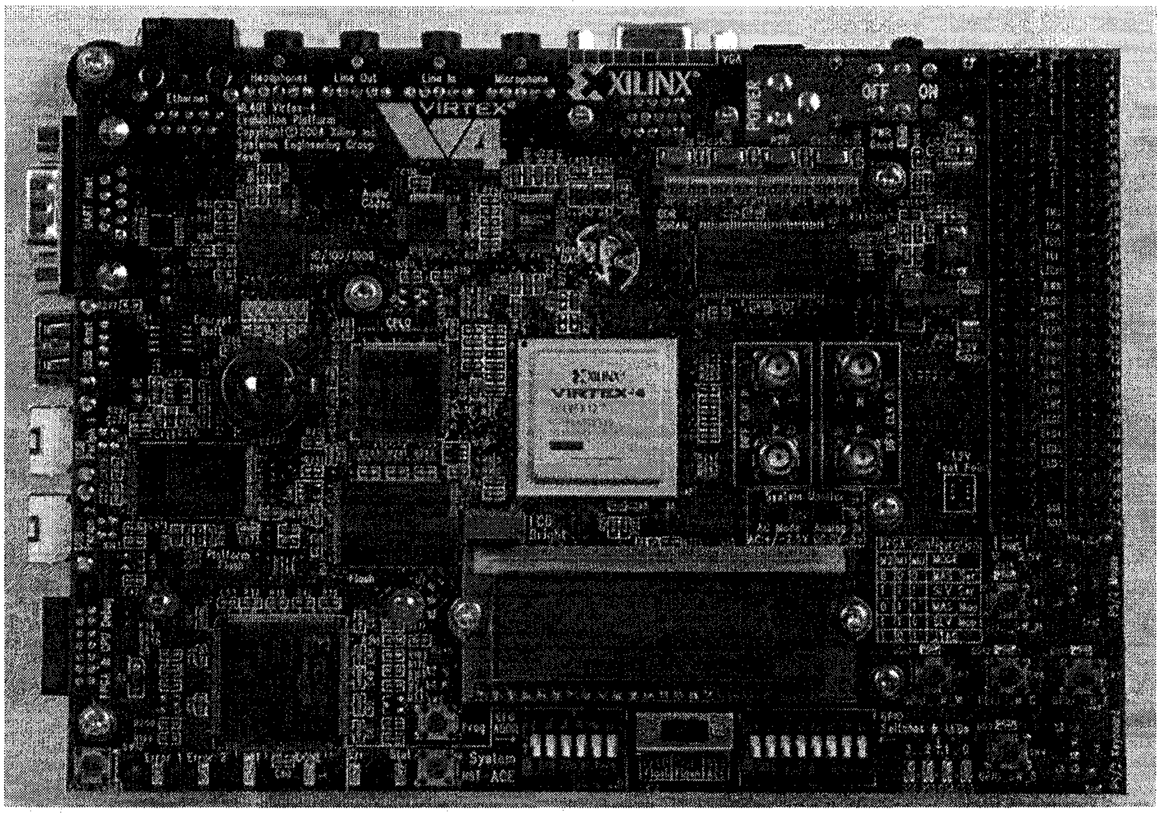


Figure 11. A photograph of the Xilinx ML403 board

3.1.2.1. Specific Virtex-4 FX12 FPGA Features

The Virtex-4 FPGA contains four embedded Digital Clock Managers (DCMs) that can divide the clock and provide an additional three clock phases at each multiple of ninety degrees [XILV4DS].

It also contains on-chip BRAM (Block RAM) useful for small software programs up to 128kB [XILML403T] and which is available for use instead of the FPGA fabric when appropriate, such as for ROMs (Read-Only Memories) and RAMs. The Xilinx "primitive" name for individual portions of BRAM is RAMB16 – individual elements incorporated in Xilinx FPGAs are known as "primitives" within the Xilinx company (not to be confused with the use of the term "primitive" in this work to mean the lowest-level functionality of a security system). The XC4VFX12 has thirty-six 18kB blocks of BRAM [XILV4DS].

The XC4VFX12 FPGA also contains 5,472 cells, or "slices", of FPGA fabric containing 10,944 LUTs (Look-Up Tables) at two LUTs per slice, which is relatively small compared to the 10,752 slices (21,504 LUTs) of the Virtex-II FPGA in the Xilinx "Microblaze" boards [XILV4DS] [XILV2DS], however, those FPGAs (XC2V2000-FF896) do not contain the hard-core PPC. Note that the suffix of the part number, such as FF668 or FF896, refers to the package type and number of pins [XILPKG].

3.1.2.2. ML403 Board Features

The ML403 board has a 100 MHz clock – which means a 10 ns clock period, and it has an expansion header of many pins, connected to FPGA pins on the circuit board, which is very useful for oscilloscope measurements. It has an RS232 serial port, 1MB of SRAM (Static RAM), and a JTAG (Joint Test Action Group) port for downloading the firmware and debugging. A special cable, the "PCIV", or "Parallel Cable IV" (pronounced "PC-four") cable is required to connect from the PC parallel port to the JTAG port.

In addition to the XC4VFX12 FPGA, the board also has an ACE (Advanced Configuration Engine – XCCACE), Flash EEPROM, (XCF32P, 8 MB), a CPLD (Complex Programmable Logic Device – XC95144XL), a Flash Configuration controller, an EEPROM (4kb IIC – Inter-IC bus – interface), an LCD (Liquid Crystal Display) screen, push buttons, LEDs (Light Emitting Diodes), and other features. For the purposes of this work, only the PPC and FPGA (integrated together in one) chip was used, of the major ICs available on the ML403 board.

3.1.3. Selection of the Base Design

The various designs examined for use as a starting point were as follows. The design by Rudolf Usselmann of ASICs.ws [USS2002], done in Verilog, was clearly documented, with detailed synthesis results for a Xilinx Spartan IIe XS2V200-6 FPGA, so it was chosen. Others looked at were a low-area implementation done in SystemC requiring 500 clock cycles to encrypt or decrypt a block for the 128-bit AES algorithm [VILL22005], a 128-bit implementation done in VHDL [SAT2004], a work in progress only tested at the gate level with placement and routing still to be done [HUR2002], and an advertised "Ultra High Speed AES (Rijndael) Crypto Processor," not in the public domain [DEV2003].

3.1.3.1. Some Aspects of the Usselmann Design

The Usselmann design [USS2002] included a *text_in* vector to input the plaintext (or ciphertext, in the case of the inverse transform, or decryptor), a *key* vector to input the key, a *text_out* vector to output the ciphertext (or plaintext), a *keyload* signal to initialize the key expand values from the key, and a "load" signal to initiate the transform. In the encryptor, *keyload* was connected to load because the key expand values are generated as the rounds require them, but in the decryptor, *keyload* was a separate input. There was a *done* pulse output in both, and a *keydone* output for the decryptor, indicating that the key expand values were generated and stored internally. *Keyload* only had to be repeated in the decryptor if the key was changed, but it had to be done before the inverse transform using that key could be done, which is always the case, since the inverse transform requires the last key expand value first.

This design features the S-box instantiated sixteen times, to process each of the bytes of the state simultaneously each round. This increases speed at the expense of FPGA slices and/or FPGA resources such as BRAM, since the S-boxes are realized as ROMs.

3.1.4. Architecture Provided by Xilinx

For the user peripheral, Xilinx provides the outer wrapper VHDL, named after the user peripheral, and a core, named user_logic.vhd [XILUT2003], [XILML403T] for VHDL, if VHDL is chosen, or ".v", if a Verilog implementation is selected. The outer wrapper is an IPIC (IP – Intellectual Property – InterConnect) which instantiates the IPIF (IP InterFace) for the OPB (On-Chip Peripheral Bus) – the IPIF is a subset of the OPB – and user_logic.vhd, and interconnects them. Note that here IP stands for "Intellectual Property". In this work, the AES encryptor and decryptor were implemented as two separate user peripherals, called "aes_enc" and "aes_dec", respectively.

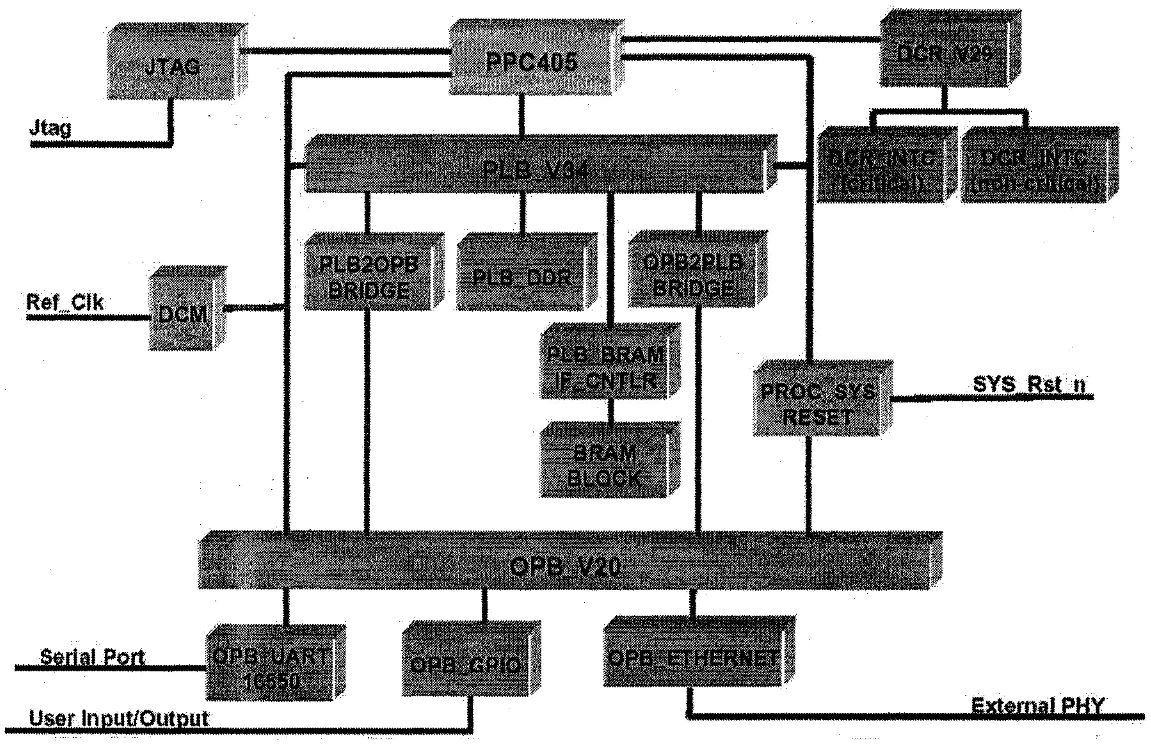


Figure 12. Xilinx system architecture

Referring to Figure 12 ([XILUT2003] pg. 21), the OPB also serves such peripherals as the UART and GPIO (General-Purpose IO), that may be instantiated as modules using the Xilinx EDK, and is connected via the "PLB2OPB" bridge to the PLB (Processor Local Bus) which is connected to the PPC CPU. The PLB also connects to the BRAM via a controller module. The DCM modules connect to the PPC and to the busses. The JTAG port connects to the PPC. The modules required are added in the EDK using the BSB (Base System Builder) wizard or from the available peripheral window pane on the left in the EDK [XILML403T].

Xilinx provides the standard IP cores used, such as the UART. In this work, a communication speed of 57,600 kbps was chosen using the BSB.

Xilinx provides an interface in the user_logic file itself for the user to interface with the outer wrapper – the user programs the *slv_regs* ("slave registers") from software and can access them in the user_logic core using the VHDL shown after (or Verilog, if selected). Xilinx also provides the software libraries in C source code to access the slave registers via software.

The C code to read and write the *slv_regs*, provided by Xilinx is shown here (the first line is just an associated variable declaration).

```
Xuint32 Reg32Value = 0;
Reg32Value = AES_ENC_mReadSlaveReg0(XPAR_AES_ENC_0_BASEADDR);
AES_ENC_mWriteSlaveReg0(XPAR_AES_ENC_0_BASEADDR, Reg32Value);
```

For a decryptor instruction, "ENC" is replaced with "DEC" in both places in each instruction. For a different *slv_reg*, the "0" in "SlaveReg0" is changed to the desired number, from 0-12 (decimal notation).

The user fills in the rest of user_logic with his HDL – in this work, the Usselmann core's top level-file. The outer wrapper is fully provided by the Xilinx tools [XILML403T] and does not need to be modified by the user unless the user desires to add something extra, such as an external connection of a signal to an FPGA pin, as was done in this work. The outer wrapper is always in VHDL and interface is accomplished using the default binding rules if user_logic is in Verilog ([XILIPTS3] pg. 28). The default binding rules state, in part:

> If the entity name is the same as the component name, then this entity is bound to the component.
> If there are multiple architectures for the same entity, the last compiled architecture for the entity is chosen [MAR2003].

The following is a portion of user_logic.vhd – showing the "slave register" interface provided by Xilinx:

```
      SLAVE_REG_WRITE_PROC : process(Bus2IP_Clk) is
        begin
          if (Bus2IP_Clk'event and (Bus2IP_Clk = '1')) then
            if (Bus2IP_Reset = '1') then
              slv_reg0 <= (others => '0');
              [...]

            else
              case slv_reg_write_select is
                when "1000000000000" =>
                  for byte_index in 0 to (C_DWIDTH/8)-1 loop
                    if ( Bus2IP_BE(byte_index) = '1' ) then
                      slv_reg0(byte_index*8 to byte_index*8+7) <=
Bus2IP_Data(byte_index*8 to byte_index*8+7);
                    end if;
                  end loop;
                [...]

                when others => null;
              case;
            end if;
          end if;
        end process SLAVE_REG_WRITE_PROC;

       SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0,
slv_reg1, slv_reg2, slv_reg3, slv_reg4, slv_reg5, slv_reg6, slv_reg7,
slv_reg8, slv_reg9, slv_reg10, slv_reg11, slv_reg12 ) is
        begin
          case slv_reg_read_select is
            when "1000000000000" => slv_ip2bus_data <= slv_reg0;
            [...]

            when others => slv_ip2bus_data <= (others => '0');
          end case;
        end process SLAVE_REG_READ_PROC;
```

3.1.5. VHDL in Xilinx

Variables are called "signals" in VHDL (in Verilog, "wire", or "reg"), and arrays of signals are called "vectors", typically declared as "std_logic" – "standard logic" in which the values allowed are shown in Table 4. Typically only "0" and "1" are used, and "U" and "X" appear in practice, as found in this work.

U	Uninitialized

X	Forcing Unknown
0	Forcing 0
1	Forcing 1
Z	High Impedance
W	Weak Unknown
L	Weak 0
H	Weak 1
-	Don't care

Table 4. IEEE VHDL std_logic values

Signal processing is typically done within "processes", each of which are associated with a clock edge. Most conditional logic blocks can only be implemented within processes. However, boolean logic and assignments can be done "asynchronously", but that should be kept to an absolute minimum in Xilinx VHDL for FPGAs. The format of a process block is shown:

```
MY_PROC: process (myclk, myrst)
begin
  if (myrst = '0') then
    myvar <= '0';
  elsif (rising_edge(myclk)) then
    [insert your logic involving myvar, etc., here]
 end if;
end process MY_PROC;
```

Note the standard reset signal, *myrst* and its syntax. This block and its elsif can be omitted, keeping only the contents of the "elsif". The part in parentheses after the key-word "process" is called the sensitivity list. The process will activate only when a signal in the sensitivity list changes.

Note that (myclk'event and (myclk = '1')) is equivalent to (rising_edge(myclk)) and that (myclk'event and (myclk = '0')) is equivalent to (falling_edge(myclk)) [XILXST].

Typically, "non-blocking" assignments are used in signal processing, which means that all assignments in the process are done simultaneously from the input data when the clock edge occurs. The syntax for a "non-blocking" assignment is "<=". In a "blocking" assignment, one assignment is performed first before the next is done; the syntax for this is ":=". Blocking assignments are often used in logic "functions," which implement combinatorial logic.

The value of a signal can be determined in complex ways and set in its own process, but write conflicts will occur if a signal is controlled from more than one process. However, signals can be used for "read" purposes in different processes. If it is attempted to reset signals using a non-reset signal in the standard reset syntax, the signal will be interpreted as a reset signal and will be connected to the design reset, causing write conflicts. Using the syntax in the Xilinx manuals is mandatory if one wishes to accomplish one's intention with Xilinx VHDL [XILXST]. As an example, the "case statement" cryptographic S-box had to be put in a process to be properly recognized as a ROM, or else the Xilinx synthesis tool would interpret the block as an asynchronous RAM and remove all but one of the required instances.

For another example, the VHLD for a dual-port RAM is shown:

```
process (clk)
begin
  if (clk'event and clk = '1') then
    if (we = '1') then
      RAM(conv_integer(a)) <= di;
    end if;
    read_a <= a;
  end if;
end process;
do <= RAM(conv_integer(read_a));
```

Note: we: write enable, a: address, do: data out [XILXST].

This is the only type of syntax that will be recognized by the Xilinx synthesizer as a dual-port RAM.

Fortunately, assignments between vectored signals declared as *big-endian* and *little-endian* (see section 3.1.7.1., "Core Design and other Modifications" for explanation) are straightforward; e.g.:

```
            signal key : STD_LOGIC_VECTOR (127 downto 0);
            signal slv_reg1 : std_logic_vector(0 to C_DWIDTH-1);
            key(127 downto 96) <= slv_reg1;
```

bit 0 of *slv_reg1* will be assigned to bit 127 of the variable *key*, and so on to bit 31 (i.e., C_DWIDTH-1) of *slv_reg1* assigned to bit 96 of *key*.

Note that range assignment is flexible; e.g.:

```
            signal w0, w1, w2, w3:  STD_LOGIC_VECTOR(31 downto 0);
            Type kbarray is array (10 downto 0) of STD_LOGIC_VECTOR(127
downto 0);
            signal kb: kbarray;
            w3 <= kb(conv_integer(read_kb))(127 downto 096);
```

and can be applied to the output of arrays, as shown.

3.1.6. Working with the Xilinx Tools

When building a module or an entire project in the Xilinx ISE, the simulation stages available are behavioral, post-translate, post-map and post-PAR (place and route), reflecting the build stages: design entry (behavioral simulation), synthesize and translate (post-translate), map (post-map) and PAR (post-PAR simulation) (Note that Xilinx uses the US spelling of "behavioural"). In synthesis, the HDL is recognized and represented as logic components such as AND gates, counters, ROMs, and so on. "Translate" is a technical stage in which the "netlist" (the list of circuit connections) format is converted. In map, the logic components are expressed using the type of logic cells available in the FPGA, which contain two LUTs per cell, or "slice", a multiplexer and two flip-flops. In "place", specific logic cells in the FPGA are chosen for the mapped content, and in "route", the FPGA switching fabric, which makes up three-quarters of the FPGA, is set with the necessary electrical connections.

BRAM resources are automatically used for ROMs by XST (the Xilinx Synthesis Tool). When necessary, separate vectors can be used to contain data instead of using

ROM syntax, and also distributed LUT (Look-Up Table) ROMs, available from the Templates in the ISE, are available.

When using the EDK and making changes to the design [XILML403T], "Project → Clean All Generated Files" must be done, or the changes will not be recognized.

Simulation has some differences from implementation – the Unisim library is required for post-PAR simulation, but only required for implementation in modules containing the LUT ROMs –the Unisim library declaration is shown:

```
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
```

The following IEEE libraries were required:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

However, the following two have to be commented-out in order to support conv_integer for array index addressing (conv_integer is used to converting std_logic to an array index) when that is used:

```
Use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
```

The file containing input stimulus to feed to the design for simulation is called the "testbench". Input setup times and output valid delay settings in the testbench must equal or exceed the "minimum arrival time before clock" and "maximum output required time after clock" specified in the synthesis report or simulations will not succeed.

Mentor Graphics ModelSim XE III 6.1e starter edition was used for simulation, which was quite reliable, and allowed examination of interior design signals, which was not possible with the simulator provided with the Xilinx ISE. Moreover, simulation of advanced build stages, such as post-PAR, in the Xilinx software was very much "broken" and unusable. The ISE allows selection of different simulators when the Xilinx project is created, and when working with the project. A VHDL vector is normally represented as a single line in simulation output and can be set for display or input in various radices (bases), such as hexadecimal, decimal, octal and binary.

3.1.6.1. Simulation Test Methodology

Simulation results, of the encryptor and decryptor core designs, were initially verified using an available Javascript implementation that showed the results and intermediate values of the transform and its inverse [STY2006]. Due to problems encountered in higher orders of simulation, such as post-translate and post-map, in which portions of the key expand values were being apparently duplicated, due to not making the S-box synchronous, and using insufficient "input setup time" and "output valid delay" timing specifications in the testbench, respectively, the arbitrarily-chosen and somewhat redundant key of 466E6172 676C6572 20426C61 67686572 (hex.), which was the ASCII (ANSI Standard Code for Information Interchange – ANSI: American National Standards Institute) for "Fnargler Blagher", was changed to 01020304 15161718 292A2B2C 3D3E3F42 (hex.), which has no planned ASCII meaning, for simulation, so that each byte would be unambiguously unique.

3.1.6.2. Software Loading, Running and Debugging

Xilinx includes the GDB (GNU DeBug – GNU: "GNU is Not Unix") software. Before invoking that, XMD (Xilinx Microprocessor Debug) must be started and connected to the Power PC target (see [XILEST], chapters 10-12). The icon to launch XMD is visually identifiable as a bug in a box. Connection is automatic. Experience showed however, that it was necessary to disconnect and reconnect in order for GDB, especially, to successfully operate: enter the command "disconnect 0", and then "connect ppc hw" in the XMD window (the quotes are delimiters here and not included in the command). XMD appears similar to a DOS (Disk Operating System) window and DOS commands will work. The user can change directories, for example. To test the compiled and linked software without using GDB, the command "dow executable.elf" can be used – but it is necessary to change to the directory containing "executable.elf", first. "run" launches execution; "stop" stops it. This is useful since the debugger slows execution so much that RS232 communication is prevented. If debugging is desired, GDB can be launched by clicking the icon next to the XMD icon, which is of a bug (not in a box). This method is also useful for loading larger amounts of code to SRAM, when the code size exceeds that available in BRAM.

3.1.7. AES Design Done in this Work

3.1.7.1. Core Design and other Modifications

Figure 13 shows the modules and their hierarchy in the encryptor and decryptor design, modified from that found in [USS2002]: the top-levels were renamed "user_logic" as required by Xilinx, and the higher-level wrappers needed for bus interfacing were added.

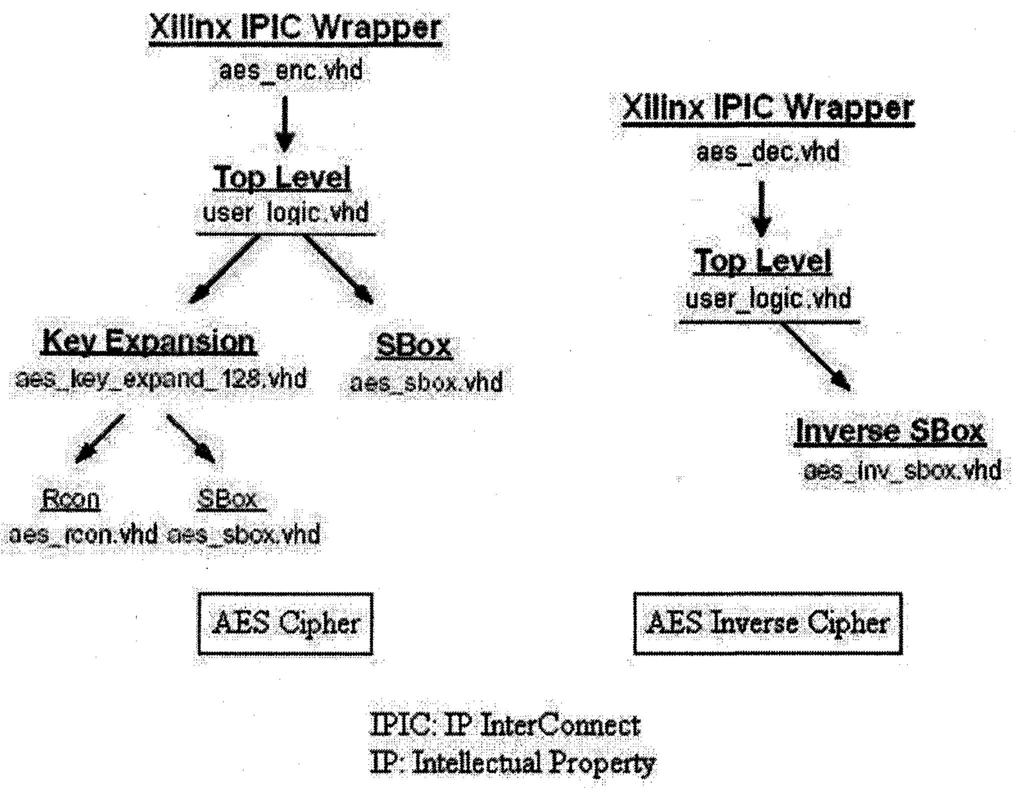


Figure 13. AES encryptor and decryptor modules and hierarchy

Note that the key expand module was not included with the inverse cipher, or decryptor, in order to allow both modules to fit in the XC4VFX12 FPGA. As detailed after, key expand readout from the encryptor and input to the decryptor, for its storage, was implemented.

Thirteen 32-bit *slv_regs* were specified for each core – *slv_reg0* for control signals, *1-4* for the key input, *5-8* for the plain/cipher text and IV input and *9-12* for cipher/plain text output and other output.

Big-endian data orientation was maintained in the design, as shown in Figure 14. In this orientation, high-order data is located in the low-order ends of the registers, since the general standard for numeric notation is to read the high order data first, left to right and top to bottom, and memory maps are generally presented low-order to high-order, left to right and top to bottom. In *little-endian* data orientation, the low-order data is located in the low-order register bits.

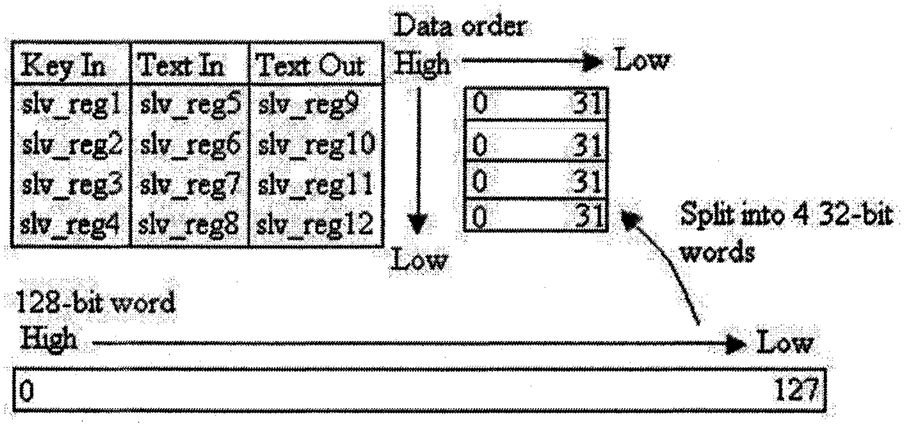


Figure 14. Big-endian data orientation and core IO register usage

Extensive modifications to the original code were required in order to make it work on a Xilinx FPGA and to interface it to the OPB. First, however, the original Verilog code was verified in behavioural simulation. The decision was made to translate the Verilog into VHDL, to avoid having to rely on mixed-language support. Then the *done* bit behaviour was revised so that it would remain set for the software to read. This was accomplished by removing a clearing condition; instead of the former:

```
            done <= (not (dcnt(1) or dcnt(2) or dcnt(3)) ) and dcnt(0);
            done <= not (dcnt(3) or dcnt(2) or dcnt(1));
```

the latter was used, where *dcnt* is a four-bit counter, counting down. This VHDL syntax selects individual bits of a vector. When the "and" of the zero-order bit was included, *done* would be cleared on the clock rising edge after *dcnt* reached zero, making *done* a pulse with a length of only one clock cycle. *ld_r* was used in order to clear *done* for another transform, where *ld_r* stands for "real load" and is timed always to occur on the same clock phase (see the section on "autoload"). A signal called *dcntbits* was defined and set to "1" for as long as *dcnt* is not "0", and used to enable updating of *text_out* while the rounds progress. When *done* is changed to "1", *dcntbits* is changed to "0" and *text_out* stops changing so that it can be read.

All major blocks of the design had to be placed within VHDL processes of the type shown before, and assigned a clock signal so that they would be synchronous. This was a major step in success of simulation beyond behavioral. Asynchronous latches are difficult to simulate, because their timing has to be followed and correctly predicted using wire and component delays, whereas a clocked flip-flop's state can be processed by the simulator at the clock transition times.

Amalgamating the shift rows step was possible because the signals being transferred to the next step only needed to be rearranged. This freed a clock cycle in the timing plan. Saving a step in each round allowed the use of two clock phases for the more computationally-intensive mix columns and inverse mix columns functions. The DCM was used since it could produce three additional clock pulses of the same frequency as the system clock, at each multiple of a 90-degree phase delay. Since the system clock was 100 MHz, its period was 10 ns, making each phase 2.5 ns apart. The XC4VFX12 FPGA on the ML403 board was speed grade 10, the slowest of grades 10, 11 and 12 [XILRIV], and the design could not achieve the timing of 2.5 ns between round steps. Therefore a second DCM was used to produce the additional three phases and the first was used to divide the clock by two. Four phases of a 50 MHz, 20 ns clock are 5 ns apart, and the design was able to meet timing. The timing plans were carefully worked out so that required data was available by the time of occurrence of the clock edge used by the process requiring the data.

Figures 15 and 16 show the timing plans developed for the encryptor and decryptor. In the encryptor timing diagram, notice the sequentially-placed *sa* (add-round-key step), followed by “sub bytes”, then “mix columns”, after two phases of time so that the mix columns logic would have 10 ns to propagate before being required at its 180-degree clock phase. The key expand value, *w*, is made ready one phase prior to being needed for the add-round-key step. Within the key expand module, *rconout* is made ready well in advance of the time that it is needed for inclusion. The *done* signal is timed to go high one phase after *text out* is ready.

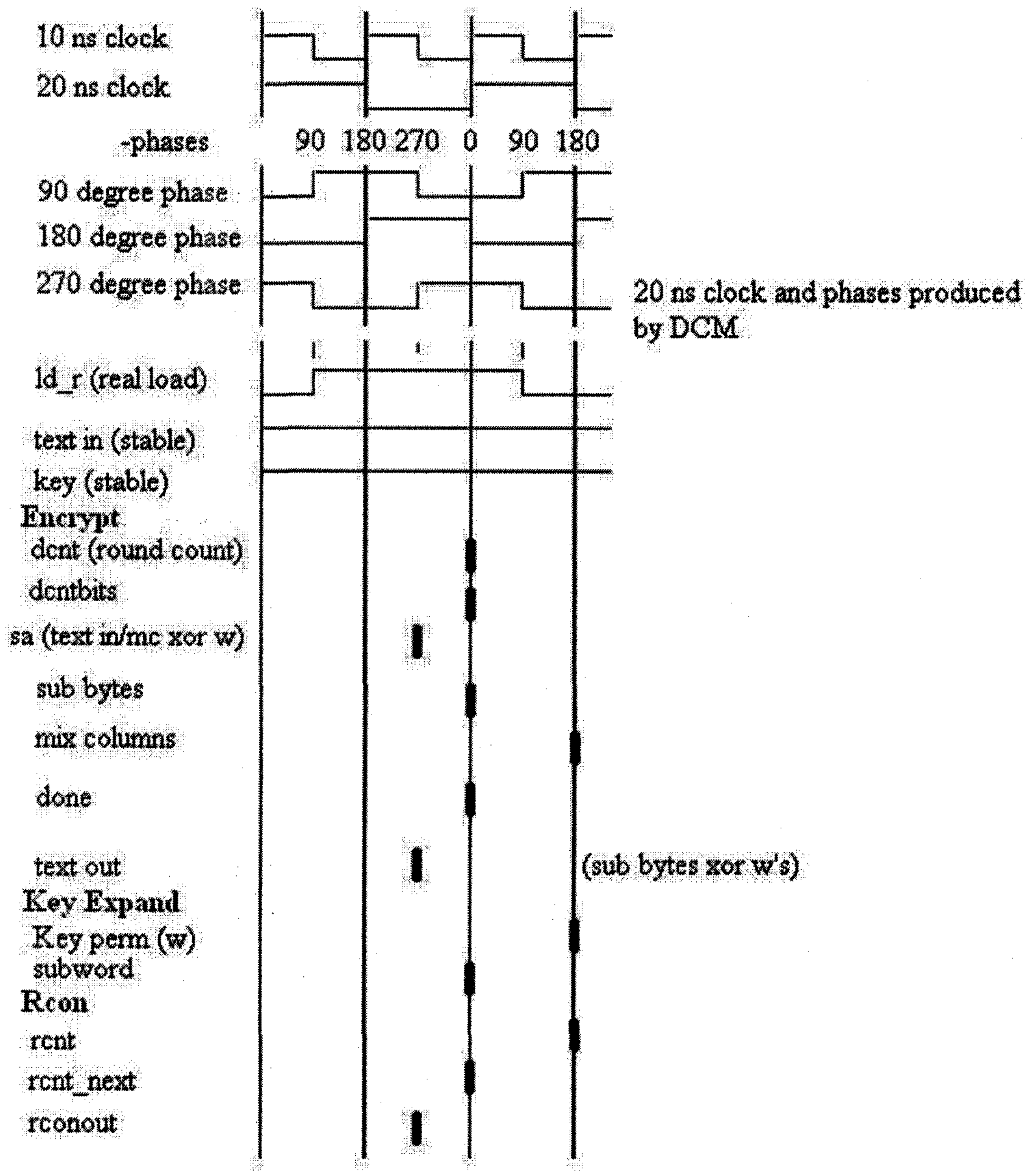


Figure 15. AES encryptor timing plan

ld_r is positioned to begin always on the same clock phase (see the section on autoload). When *ld_r* is high, *text_in* is used to XOR with the key expand value, whereas after that the key expand value is XORed with the final step (mix columns) of the previous round.

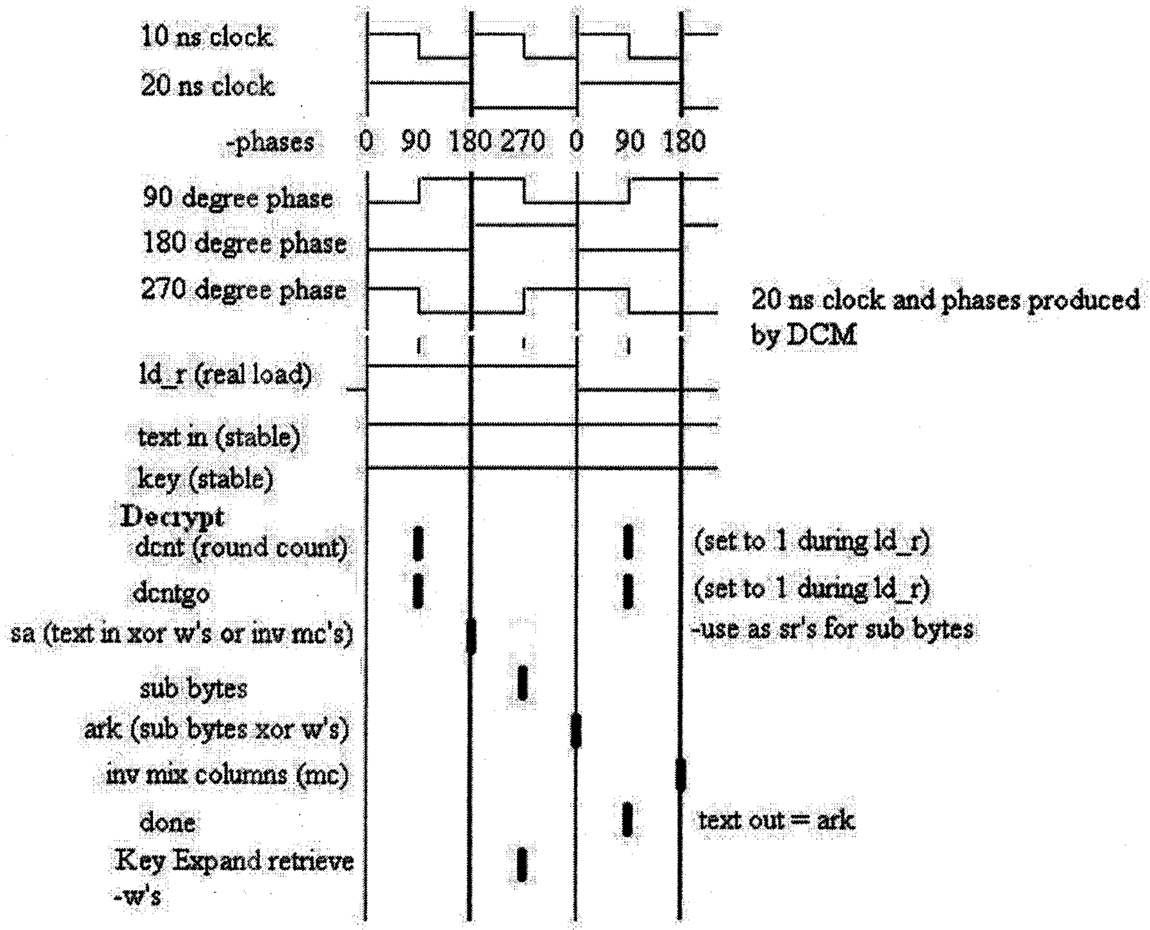


Figure 16. AES decryptor timing plan

In the decryptor, the shift rows operation occurs before the "sub-bytes", but this operation is made implicit from the *sa* data by simple arrangement in the VHDL assignment statements (as in the encryptor). The key expand value has to be retrieved for use; it cannot be generated concurrently as in the encryptor, since the final key expand value is needed first, in the decryptor. The key expand values are loaded into the decryptor before any decryption transform is initiated. By initializing the step counter *dcnt* to 1 during *ld_r*, and also initializing it to zero prior and by default, the number of clock cycles required for decryption was reduced from eleven to ten – note that *dcnt* actually is counted upwards in the decryptor. It should be possible to adjust the encryptor timing plan and logic in a similar manner in order to reduce the time required by a clock cycle.

The encryptor (aes_enc), along with the other supporting modules, used all assigned BRAM resources. Therefore, in the decryptor (aes_dec), separate vectors to store the key expand values were used, instead of the dual-port RAM structure that was otherwise available, and Xilinx distributed LUT (Look-Up Table) ROMs, available from the Templates in the ISE, were used for the S-boxes. Modifying the libraries to be included in the project for the decryptor array declaration when the dual-port RAM was being used, seemed to cause the *slv_reg* contents to be displayed backwards in post-translate and later simulations. Declaring them as big-endian in the decryptor was the work-around.

The slave register interface described before was modified in its write process so that selected bits could be written to the slave registers by the core while the slave registers were not already being written to by the bus. It is not possible to write, from the core, a bit in a *slv_reg* that is regularly being driven from the bus, originating from software, since the bit will be overwritten and its value will not change. However, it is possible to write to bits individually, thus choosing the role of each bit. The only bit required in *slv_reg0* as an output from the core is the *done* bit; the rest of the control bits defined in this work for *slv_reg0* are inputs. Therefore the "case" statement in the *slv_regs* write process was separated using "if" statements into groups of *slv_reg0* on its own, *slv_regs9-12*, which are also used as outputs, and *slv_regs1-8*, which are only used as inputs. The condition used in the "if" statements is the write select to the registers from the bus; in the "else", writes from the core were placed. Bit 0 of *slv_reg0* was assigned as the bit to which *done* from the core is written, and is set when *done* is "1" and cleared when *ld_r* is "1". The clock to the *slv_reg* write process is the bus clock undivided; the pulses from the core, being twice as long, would therefore always occur for enough time to allow a bus clock rising edge to occur. The bits which are defined as control inputs from *slv_reg0* are assigned asynchronously in the core from their *slv_reg0* bit positions. In the block involving *slv_regs9-12*, those registers are assigned from *text_out* when *done* is "1".

3.1.7.2. Additional AES Implementation Features

3.1.7.2.1. Autoload

Originally in the development of the encryptor and decryptor, a bit from *slv_reg0* was defined as the "load" bit and *ld_r* would be set to "1" on its clock, if "load" was high. This caused an avoidable delay in setting the bit in software and then clearing it – it would have to be cleared or *ld_r* would repeat – in fact, *ld_r* always repeated when this method was used until the "load" bit was cleared, because the speed of the transform, of eleven clock cycles, was more than twice as fast as the software could set and clear the "load" bit. Instead, a signal called *start_load* was defined which is set to "1" when *slv_reg8* is written, and cleared when *ld_r* is detected to be set to "1". Therefore *start_load* turns on at some arbitrary time and waits for the *ld_r* process to detect it on the chosen clock for *ld_r*, and then turns itself off. In the *ld_r* process, *ld_r* is turned off on its own chosen clock rising edge if it is set, thus ensuring that it stays high for only one of its clock cycles. Thus the write to the final *text_in* register neatly triggers one *ld_r* pulse and one transform. This could also be useful if DMA (Direct Memory Access) ability is added to this core at some future date. DMA ability is available [XILOPBIP2H], and was investigated as part of this work, but adding it is a considerable undertaking.

3.1.7.2.2. Key Expand Readout, Storage and Readback

A method to induce the encryptor to produce and hold the key expand values one at a time to be read by software, was developed in a resource-efficient manner by utilizing control bits connected from *slv_reg0*. Since software sets the control bits, a slice-consuming VHDL process to generate the control bits as a signal did not have to be used, and the control signals need only be read by the core.

One control bit, *krd* is used only to signal that the key expand module is in "key expand readout" mode. If it is set, *slv_regs9-12* are set from the key expand value, not from *text_out*. A second control bit, *kstep*, is used to proceed to the next key expand value on the next clock when it is high, or to hold the current key expand value when it is low. A third control bit, *kstepend*, was found necessary to prevent the state variable, *rcnt_next* from free-running (repeatedly causing its own update) in the "rcon" (round constant) module. It was necessary in the end to add a small process to time the exact transition of *kstep* and *kstepend* to the clock pulse before *rcon* is generated for its use in the key expand module. Then, as long as *krd* is "1", setting *kstep* to "1", then setting *kstepend* to "1" via software, would cause the subsequent key expand value to be available for the software to read from *slv_regs9-12*. Simultaneously clearing *kstep* and *kstepend* is necessary to prepare for the generation of the next variable. To begin the whole process, *load* (via autoload) must be done after setting *krd*. The first key expand value is the key itself, and is read before using *kstep*. When all eleven key expand values have been read, *krd* must be cleared in order to use the encryptor in its regular transform mode.

A random-access protocol was added to the decryptor to store the key expand values. Four bits of *slv_reg0* in the decryptor were assigned for selection of eleven internal storage vectors, *kcnt*, internally, and a VHDL case statement was used to select the storage vector to receive the contents of *slv_regs1-4* (the key input slave registers) when *kld* ("keyload", from *slv_reg0*) is "1". When *kld* is "0", *dcnt* is used to address the particular key expand value when the decryptor is operating in its regular transform mode; as noted before, *dcnt* is actually counted upwards from zero in the decryptor; note that this means that the final key expand value (in encryption order) is located in address zero of the internal decryptor storage and thus the entire set of the eleven key expand values is stored in the decryptor in reverse order.

Also when *kld* is "0", a *kbrden* ("key buffer read enable") signal from *slv_reg0* is used to select key readback, which was used as a confidence test when the decryptor was under development. An asynchronous assignment was used to determine the random-access address, *read_kb*, used to obtain the selected key expand value when *kld* is "0", since *dcnt* is used when *kbrden* is "0", for the regular transform mode, and *kcnt*, determined from the random-access address set in *slv_reg0*, is used when *kbrden* is "1":

```
read_kb <= ((not(kbrden & kbrden & kbrden & kbrden)) and dcnt)
           or ((kbrden & kbrden & kbrden & kbrden) and kcnt);
```

Note that "&" is a concatenation operator, making a four-bit vector from *kbrden* for selection of either *dcnt* or *kcnt*. If *kbrden* is set in *slv_reg0*, the corresponding key expand value, according to the number set in the *kcnt* bits set in *slv_reg0*, can be read from *slv_regs9-12* under software control.

3.1.7.2.3. Cipher-Block Chaining (CBC) Mode

In a block cipher such as AES, the same block encrypted with the same key always gives the same ciphertext. This mode of encryption is known as "Electronic Code Book" mode, and is obviously cryptographically weaker than if the plaintext could be "salted" in some continually-varying way – in cryptography, "salting" the message means to add unrelated content before encryption in order to attempt frustration of cryptanalysis.

In Cipher-Block Chaining (CBC) mode, the plaintext is XORed with the output of the previous encrypted block before being fed to the encryptor core for encryption. In decryption, each decrypted block is XORed with the previous block of ciphertext to reveal the plaintext. This is possible due to the property of the XOR operation that it is its own inverse. When the first block in an encryption or decryption sequence is processed, an "IV", or "Initial Vector" is used in place of the ciphertext of the previous block.

This was added as a non-optional feature – in the encryptor a one-time *done* pulse was added during which time the ciphertext is transferred to the *IV* vector, which was reused for this purpose as well as for the actual IV, for simplicity. In the decryptor, two such pulses were required; in the first, the output is XORed with the *IV* to form the plaintext; in the second, the *IV* is updated from the current ciphertext.

To enter the *IV*, an *ivload* bit was defined in *slv_reg0* for both the encryptor and the decryptor. When "1", the contents of *slv_regs5-8* are copied to the internal *IV* vector. An *ivrdback* bit was defined in *slv_reg0* for the decryptor. When set, the current value of the *IV* is copied to *slv_regs9-12* for read-back. This feature was not added to the encryptor, due to space concerns.

3.1.7.2.4. Timing Diagnostic Output for Test

In each of the encryptor and the decryptor, a "test process" was added in which a signal, called *loadtodone* is set high when *ld_r* is detected and cleared when *done* is detected. This signal was defined as an output in the formal parameter list of the module and passed up to the outer wrapper where it in turn was defined as an output in its formal parameter list. This change was the only change necessary to the outer wrapper. In the EDK, this signal from the encryptor was connected to pin AF24 of the FPGA, which is connected on the ML403 board to J6 pin 64, and this signal from the decryptor was connected to pin AA24 of the FPGA, which is connected to J6 pin 2. As the top side of the ML403 board is viewed so that the large, gold-coloured "Virtex V4" and "Xilinx" labels are the correct way up for reading, J6 is the large header on the far right of the board. It is a double-column header; the third column of pins placed to the left of J6 to make it appear like a three-column header is actually J3. There is another three-column header to the left of J6 and J3, with some PCB (Printed-Circuit Board) space visible between it and J6 and J3. Pin 2 of J6 is at the upper right of the header and pin 64 is at the bottom right; both are corner pins, making attachment by an oscilloscope probe as easy as possible. The comparatively large, threaded brass cable connectors nearby on the PCB make a useful ground connection for the ground alligator clip of the oscilloscope probe. The "net" connections to the FPGA pins from the design can be found in "system.ucf" in the \data\ directory of the EDK project and are shown here:

```
Net aes_enc_0_loadtodoneout_pin LOC=AF24;
Net aes_dec_0_loadtodoneout_pin LOC=AA24;
```

In an earlier version of the AES core, an output signal set from the "load" *slv_reg0* bit was routed to one of the pins noted before, as well as *loadtodone*, when only one core at a time was being tested in the FPGA, and before the autoload modification was done. This allowed the time taken to set and clear the load bit via software to be measured using an oscilloscope.

Figure 17 shows the locations of the control bits in *slv_reg0* in the encryptor and decryptor:

Encryptor - slv_reg0

LE	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
BE	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

										I		Ke	Kr	Ks	L	D
LE	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
BE	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

D - Done, L - Load (Go) (no longer used), I - IV load
Key expand mode: Kr - Key read (Key expand mode), Ks - Key step, Ke - Key step end (Key expand values are read from slv_regs9-12)

Decryptor - slv_reg0

LE	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
BE	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

					K3	K2	K1	K0	Ir	I		Kb	Kl		L	D
LE	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
BE	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

D - Done, L - Load (Go) (no longer used), I - IV load, Ir - IV readback
Key storage and read back: Kl - Key load, Kb - Key readback, K3-K0 - Key schedule index (reverse of encryption order)
LE - Little Endian, BE - Big Endian

Figure 17. Control bit locations in slv_reg0 for both AES user peripherals

Bit 02 (BE – Big-Endian) in the decryptor was for "Key Expand Done", but that is not used since the key expand values are supplied to the decryptor.

3.1.7.2.5. A "Stepper" Version of the Decryptor

Blocks of logic were added to the decryptor to enable their corresponding clock phases in the core when the software would pulse a bit in *slv_reg0*. A counter was used to count through the clock phases, incrementing once each time the bit in *slv_reg0* went high, and an enable for each phase was timed to be "1" when the rising edge of that phase was to occur. These enables were added as a condition to each process that uses a clock phase. The output of each resulting step of the three steps per round was placed in *slv_regs9-12*, resulting in a readout of the current state for each step of each round. This was written as a debugging check, and remains of theoretical interest. The method used, of implementing a separate process to enable each clock phase with its own pulse, was costly in terms of FPGA fabric and led to the realization of the simpler method used in the key expand generation feature.

A simple CLI (Command-Line Interface) program was written to write the key and the plaintext to the decryptor peripheral's slave registers, pulse the load bit, then repeatedly pulse the step bit in *slv_reg0*, and read *slv_regs9-12*, and send their hex-ASCII-encoded values for display via RS232 when the user presses a printable character key on the computer keyboard – see the section after, "CLI and Simpler Programs" for more description of this general programming technique.

A bug was found in which the initial key expand value used was not correct due to the repeated *ld_r* during the "load" pulse causing the array index used to access the array of key expand values being incremented; the index was not being initialized along with the value of *dcnt* when the latter was initialized to zero. Due to arbitrary key expand values being left from previous rounds of operation, the "stepper" would eventually produce a series of correct output values. This knowledge was used to correct the decryptor's operation by initializing the key expand values always from the first element of the key expand array rather than relying on the initialized array index; in the subsequent round, *ld_r* occurred, and the index was always being set to the second value. This fix was added to the "stepper", as well, which should correct its operation, but the "stepper" was not subsequently tested.

3.1.8. Test and Demonstration Software

3.1.8.1. CLI and Simpler Programs

Numerous small CLI programs were written to be loaded into the ML403 board to test the AES cores, beginning with one to successfully write and read back the slave registers. The standard program, "Hyperterminal", included with all versions of Windows, was used on the PC to view the RS232 output from the board and send user keystroke data. Xilinx provided easy-to-use function calls to use the UART and the RS232 port, such as inbyte() and outbyte() – the parentheses following the identifier is syntax that indicates a C function, and, in actual implementation, may or may not contain a list of parameters being passed to the function. In addition, Xilinx provided a written "TestApp_Memory" C program that tested memory and reported via RS232, making a useful test of system “liveness”. Next, one was written to write the key and plaintext and to turn the “load” bit on and off, loop to wait for the “done” bit, and then read *slv_regs9-12*; later, this was also written to test the decryptor. These confirmed basic operation of the user peripheral, once the same cryptographic output appeared that was seen in simulation.

Functions were written to encode and decode numeric values and strings, because raw binary data should not be sent over an RS232 link, for the reason that some numbers are RS232 control codes that can stop the link from apparently functioning. Any numbers sent over an RS232 link that are not intended as control codes should be encoded as printable ASCII. The encode function, called “hex-ASCII encoding” in this work, interprets numeric data as hexadecimal and creates the printable characters representing the hexadecimal digits. The decode function takes a string of characters that represent hexadecimal digits and converts it into the numeric data that was represented. Note that an ASCII character requires eight bits and a binary hexadecimal digit requires only four bits, meaning that this type of encoding doubles the storage space required (when the data is not immediately decoded upon reception). When the program in the board sends numeric data for display, it encodes the digits, and when it receives numeric data typed by the user from the keyboard, it decodes the characters received, to determine the numeric value of the data.

Versions for each of the encryptor and the decryptor were written to loop, setting and clearing the “load” bit, when that was used, looping to wait for the “done” bit, and then repeating. This was used to determine the maximum possible processing speed available using this overall design, and did not even include any RS232 output, being intended for measurement of the diagnostic output signals using an oscilloscope. Later, these programs were revised to use autoload and to include a full write of the input and a full read of the output.

A full CLI test program was written to exercise all features included with the cores; this program evolved as features were added, originating as a version to test only the encryptor. Its basic design is an endless loop with no exit criterion, containing a wait for a character to be received via RS232, following which a C "switch" statement (which is the C-language equivalent of the VHDL "case" statement) was used to act on valid command characters received. An infinite outer loop is the basic computer operating system in its simplest form. In it, the list of tasks to be done is placed, to be processed in a round-robin fashion – i.e., repeatedly. Notably, this test program echoes the key expand values one at a time via RS232 when the key is loaded, as they are copied to the decryptor core. The inbyte() function waits for input if none is available, making it useful for halting processing to allow the user to view the output generated. Typically, a string such as "Press any key to continue" is first sent for display, following which the wait-for-input function call is invoked. The character typed by the user is generally not otherwise used in this specific situation.

Since Cipher-Block Chaining mode was added to the AES cores, the IV is loaded to both by this test software, and the encrypted block can be seen to vary, following which it is always decrypted to the correct plaintext, upon repeated test encryptions.

Commands available in this program are: "i - enter the IV; v - view decryptor's IV; k - enter the key (and do key exp); p - enter the plaintext; e - encrypt and decrypt; x - Display the decryptor's key expand values". The command characters are made case-insensitive in the switch statement by using pairs of case statements for each block. This saves the memory required to add in an extra library; moreover the library containing the standard C "toupper" function was not found in the GNU libraries provided by Xilinx.

This basic design was used for the ML403 board code that works with the demonstration GUI: the demonstration GUI sends the individual command characters.

3.1.8.2. The AES Demonstration GUI, “AESfile”

The demonstration GUI sends the individual command characters to the board and then has a programmed dialog with the board until the specific dialog for that command character finishes.

In this demonstration, a text file with a given name, “plaintextin.txt” is sent block-by-block to the board and encrypted. Each encrypted block is hex-ASCII encoded by the board and sent back to the PC for storage in a text file, “ciphertextout.txt”. The GUI-board system also does decryption by reading a file called “ciphertextout.txt” and sending the block of hex-ASCII-encoded bytes back to the board, which sends back the block of sixteen characters (128 bits); the GUI saves these in a file called “plaintextout.txt”. Note that the block of hex-ASCII-encoded bytes requires 32 ASCII characters, one to represent each hexadecimal digit in 128 bits. The ASCII values of the block of sixteen plaintext characters are treated as numeric data by the board for encryption purposes. For decryption purposes, the 32 hex-ASCII-encoded characters received are first decoded, by the board, to sixteen bytes of numeric data.

3.1.8.2.1. The ML403 Board Code

When ‘k’ is received as a command, the board then expects the hex-ASCII-encoded key from the GUI, which it decodes, writes to the encryptor, does key expand and writes the key expand values to the decryptor.

When ‘i’ is received as a command, the board then expects the hex-ASCII-encoded IV from the GUI, which it decodes and writes to the encryptor and the decryptor.

When ‘e’ is received as a command, the board encrypts the IV using an IV of zero, sends it to the GUI, updates the encryptor with the IV, and then loops: receiving blocks from the GUI, encrypting them and sending them back. It stops when the ASCII character “ETX” (End-of-Text) is received and includes it as the last character to encrypt, padding any remnant of the last block with zeroes.

When ‘d’ is received as a command, the board first receives the encrypted IV from the GUI (hex-ASCII-encoded), decodes it, and decrypts it using the key and an IV of zero. Then it updates the decryptor’s IV with the decrypted IV. Then it loops, receiving hex-ASCII-encoded blocks of ciphertext from the GUI, decoding them, decrypting them, and sending them back. When the ETX is found, the loop exits.

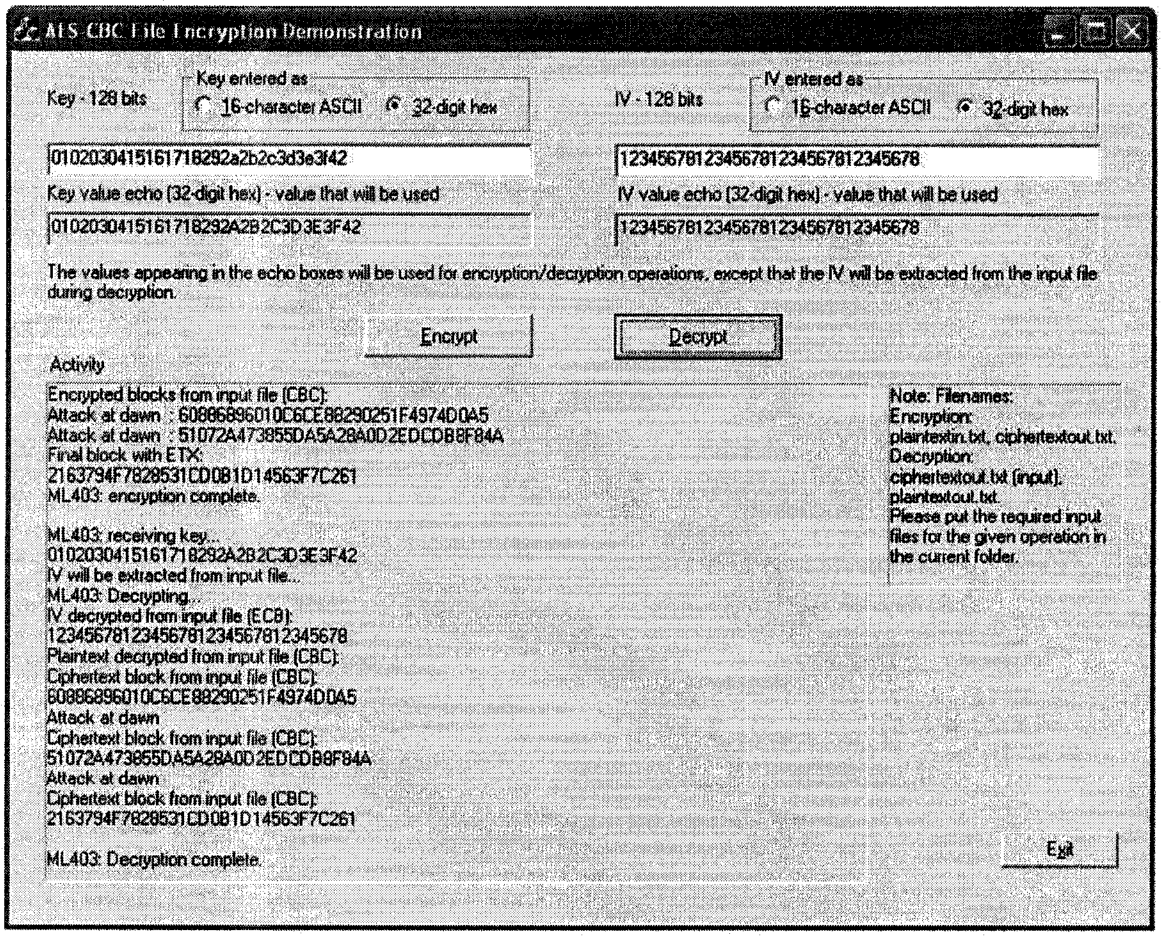
3.1.8.2.2. The PC Demonstration GUI, “AESfile”

A package written for MSVC++V6, to do serial IO, was located [KLE2003], and incorporated into a MSVC++V6 project. Its function calls (“methods” in C++), provided the ability to communicate with the ML403 board via RS232.

The GUI, named “AESfile,” provides two “edit boxes” for entry of the key and IV, and two “static text boxes” next to these for display of the resulting numeric key and IV, since, if characters are typed into the “edit boxes”, their ASCII values are used as the numeric cryptographic data. Radio buttons are provided to allow interpretation of the user’s entry in the edit boxes as either hex. digits or ASCII characters.

Two buttons are provided, one to encrypt and the other to decrypt. A large static text box is provided in which activity echoing is shown, such as the data being encrypted or decrypted. A function was designed and implemented to add characters to the activity display and delete the oldest characters when the text box becomes full, giving the appearance of “scrolling”. When a button is clicked, the files are read and the board is commanded to encrypt or decrypt, via the procedure described before. The IV is encrypted and added to “ciphertextout.txt” so that only the correct key is needed to decrypt an encrypted file. Encrypting the IV was realized to be somewhat of a cryptographically faulty idea, as described before (see section 1.2.4.2. “Encapsulating

Security Payload Protocol"). Figure 18 shows a "screenshot" of this demonstration in action.



AES-CBC File Encryption Demonstration

Key - 128 bits — Key entered as: ( ) 16-character ASCII [✓ 32-digit hex]

IV - 128 bits — IV entered as: ( ) 16-character ASCII [✓ 32-digit hex]

```
0102030415161718292a2b2c3d3e3f42
```

Key value echo (32-digit hex) - value that will be used

```
0102030415161718292A2B2C3D3E3F42
```

```
12345678123456781234567812345678
```

IV value echo (32-digit hex) - value that will be used

```
12345678123456781234567812345678
```

The values appearing in the echo boxes will be used for encryption/decryption operations, except that the IV will be extracted from the input file during decryption.

Activity

```
Encrypted blocks from input file (CBC):
Attack at dawn : 6088689601 0C6CE88290251F4974D0A5
Attack at dawn : 51072A473855DA5A28A0D2EDCDB8F84A
Final block with ETX:
2163794F7828531CD0B1D14563F7C261
ML403: encryption complete.

ML403: receiving key...
0102030415161718292A2B2C3D3E3F42
IV will be extracted from input file...
ML403: Decrypting...
IV decrypted from input file (ECB):
12345678123456781234567812345678
Plaintext decrypted from input file (CBC):
Ciphertext block from input file (CBC):
60886896010C6CE88290251F4974D0A5
Attack at dawn
Ciphertext block from input file (CBC):
51072A473855DA5A28A0D2EDCDB8F84A
Attack at dawn
Ciphertext block from input file (CBC):
2163794F7828531CD0B1D14563F7C261

ML403: Decryption complete.
```

Note: Filenames:
Encryption:
plaintextin.txt, ciphertextout.txt.
Decryption:
ciphertextout.txt (input),
plaintextout.txt.
Please put the required input files for the given operation in the current folder.

Figure 18. A "screenshot" of the "AESfile" demonstration

Test methodology for this, to determine the encryption speed, consisted of timing the file encryption and decryption processes as they progressed, using a digital watch that counts seconds. Due to expected delays caused by the RS232 transmission and the text display to the activity window, a more accurate timing method did not seem justified.

It seemed that other programs installed on the PC would hold access to the serial port and prevent communication from working. One such seemed to be the Tektronix PC Communications software, whose use is described after for obtaining images from the oscilloscope. Another seemed to be MSVC++V6; it was necessary to repeatedly begin and stop debugging in its IDE (Integrated Development Environment) in order to obtain access to the serial port – which, at least, was faster than rebooting the PC. "Hyperterminal" could be used to determine if access to the serial port could be obtained, as it would report a dialog box if it could not obtain access to the serial port; however, that indication was not reliable when the Tektronix software was seemingly preventing access. "Hyperterminal" must itself be "disconnected" using its menu option or exited if it is required to use another PC program that accesses the serial port.

3.2. Design of a Combination LFSR-CASR Pseudo-Random Number Generator

3.2.1. Selection of the Base Design

The availability of random number generators was somewhat limited; on the "Open Cores" website [OPENCORES], there were only two selections available. The Verilog/SystemC LFSR-CASR (Cellular Automata Shift Register) RNG was chosen for its claimed good statistical properties [VILL2005]. The other, a library of RNGs, was indicated as not being synthesizable [DRA2004].

3.2.2. Description of the Tkacik-Villar LFSR-CASR PRNG

This PRNG was made available in SystemC and Verilog, and based on the design by Thomas E. Tkacik [TKA2002].

The LFSR contains bits numbered from 0 to 42; each clock, each of bits 0, 19 and 40 are replaced by their XOR with bit 42, then the contents of the LFSR is rotated one bit to the higher direction: bit 0 becomes bit 1, and so on to bit 42 becoming bit 0. The resulting output has a cycle length of (2^{43})-1 and a bias of 2^{-43}.

The CASR contains bits numbered from 0 to 36; each clock, each bit is replaced by the XOR of its two neighbour bits, with bit 36 and bit 0 being considered neighbours, and bit 27 is specially included as a third XOR input for its subsequent value. This is a "cellular automata" reminiscent of "Life", and introduces non-linearity. "Life" is played or run on a two-dimensional matrix of square cells of (ideally) infinite extent. Every tick of its "clock", each cell is declared as "live" if it previously had exactly three "living" neighbours out of its eight, unchanged if it had exactly two "living" neighbours, and "dead" for any other status. It produces many interesting non-linear combinations of cells, and it in particular, as well as cellular automata, are a separate field of study. The 37-bit CASR has a cycle length of (2^{37})-1 and a bias of 2^{-37}.

Each clock cycle, the low-order 32 bits are XORed together to form the output. The LFSR-CASR combination has a cycle length of $2^{80}-2^{43}-2^{37}+1$ and a bias of 2^{-80} [TKA2002]. The final XOR and the use of only the lower 32 bits of each state conceals the states from cryptanalysis.

The combination produces a good randomized output ([TKA2002] pg. 7), and does well on the "Diehard" tests ([TKA2002] pg. 8) – see section 2.7., "Random Number Generators", before.

3.2.3. Test Methodology and Use in this Work

In this work, this PRNG was first translated from Verilog to C++ – which was a significant coding change, and its output was verified for correctness against simulation of the original Verilog. Then the C++ program was made to output, in hexadecimal, up to 100,000 32-bit numbers, and up to 100,000 128-bit numbers by grouping the 32-bit numbers in fours. DOS Sort was used to sort the numbers into numerical order, and a second C++ program was written to count the numbers that fell within groups of values of the same first two and three digits, giving 256 and 4096 groups of numbers, respectively. The count of the quantity of numbers that fell into each group were plotted against the location of the groups in the number-lines of magnitude 2^{32} and 2^{128}.

The C++ code was then modified into C code, in a minor change, for inclusion in the IPsec implementation.

3.3. Design of an IPsec Implementation, "IPsecImp", Using the Five-Layer Security Framework

3.3.1. C vs. C++ for Embedded Systems

Since the IPsec implementation will have to go into an embedded system, amount of memory used is critical. Although the ML403 board has 1MB of SRAM, it is desirable for test and demonstration purposes to fit the entire software package into the available BRAM on the FPGA/CPU chip itself, for one-step downloading from a bit file. The implementation created along with the AES peripherals provided 32kB of BRAM. C++ code with a class and a constructor and small method of only approximately two lines, as generated by the GNU C++ compiler that came with the Xilinx EDK, required about 40k. In comparison, 64k times 16 would be the entire SRAM of 1MB. Clearly C++ is too costly in terms of memory usage for the ML403 board. Equivalent code compiled with the GNU C compiler required about one-seventh the memory, and the IPsec portion implemented used slightly less than 32kB (0x8000 bytes).

3.3.2. Top-Level Design of a Peer

An IPsec implementation, designed for demonstration and testing, was developed. The important functionality is located in the ML403 board, and called "IPsecImp" with a GUI to operate it, via an RS232 serial connection, located on a Windows PC, called "IPsecGUI". The GUI commands the OSI layer functionality to start, sends it its operating settings as chosen by the user, and displays results.

Figures 19, 20 and 21 give an ambitious top-level design for an IPsec peer, showing all five layers of the security framework. Following that, Figures 22 and 23 show the portion that was implemented.

Layering helps the design a good deal, by reducing it to an exercise in "connecting the dots" – but the design of the management layer still is not entirely rigorous. SLAs (Service Level Agreements) are used to determine the policy settings for the SMIB. The layers used are numbered as follows: 1., Policy, 2., Management, 3., Services, 4., Mechanisms, and 5., Primitives.

It was planned to use public-domain code for all standard mechanisms and primitives – which was done for the two primitives implemented, AES and the PRNG.

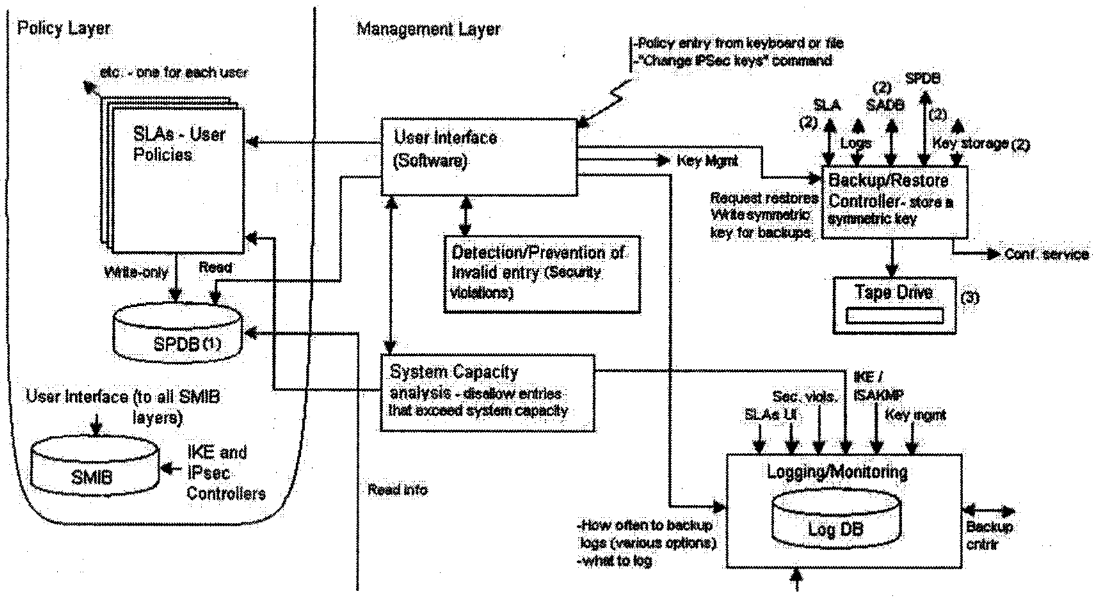

Policy Layer
Management Layer
etc. - one for each user
SLAs - User Policies
Write-only
Read
SPDB (1)
User Interface (to all SMIB layers)
SMIB
IKE and IPsec Controllers
-Policy entry from keyboard or file
-"Change IPSec keys" command
User Interface (Software)
Key Mgmt
Detection/Prevention of Invalid entry (Security violations)
System Capacity analysis - disallow entries that exceed system capacity
Read info
SLA
(2)
SADB
(2)
SPDB
(2)
Logs
Key storage (2)
Request restores
Write symmetric key for backups
Backup/Restore Controller- store a symmetric key
Conf. service
Tape Drive
(3)
SLAs UI
Sec. viols.
IKE / ISAKMP
Key mgmt
Logging/Monitoring
Log DB
-How often to backup logs (various options)
-what to log
Backup cntrlr

Figure 19. The Policy layer and upper portion of the Management layer – an ambitious design

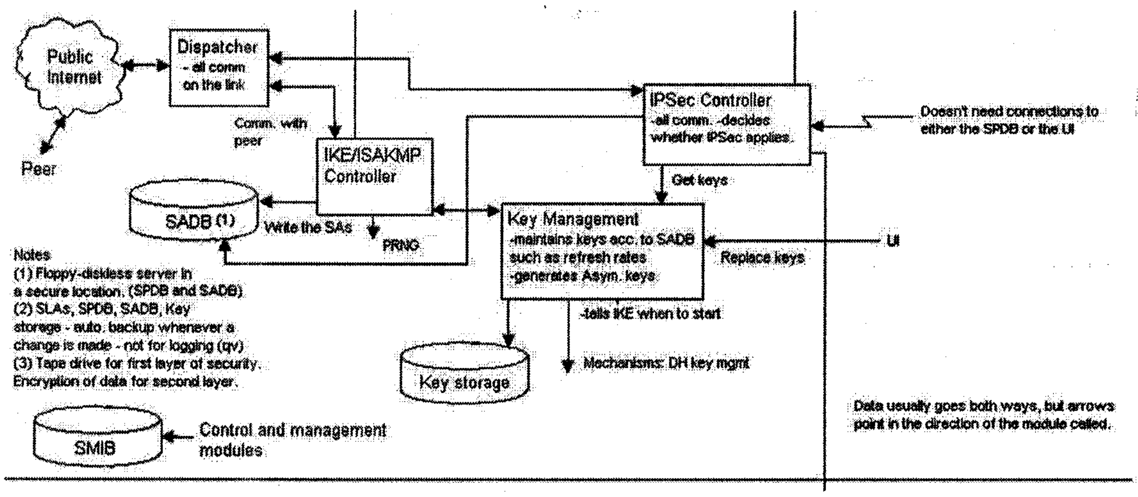

Public Internet
Peer
Dispatcher - all comm on the link
Comm. with peer
IKE/ISAKMP Controller
SADB (1)
Write the SAs
PRNG
IPSec Controller -all comm. -decides whether IPSec applies.
Doesn't need connections to either the SPDB or the UI
Get keys
Key Management -maintains keys acc. to SADB such as refresh rates -generates Asym. keys
Replace keys
UI
-tells IKE when to start
Mechanisms: DH key mgmt
Key storage
Notes
(1) Floppy-diskless server in a secure location. (SPDB and SADB)
(2) SLAs, SPDB, SADB, Key storage - auto. backup whenever a change is made - not for logging (qv)
(3) Tape drive for first layer of security. Encryption of data for second layer.
SMIB
Control and management modules
Data usually goes both ways, but arrows point in the direction of the module called.

Figure 20. The lower portion of the Management layer – an ambitious design

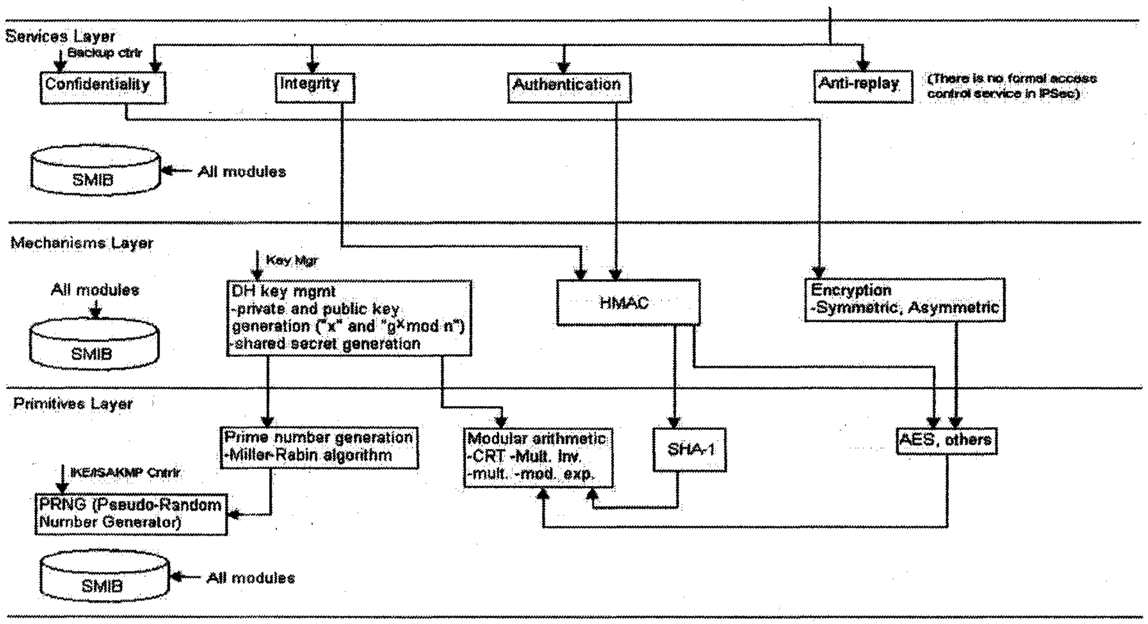


Figure 21. Service, Mechanism and Primitive Layers – an ambitious design

The portion implemented was a subset of the design shown in Figures 19, 20 and 21, as shown in Figures 22 and 23. See also Appendix A, for pseudo-code.

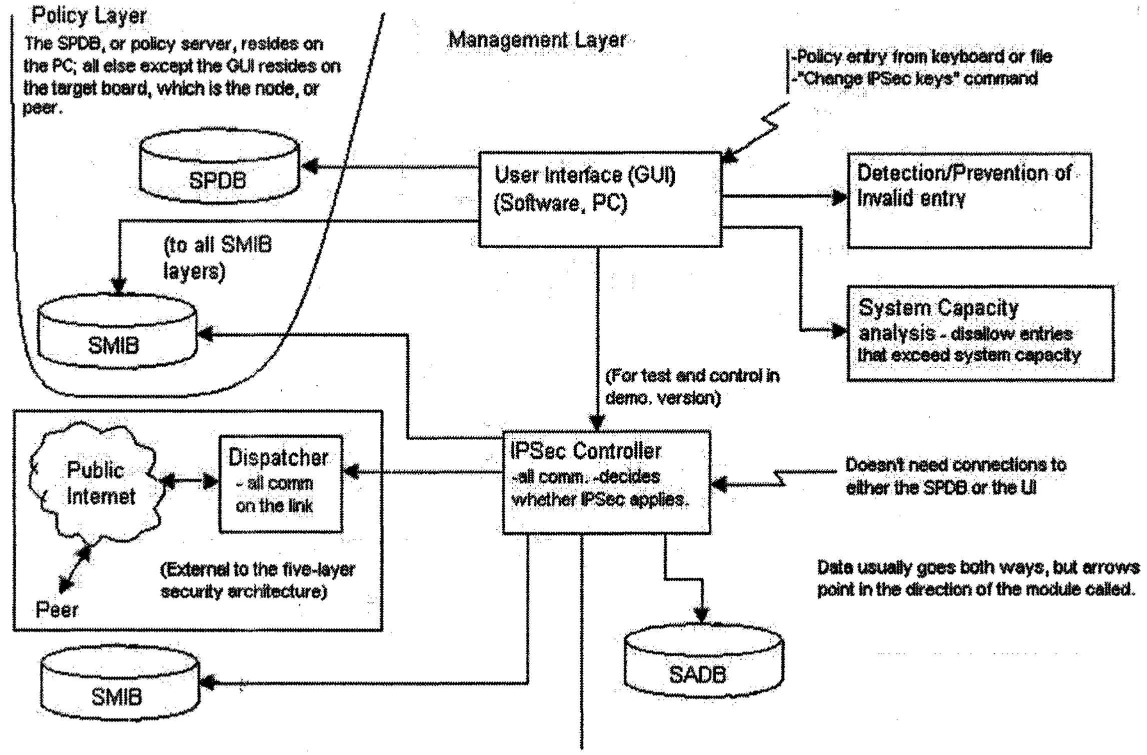

Policy Layer
The SPDB, or policy server, resides on the PC; all else except the GUI resides on the target board, which is the node, or peer.
Management Layer
-Policy entry from keyboard or file
-"Change IPSec keys" command
SPDB
User Interface (GUI) (Software, PC)
Detection/Prevention of Invalid entry
(to all SMIB layers)
SMIB
System Capacity analysis - disallow entries that exceed system capacity
(For test and control in demo. version)
Public Internet
Dispatcher - all comm on the link
IPSec Controller -all comm. -decides whether IPSec applies.
Doesn't need connections to either the SPDB or the UI
(External to the five-layer security architecture)
Peer
Data usually goes both ways, but arrows point in the direction of the module called.
SADB
SMIB

Figure 22. The Policy and Management Layers – portion implemented

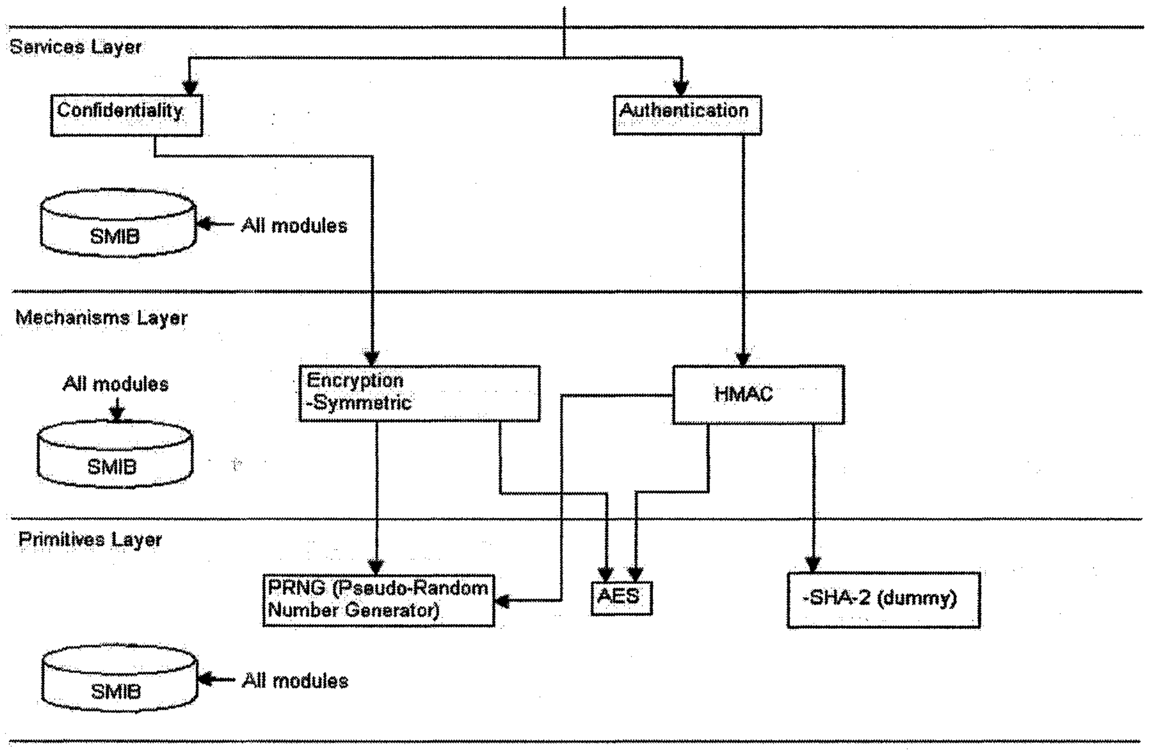


Figure 23. The Service, Mechanism and Primitive layers – portion implemented

The basic design is the same as used previously, in which an infinite main loop is used, within which a command character is awaited from the GUI via RS232, and acted on using a switch statement, with the difference that a "non-blocking" check for a received character is used in order to allow the outer loop to run freely. That allows other processing to be inserted before the command character check. In this design, two 32-bit counters were added to continually count, in order to be used as a random seed value for the PRNG. Since the PRNG used (see before) has internal states of more than 32 bits, only a portion of the higher-order 32-bit counter is used. That one is counted down (in a cycle) from 0xFFFFFFFF (hex.) and the lower-order one is counted up, from 0x00000000. Other processing is the functions of the OSI layers, including the IPsec sublayer, within an "if" block that is activated by a Boolean variable, *PacketProcessing*, that is turned on when a command character is received, sent by the controlling GUI when it is ready to send it a packet to process. The function calls for the three implemented OSI layers are listed within that "if" block, and they are repeatedly called, every iteration of the outer loop, until there is no more processing to do, in which case the

PacketProcessing variable is turned off. This top-level loop is considered the management layer; Figure 24 illustrates this design. See section 3.5., "Design of a CLI Version, 'IPsecLoop,' to Facilitate Testing," for a slightly differing version. Pseudo-code for most functions implemented can be found in Appendix A.

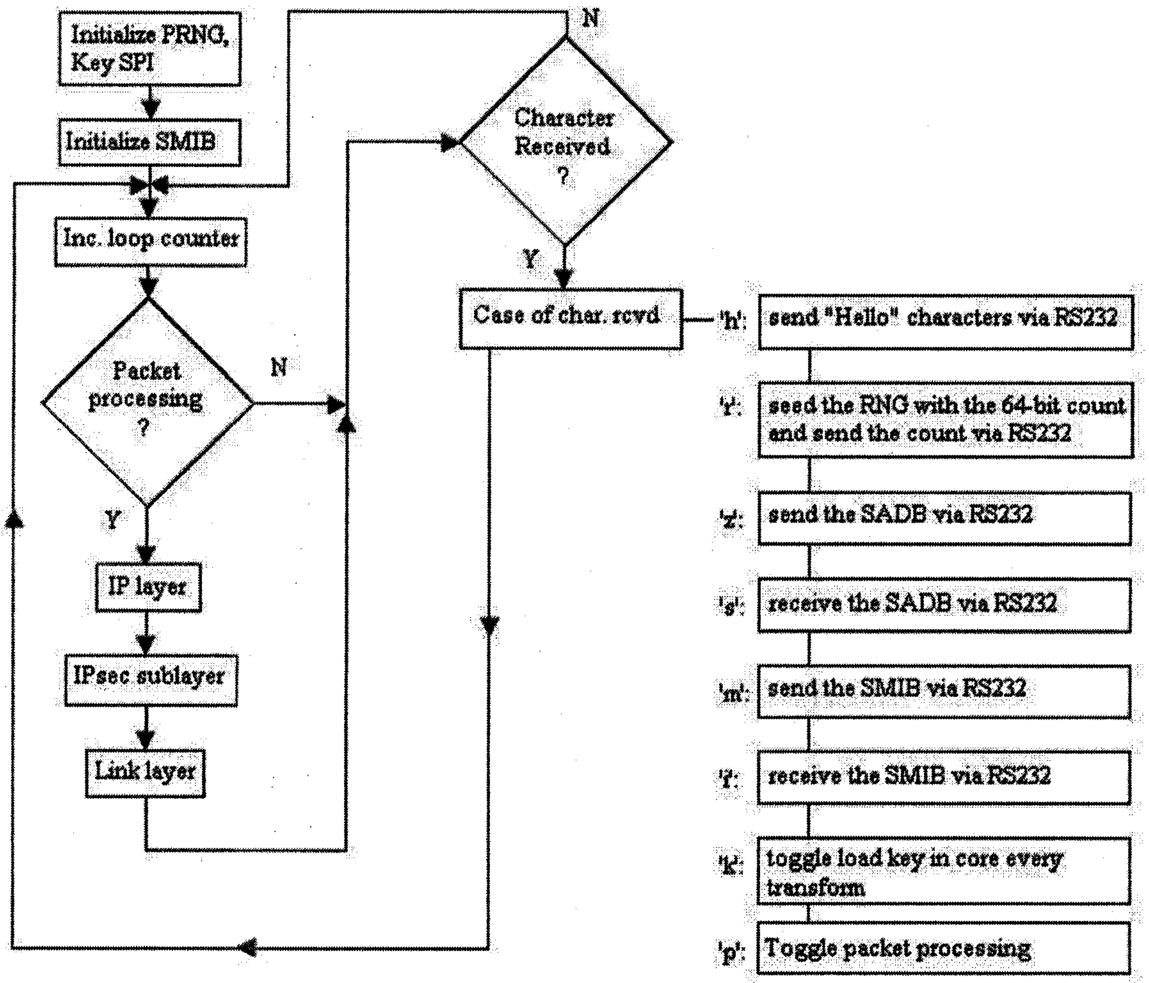

Initialize PRNG, Key SPI
Initialize SMIB
Inc. loop counter
Packet processing ?
N
Y
IP layer
IPsec sublayer
Link layer
Character Received ?
N
Y
Case of char. rcvd
'h': send "Hello" characters via RS232
'r': seed the RNG with the 64-bit count and send the count via RS232
'z': send the SADB via RS232
's': receive the SADB via RS232
'm': send the SMIB via RS232
'r': receive the SMIB via RS232
'k': toggle load key in core every transform
'p': Toggle packet processing

Figure 24. "IPsecImp" top-level loop flowchart

The command characters are the following: 'r': the PRNG is seeded from the free-running counter values. The unpredictable time at which a human operator would cause the GUI to send this command introduces the true random element. The seed value is also sent to the GUI. 'h': the characters of the string "Hello" are sent to the GUI, as an indication of board "liveness". 'z': the SADB is sent from the board to the GUI. 's': the SADB is received from the GUI by the board. 'm': the SMIB is sent from the board to the GUI. 'i': the SMIB is received from the GUI by the board. 'k': a variable, *KeyIsAlwaysTreatedAsNew* is toggled that determines whether the key should always be loaded to the core for every IPsec packet transform or if it should only be loaded when it changes. Internally, there is a *NeedNewIV* Boolean variable that is set when the IV from the SADB (and otherwise when required, see after) is zero. 'p': *PacketProcessing* is toggled. It is only turned on if there is a protocol and mode set in the SADB. This code can also be operated via Hyperterminal for debugging purposes. For the purpose of discarding data sent when the board is not ready for it, none of these command characters are characters that represent hexadecimal digits.

Before this outer loop is entered, the SMIB is initialized to the services, mechanisms and primitives made available by the embedded software and FPGA configuration. A cleared SADB is also created.

The C language syntax provides the ability to define variables for the compiler's pre-processor to read; these are called "#define" ("pound-define"), and can be checked using "#ifdef". In this design, the C function prototypes and their external declarations were put in the same file, and a "#ifdef" was used to check the alternative to be used, using conditional compilation. A header file that included the main header file was used to be included in files that used another file's functionality, that "#define"ed the variable to be checked. The native header file was included in its own source code file, so that the variable would not be defined and the native file would have its function prototypes selected. This technique can be used to implement a limited form of the object-oriented concept of private methods and data; generally a header file's #defines are made available to other files that need to use that file's functionality in this technique. Another use of "#define" and "#ifdef" is to "#define" a variable at the top of a header file if it is not already defined, and to include the contents of the header file as well, in that conditional compilation. This prevents errors if header files included in the header file include the same header file at some level of inclusion; the header file will not include its content if its variable is already defined.

3.3.3. SMIB (Security Management Information Data Base)

The SMIB is organized using C language "structs" (data structures), one representing each layer. The SMIB itself is a "struct", containing the lower-level "structs". Some ideas for general content of an SMIB were found in [KEN1994] (see also section 1.2.2., "Security Policy Database and SMIB", before). The policy "struct" contains an integer giving the number of policies, and a pointer to the defined policy "struct", allowing more than one policy to be pointed to, by allocating the memory required. Although the SMIB contains a value for the number of policies, only one policy at a time is used in the board at this time. The management layer "struct" contains four bytes for the local IP address, an integer for the number of clients, a Boolean variable indicating whether the client addresses should be interpreted as pairs, for range purposes, and a pointer to an array of four-byte client addresses, so that any number of client addresses can be referenced. The Services layer "struct" contains a "struct" for each of AH and ESP, each of which contains a Boolean variable to indicate whether the service is available, and an integer to indicate the mechanism number. In future work, the AH and ESP structs can be replaced with pointers to them so that more than one can be stored. This would allow selection of different services. In the same way, the mechanism number would allow selection of different mechanisms. The mechanism layer is designed in a similar way, as is the primitives layer, which could allow a great deal of flexibility in choosing combinations of algorithms. The numbers in the primitive layer "structs" refer to primitive algorithm numbers which can be chosen for each type of primitive.

3.3.3.1. The Policy Layer

The policy "struct" has two conceptual groups of variables: selectors and SA negotiation goals. The selectors determine whether a packet is to be processed, in flexible ways; each has a Boolean variable associated with it to determine if it should be used, or interpreted to match any packet. The six selectors implemented are understood to be combined together in an "AND" sense, since each selector represents an additional criterion to check. An overall Boolean is used to set whether the result of the overall selection should be taken in the opposite sense.

In the "negotiation goals" group of variables, the protocols and modes desired should be considered to be non-negotiable. A "negotiate" Boolean is provided to select whether it is acceptable to negotiate different service numbers of those available. Two sets of negotiation goals are provided, inner and outer, so that IPsec protocols can be nested in pairs.

3.3.4. SADB (Security Association Data Base)

See the SADB plan in the pseudo-code in Appendix A for the complete list of contents of the SADB.

Note that the Service number field in the SADB is the location where the number chosen in negotiation from one of the available services is stored.

The "IV Constant" field in the SADB is used to clear *NeedNewIV* (equivalent to setting it to "FALSE", in C – any non-zero value in a C variable is interpreted as "TRUE"), unless the IV is zero, in which case *NeedNewIV* is set on a one-time basis. This helps to counteract the natural user disinclination to originate cryptographic material. *NeedNewIV* is passed down the layers via function calls to the mechanisms layer, where it induces the getting of a new IV using the RNG in the sending case, when encryption is used.

As it was realized that an SA is associated only with a communication in one direction, considerable simplification resulted from reducing the number of keys and IVs from eight to two, since incoming and outgoing cryptographic material did not need to be stored in the SA, and neither did that material need to be stored on behalf of the other entity.

3.3.5. OSI Layers Implemented

The IP (Internet Protocol) layer, IPsec layer and Link layer are represented in this work. The Link layer is a dummy layer that calls the dispatcher to get the IPsec packet and send it back to the IP layer. The IP layer is largely a dummy layer that receives the packet from the GUI via reliable data transfer, in "IPsecImp", or copies the established test packet in memory, in "IPsecLoop" (see after). When a layer receives a pointer to allocated memory via the dispatcher, and uses it, it is responsible for deallocating that memory. When the IP layer receives the IP packet back via the dispatcher from the IPsec layer, it echoes it and then deallocates its memory.

The IPsec layer is based on the service calls. It checks the dispatcher for the presence of a packet. There are two outer if blocks, one for the transmit, sending, outgoing, or IP to IPsec case (in which case the IP packet is converted to an IPsec packet) and one for the receive, incoming, or Link to IPsec case (in which case the IPsec packet is converted to an IP packet). In the transmit block, the SADB for outgoing packets is checked for the protocol to use and the appropriate service call is made. In the receive block, the protocol field of the datagram is checked and then the SPI is checked for a match against that set in the SADB for incoming packets, before the service call is made. The pointer to the "incoming" SADB is set, in the management layer, to the "outgoing" SADB so that they will be identical for the purposes of this research. Any error codes generated from lower levels are passed to the Management layer for appropriate action, such as stopping packet processing and displaying error messages. Before each service call, the reliable data transfer routine is used to send a short synchronization message, literally the characters "Synch", to the GUI, to alert the GUI to start its timer to measure the duration of the service call. Following the service call, regular RS232 output is used to alert the GUI that processing has ended. Following the "receiving" AH service call, a message indicating whether the authentication succeeded is sent via RS232; reliable data transfer was used to prevent the GUI from missing it. Figure 25 illustrates the design of the IPsec layer.

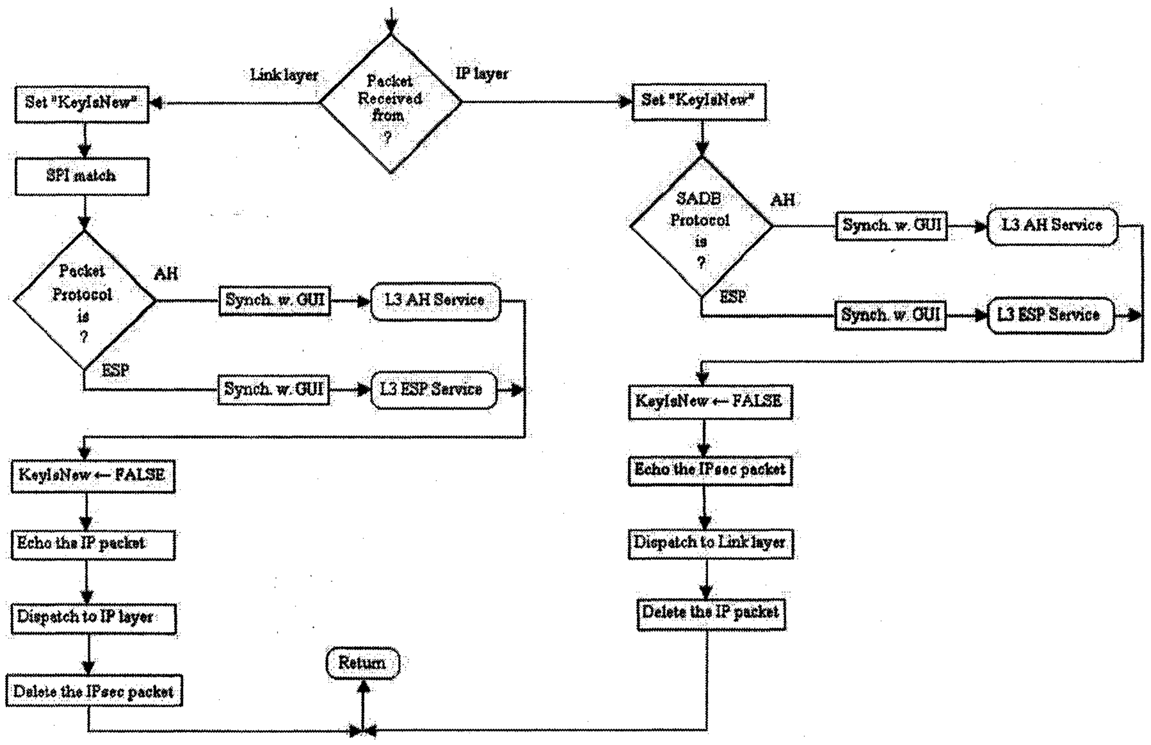


Figure 25. The IPsec layer flowchart

3.3.6. Board-GUI Reliable Data Transfer

In development of the AES file encryption demonstration, "AESfile", a considerable number of extraneous characters were observed being received by the GUI. Their source is unknown, whether that was a characteristic of the serial package used [KLE2003]. In an attempt to deal with that in the "IPsecGUI", first it was attempted to require all characters, even printing characters themselves, to be hex-ASCII encoded and allow reception of only characters representing hex. digits, discarding any others. However, extraneous characters could still conceivably be hex. digits (0-9, a-f, or, A-F), and debugging using Hyperterminal was made tedious since even the text strings sent were encoded and readable only with much difficulty. What was really needed was reliable data transfer.

The requirements for this feature were the sending of a checksum, its verification, as well as three-way handshaking. Not only should the sending code receive an "ACK" or "NAK", but the receiving code should have it echoed back to know that its response has been received. Finally, the sending code should receive an acknowledgement that the receiver knows that the sender has received its response. Another requirement was that communication should not "deadlock", for example, with the receiver waiting for a character and the transmitter not sending any more.

The above requirements were satisfied by the following. First, the "receive" code sends "STX" when it begins to wait for a transmission. The sender begins sending when it receives an "STX". When the receiver receives something, it stops sending "STX". The "send" code sends its message, followed by a hex-ASCII-encoded checksum, and then sends "ETX" characters if it does not receive a response, until it receives a response. The message is hex-ASCII-encoded if it consists of numeric data, and the checksum is calculated from the unencoded data. The receiver uses the numeric ASCII value for "ETX", 0x03, if dropouts occur and it needs some "ETX"es to make up the required length of the message, which is supplied to the "receive" or "send" code by its respective calling function. In that case, the message verification fails (which was tested as being quite reliable). When the receiver has done the message verification, it then sends "ACK" or "NAK", until it receives back either one, and then stops sending. The "send" code is aware that the "receive" code has received back its echo when the receiver stops sending. If "NAK" was sent by the receiver, both routines begin again. In practice, these routines had to be carefully adjusted with delay loops so that the board, with its 100 MHz clock, could successfully communicate with the PC used, with its 2.39 GHz clock, and with the laptop, with its 701 MHz clock (Pentium III CPU). Figures 26 and 27 illustrate the design of the send and receive code.

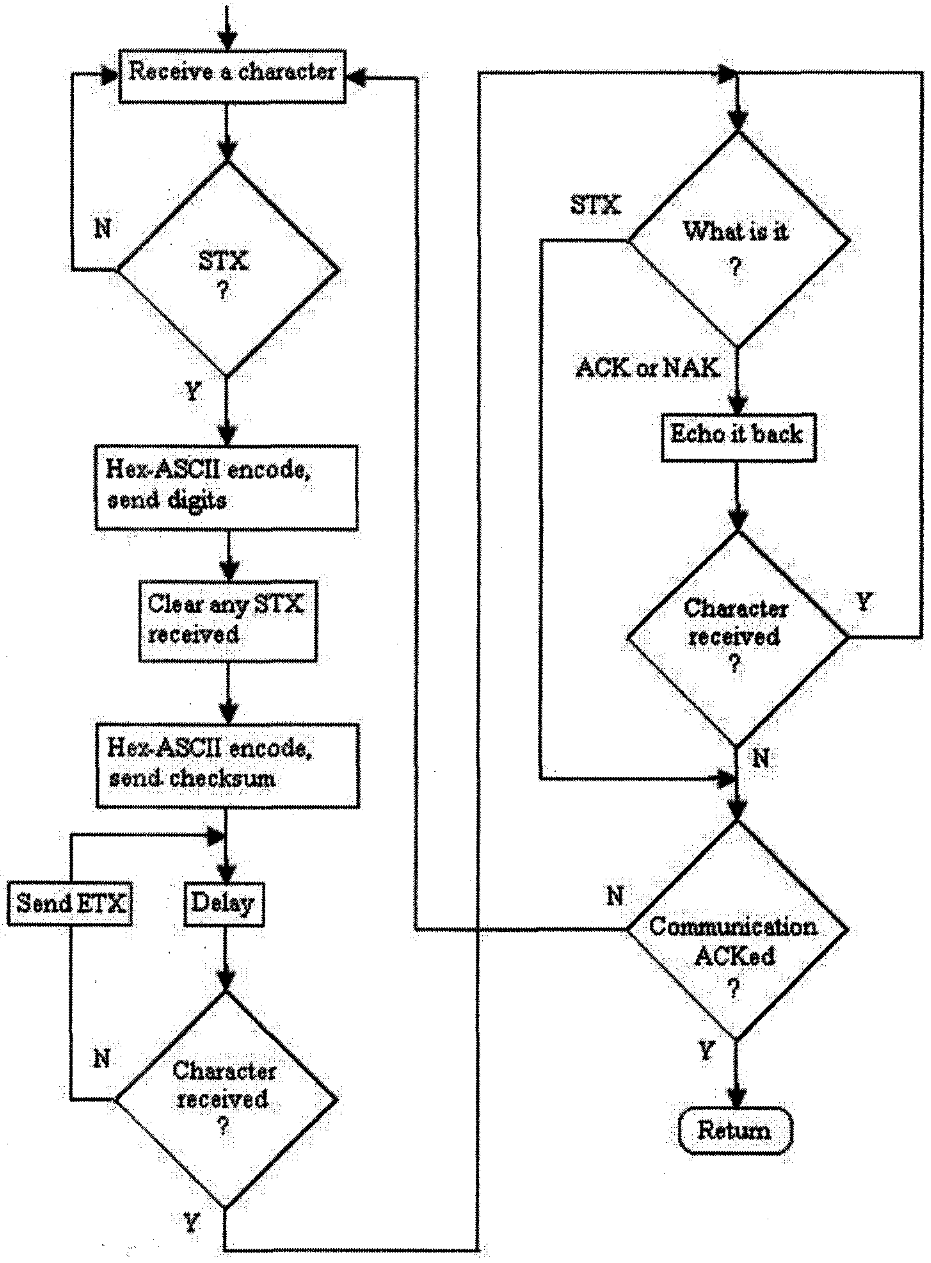


Figure 26. The reliable data transfer "send" flowchart

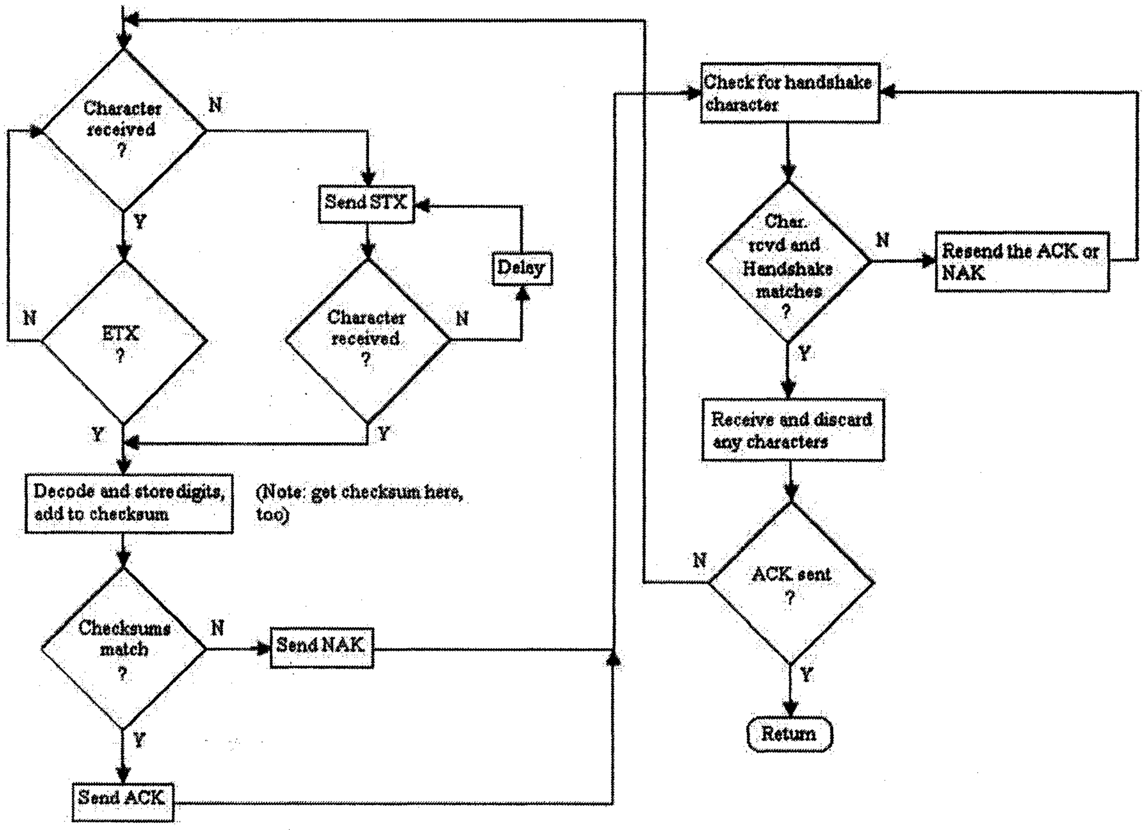

Character received ?
N
Y
ETX ?
N
Y
Send STX
Delay
Character received ?
N
Y
Decode and store digits, add to checksum
(Note: get checksum here, too)
Checksums match ?
N
Y
Send NAK
Send ACK
Check for handshake character
Char. rcvd and Handshake matches ?
N
Y
Resend the ACK or NAK
Receive and discard any characters
ACK sent ?
N
Y
Return

Figure 27. The reliable data transfer "receive" flowchart

These *routines* were only used to transfer numeric data; the "send" function included an input indicating whether the string pointed to should be sent via reliable data transfer, so that the same function could be reused to send unencoded characters without expecting any response, in order to avoid code duplication. The ML403 board code and the GUI code were written so that it was known at a given program location whether reliable data transfer was to be used. Each entity, "IPsecImp" in the board and "IPsecGUI" in the PC, has both a send and a receive function. If data of varying length was to be sent, the length was first sent as numeric data using a variable of known length in order to send a known number of bytes.

3.3.7. The Services Layer

Both an AH and ESP service were written. A pointer to the SMIB and to the SADB are passed to these routines, which use the service number in the SADB to trace the primitives to use, via the SMIB, from the service number, via the mechanism number, to the primitive algorithm number. This is necessary because it is these routines that allocate the memory for the transformed packet and therefore must access information from the Primitives layer in order to determine precise data sizes needed. This is an example of strict layering making implementation difficult (see section 1.5., "Motivation for General Layering"). Since the goal is a successful implementation, the end should not be sacrificed to the means; doing that might constitute fanaticism, which sometimes occurs in the form of increasing effort while losing sight of objectives.

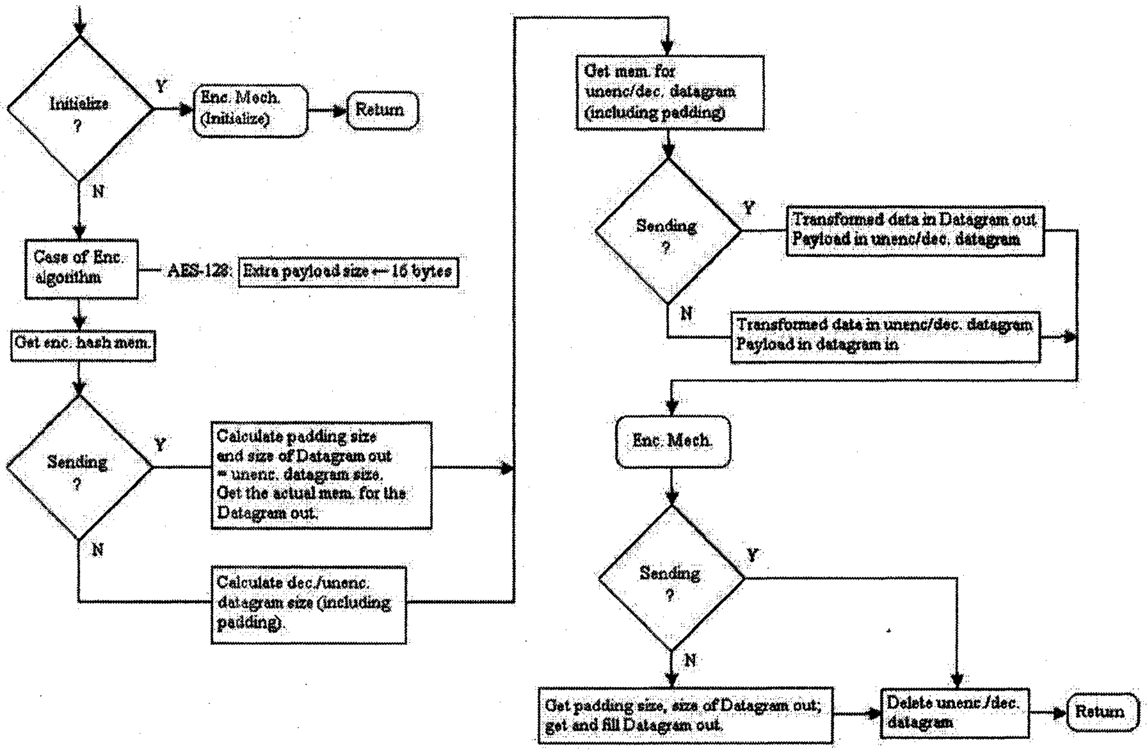
These routines also read and build IP and IPsec packets. IPv4 is supported in this work ([BAC1997], "IP Packet Structure," http://www.freesoft.org/CIE/Course/Section3/7.htm), [RFC0791]. Note that in the RFCs and in other standards documents, the term of choice to replace "byte," is "octet," since the term "byte" is sometimes used loosely, to refer to other than eight bits; in RS232 transmission, the presence of start, stop and parity bits often means that a "byte" is nine, ten or eleven bits. Note also, that in this design, the AES IV is included, unencrypted, in every packet that contains data encrypted using AES, at the beginning of the section, immediately before it.

3.3.7.1. The ESP Service

In the ESP service, for sending, the size of the space required for the IPsec datagram (a specific term for a packet at the IP level [KUR2000]) is calculated and the space for the datagram is allocated from the system heap, using the standard C function, malloc() [KER1988]. In addition, space for at least the data to be encrypted is allocated, since padding has to be added to conform to a natural number of encryption block sizes, and to hold the padding size information itself. This latter space has at least the packet payload copied to it and is configured with the padding and the padding size, in order to send this space to the mechanism for encryption. Any unused padding area is set to zeroes, for confidentiality, in order to prevent any leak of data left in the memory area used, which should not be sent along with the packet. The Encryption mechanism routine is called and is passed a pointer to the destination address for the ciphertext, the source address of the plaintext, and the data length. In addition, the mechanism is sent the mechanism number to use, determined from the SMIB, and the key and the IV from the SADB. It is also sent the SPI from the SADB for the purposes of only updating the key in the encryption core when necessary.

Upon reception, the IP datagram can only be allocated after the mechanism is called, because the padding size is not known until then. Instead, space is allocated for the size of the IPsec datagram payload, less only the size of the IV, as a destination for the decrypted data from the mechanism. The IP datagram payload only is then transferred to the IP datagram built.

Before exiting, the temporary payload area is freed, leaving only the datagrams-in and -out, of the service call. Figure 28 illustrates the design of the ESP service.



Initialize ?
Y
Enc. Mech. (Initialize)
Return
N
Case of Enc. algorithm
AES-128:
Extra payload size ← 16 bytes
Get enc. hash mem.
Sending ?
Y
Calculate padding size and size of Datagram out = unenc. datagram size. Get the actual mem. for the Datagram out.
N
Calculate dec./unenc. datagram size (including padding).
Get mem. for unenc/dec. datagram (including padding)
Sending ?
Y
Transformed data in Datagram out Payload in unenc/dec. datagram
N
Transformed data in unenc/dec. datagram Payload in datagram in
Enc. Mech.
Sending ?
Y
N
Get padding size, size of Datagram out; get and fill Datagram out.
Delete unenc./dec. datagram
Return

Figure 28. The ESP service flowchart

3.3.7.2. The AH Service

In the AH service, the datagram must always be hashed, in order to create the authentication data when sending, or for comparison with the decrypted authentication data when receiving. Therefore space for the generated hash must always be allocated. If receiving, space for the hash, as decrypted from the authentication data, must also be allocated. In addition, a copy of the IPsec datagram should always be created, in order to build a copy to send to the HMAC mechanism for hashing and signing (encrypting, if sending). This copy requires mutable fields (those fields that are changed when the packet is transmitted across links) to be zeroed. This copy can become the IPsec datagram when sending; doing this increases processing speed. When receiving, this datagram is created in this design, but it would be possible to increase processing speed by modifying the IPsec datagram that was passed in to the service, since it is always deleted by the IPsec sublayer following the service call. The IP datagram is directly created from the IPsec datagram if receiving since doing so amounts to only removing the AH header, and, if in Tunnel mode, the extra IP header.

An IPsec datagram is thus always passed to the HMAC mechanism for hashing, if sending or receiving. The pointer to the encrypted authentication data is set to its location in the original IPsec datagram if receiving, and set to separate allocated storage if sending, because the datagram to be hashed must not be written to while it is being read; this is passed to the mechanism. Pointers to the generated hash storage and the decrypted hash storage are also sent to the mechanism. As in the ESP mechanism call, the mechanism is sent the mechanism number to use determined from the SMIB, and the key and the IV from the SADB. It is also sent the SPI from the SADB for the purposes of only updating the key in the encryption core when necessary.

After the call to the mechanism, if sending, the authentication data is copied to the AH area in the IPsec datagram. If receiving, the generated hash is compared to the decrypted hash and a Boolean variable indicating whether the verification succeeded is passed out of the AH service call. All storage is freed except for the datagrams in and out, of the service call. Figure 29 illustrates the design of the AH service.

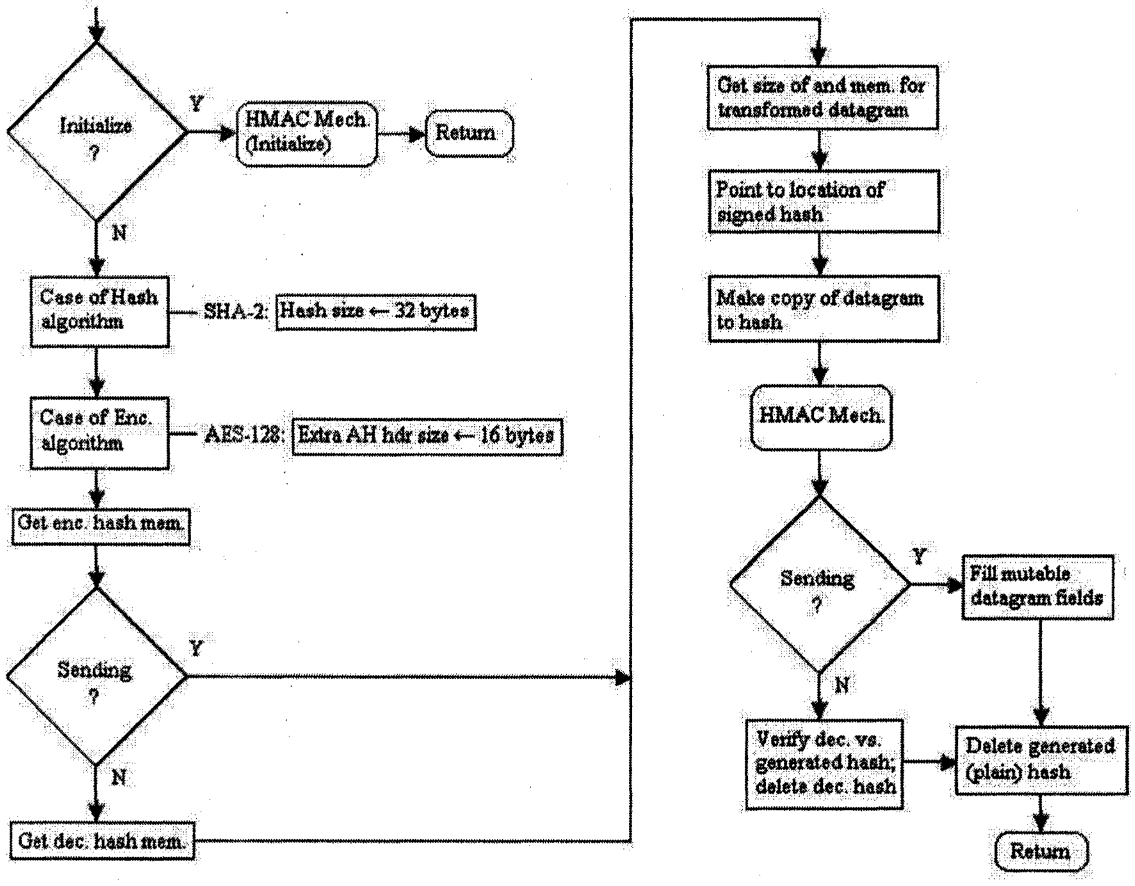

Initialize ?
Y
HMAC Mech. (Initialize)
Return
N
Case of Hash algorithm
SHA-2: Hash size ← 32 bytes
Case of Enc. algorithm
AES-128: Extra AH hdr size ← 16 bytes
Get enc. hash mem.
Sending ?
Y
N
Get dec. hash mem.
Get size of and mem. for transformed datagram
Point to location of signed hash
Make copy of datagram to hash
HMAC Mech.
Sending ?
Y
Fill mutable datagram fields
N
Verify dec. vs. generated hash; delete dec. hash
Delete generated (plain) hash
Return

Figure 29. The AH service flowchart

3.3.8. The Mechanisms Layer

3.3.8.1. The Encryption Mechanism

The encryption mechanism contains the code to initialize the SPI-based key tracker, and the RNG. If the mechanism is passed the command to initialize these, it does so, seeding the PRNG with all "1"s. It exits immediately if it does initialization.

The mechanism mainly consists of an outer switch statement that is used to select the desired processing for the desired set of primitives found for the particular mechanism number that was passed in. AES-128 is the only primitive so far implemented for encryption. In that case the SPI-based key tracker is used to help determine whether the key needs to be updated in the core. In addition, if the *KeyIsNew* variable passed in from the upper layers is true (where it is determined from *KeyIsAlwaysTreatedAsNew*, and other factors, if that is false), the core is updated with the key. When the key is updated in the core, the SPI-based key tracker is also updated with the SPI used. If sending, if the *NewIVNeeded* variable passed in from the upper layers is true, the selected RNG is used to obtain a new IV. Since the IV storage area was passed to the mechanism via a pointer (i.e., as a memory address), this updates the IV in the SADB. The IV is also copied to the beginning of the destination transform area whose pointer was passed in to the mechanism. If receiving, the IV is retrieved from the beginning of the transform area. The IV is written to the core via the primitives, and the primitive is called to encrypt or decrypt the data starting from the address of the origin transform data and to place the transformed data in the destination transform area. The starting addresses have to be adjusted to follow the IV in the destination transform area if encrypting, and in the origin transform area if decrypting. An error code is accepted from the primitive and passed out of this mechanism when it completes. Figure 30 illustrates the design of the encryption mechanism.

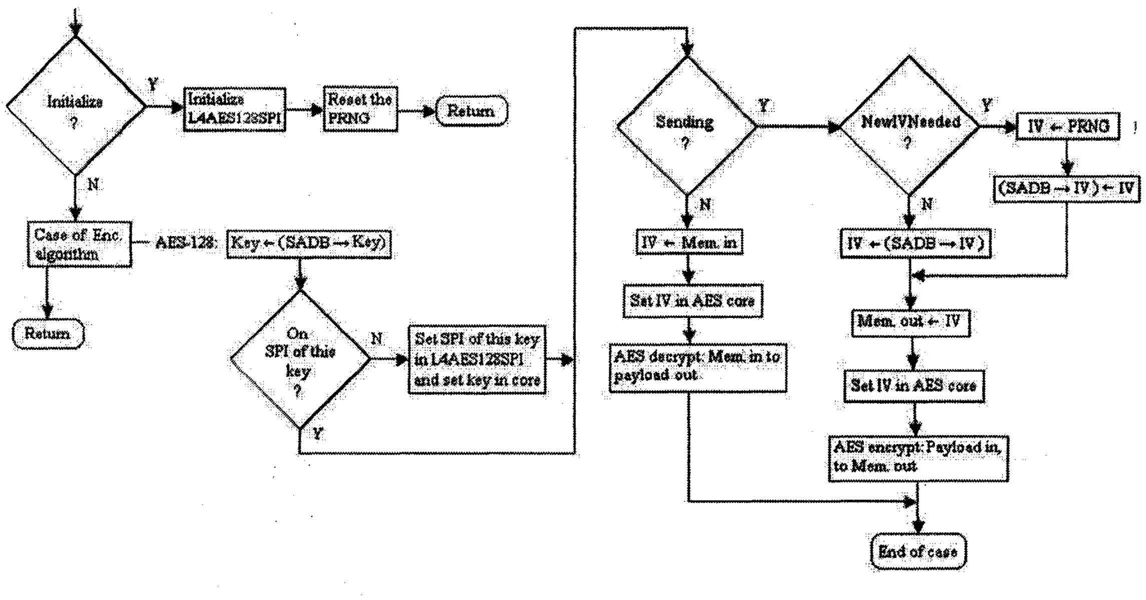


Figure 30. The encryption mechanism flowchart

3.3.8.1. The HMAC Mechanism

The HMAC mechanism contains an empty "if" block for initialization in case any HMAC-specific initialization needs to be done.

The HMAC mechanism contains a switch statement to select and use the Hash function specified in the SMIB. The generated hash is passed out of the mechanism using the pointer that was passed in. A second switch statement selects the encryption method, whose functionality is the same as that explained for the encryption mechanism. In future work, considerable object code could be saved by making that encryption block into a function. The decrypted or encrypted hash is passed out using the passed-in pointer. Figure 31 illustrates the design of the HMAC mechanism; note the similarity in the case of the encryption algorithm to that in the Encryption mechanism.

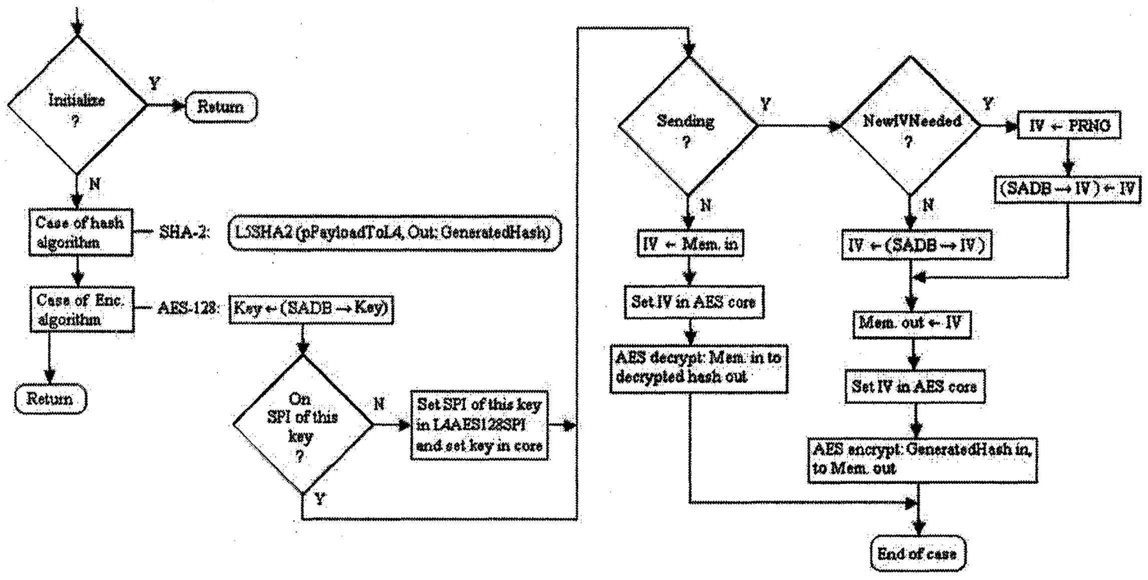


Figure 31. The HMAC mechanism flowchart

3.3.9. The Primitives Layer

3.3.9.1. The Hash Primitive

The hash primitive is a dummy hash, intended to be filled with the SHA-2 hash routine in future work. Thus 256 bits, or eight 32-bit words, are passed back from this function, each containing, alternately, 0x5A5A5A5A and 0xA5A5A5A5. See the figure in section 3.3.9.3., “The RNG Primitive,” for a diagram.

3.3.9.2. The Encryption Primitive

Three functions are provided, to update the IV in both the AES encryption and decryption cores in one, to update the key in the cores, only doing the key expand process and copy to the decryptor core when specifically commanded, and to encrypt or decrypt a block of memory from an origin starting address to a destination starting address, for a length of memory given in bytes. The length is passed in twice to this latter function, for error checking and the length is also checked to confirm that is a natural-number multiple of the 128-bit block size of sixteen bytes. An error code is returned if this check fails. Figure 32 and 33 illustrate the design of this primitive.

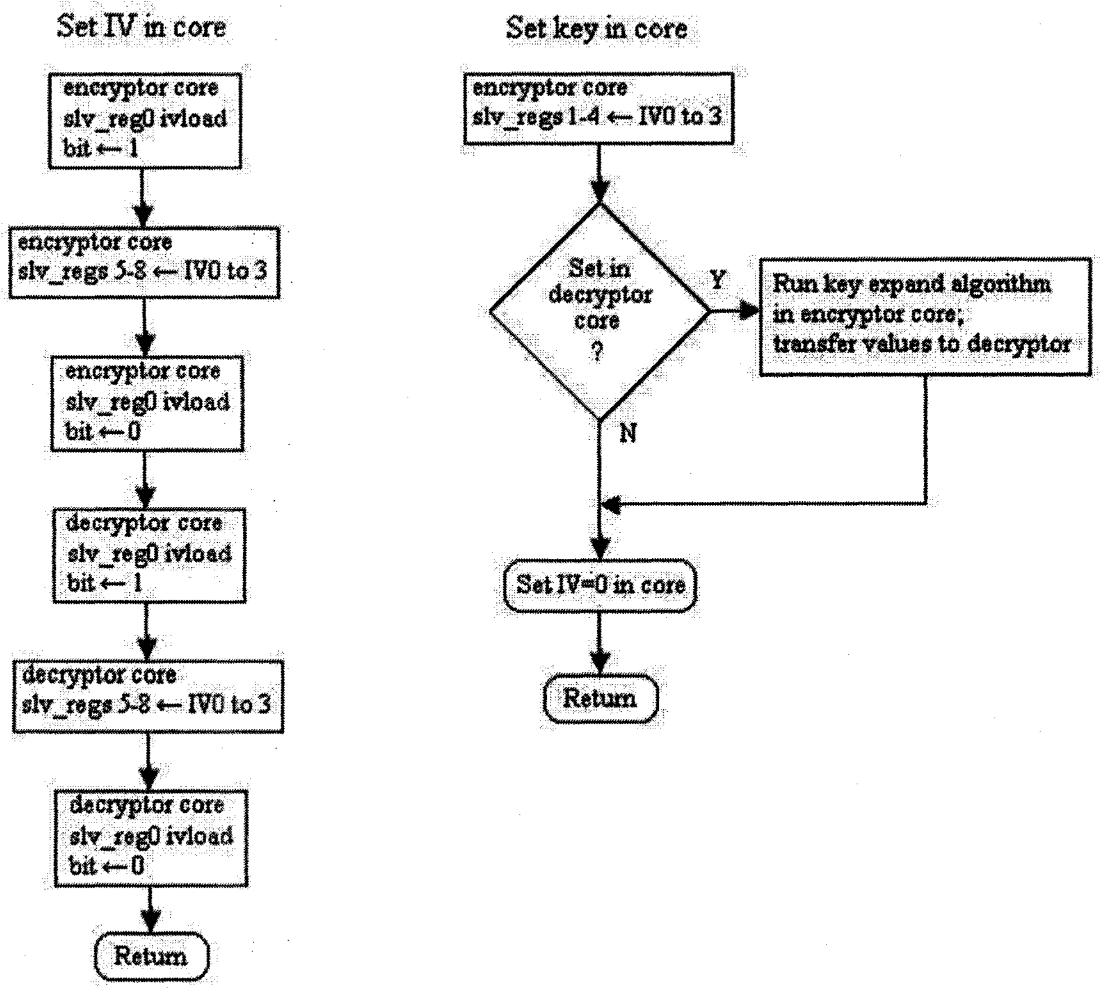

Set IV in core
encryptor core slv_reg0 ivload bit ← 1
encryptor core slv_regs 5-8 ← IV0 to 3
encryptor core slv_reg0 ivload bit ← 0
decryptor core slv_reg0 ivload bit ← 1
decryptor core slv_regs 5-8 ← IV0 to 3
decryptor core slv_reg0 ivload bit ← 0
Return
Set key in core
encryptor core slv_regs 1-4 ← IV0 to 3
Set in decryptor core ?
Y
N
Run key expand algorithm in encryptor core; transfer values to decryptor
Set IV=0 in core
Return

Figure 32. Encryption primitive IV and key load flowcharts

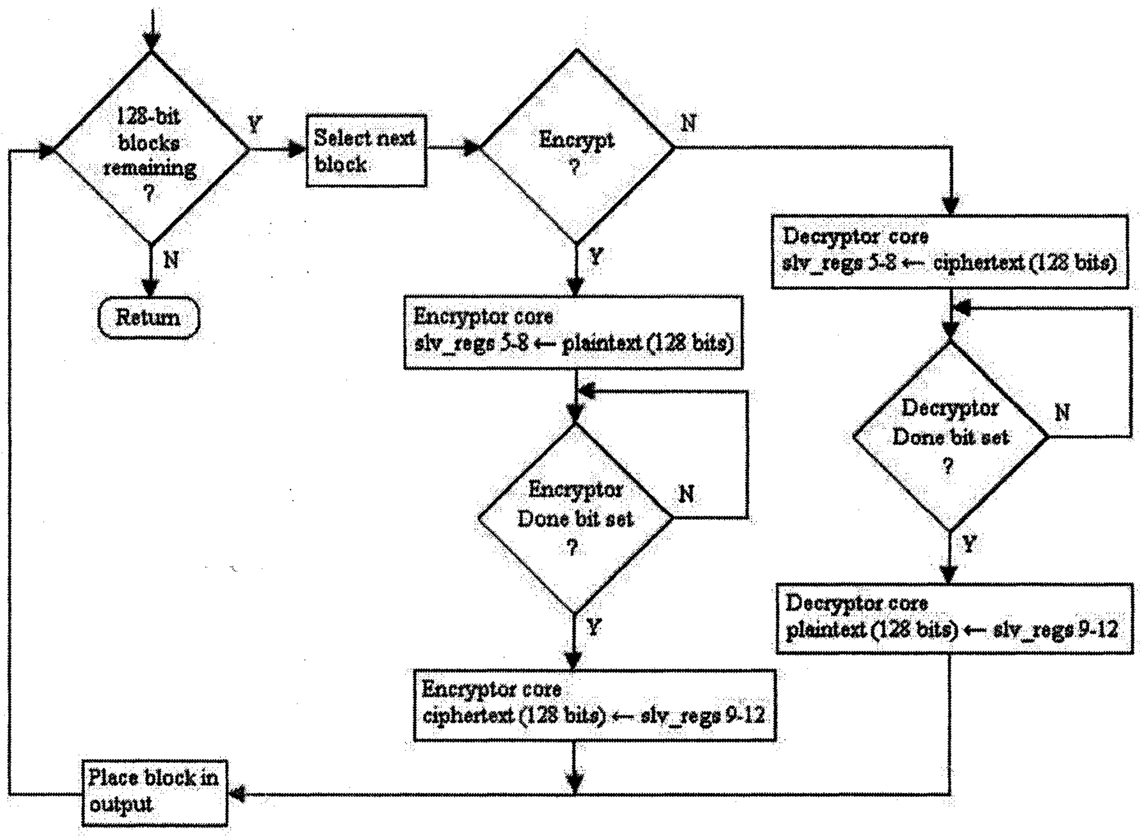

128-bit blocks remaining ?
Y
N
Return
Select next block
Encrypt ?
N
Y
Encryptor core slv_regs 5-8 ← plaintext (128 bits)
Encryptor Done bit set ?
N
Y
Encryptor core ciphertext (128 bits) ← slv_regs 9-12
Decryptor core slv_regs 5-8 ← ciphertext (128 bits)
Decryptor Done bit set ?
N
Y
Decryptor core plaintext (128 bits) ← slv_regs 9-12
Place block in output

Figure 33. The encryption primitive flowchart

3.3.9.3. The RNG Primitive

Four functions are provided. One function acts as the storage for the state of the LFSR-CASR PRNG. Static variables are used in this function to contain the two 64-bit state variables in the form of four 32-bit words. As explained before, not all of the high-order 32-bit words for each of the LFSR and CASR are used. The function will store or retrieve these values depending upon the command passed in. Another function resets the four 32-bit words passed in. Another seeds the four 32-bit words that are passed in. The most important function generates the pseudo-random number generator from the current state (the four 32-bit words) passed in, and returns a 32-bit pseudo-random number. Generation updates the state variables, as described before (see section 3.2., "Design of a Combination LFSR-CASR Pseudo-Random Number Generator"). Whenever the state variables are required for passing into the generation function, they must be retrieved from the storage function first (except of course if they are updated repeatedly before the local state values are lost such as by exiting the current function) and whenever they are updated by one of the reset, seed, or generation functions, they must be stored using the storage function. Figure 34 illustrates the design of the functions written to make use of the PRNG primitive (as well as an illustration of the dummy hash routine).

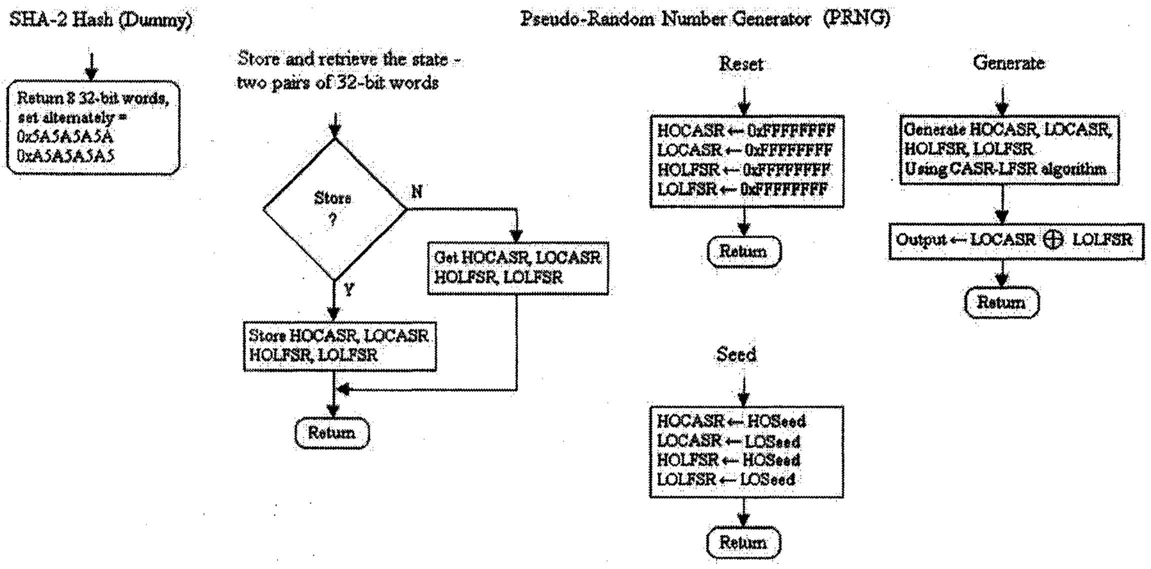

SHA-2 Hash (Dummy)
Return 8 32-bit words, set alternately = 0x5A5A5A5A 0xA5A5A5A5
Pseudo-Random Number Generator (PRNG)
Store and retrieve the state - two pairs of 32-bit words
Store ?
N
Y
Get HOCASR, LOCASR HOLFSR, LOLFSR
Store HOCASR, LOCASR HOLFSR, LOLFSR
Return
Reset
HOCASR ← 0xFFFFFFFF
LOCASR ← 0xFFFFFFFF
HOLFSR ← 0xFFFFFFFF
LOLFSR ← 0xFFFFFFFF
Return
Generate
Generate HOCASR, LOCASR, HOLFSR, LOLFSR Using CASR-LFSR algorithm
Output ← LOCASR ⊕ LOLFSR
Return
Seed
HOCASR ← HOSeed
LOCASR ← LOSeed
HOLFSR ← HOSeed
LOLFSR ← LOSeed
Return

Figure 34. The hash and pseudo-random number generator flowcharts

3.3.10. Versions

Four different versions of the actual layers to be tested for performance, Service, Mechanism and Primitive, were created, mostly involving changes to the Service functions, due to their complexity. Some changes were made to the mechanisms, but none to the primitives. A design change was made in *ver. 7* to conform better to the layering idea, and to use the SMIB. Some additions were made to the SMIB. The version numbers are from the IPsecLoop CLI version of the project, but the same actual files and therefore the identical code is used for the Services down to the Primitives layers in both versions. This replaceability (or fungibility, to use the correct, but obscure word) due to modularity is a huge benefit derived from layering.

ver. 4: This was the initial version tested.

ver. 6: The ESP service was modified to remove "for" loops that cleared the entire datagram, which would cause delays proportional to the packet size.

ver. 7: Clearing of the padding area in ESP was restored that was removed in *ver. 6*, to prevent security leaks, and additional reduction of processing time in ESP and AH was done, involving only clearing unused areas of the datagram rather than pre-clearing the entire datagram before filling it; the significant fix in ESP was done in *ver. 6*. The design was corrected to use the SMIB to look up the primitive numbers as is implicit in the layering scheme; algorithm numbers from the SADB had been passed-down – the SADB was changed to have the service number (intended to be obtained from the SA negotiation) set in the SADB. Future work may cause additional processing delays when multiple services, mechanisms and primitives cause full SMIB looking-up to be done, however, it probably will not be significant compared to processing times incurred by packet payloads.

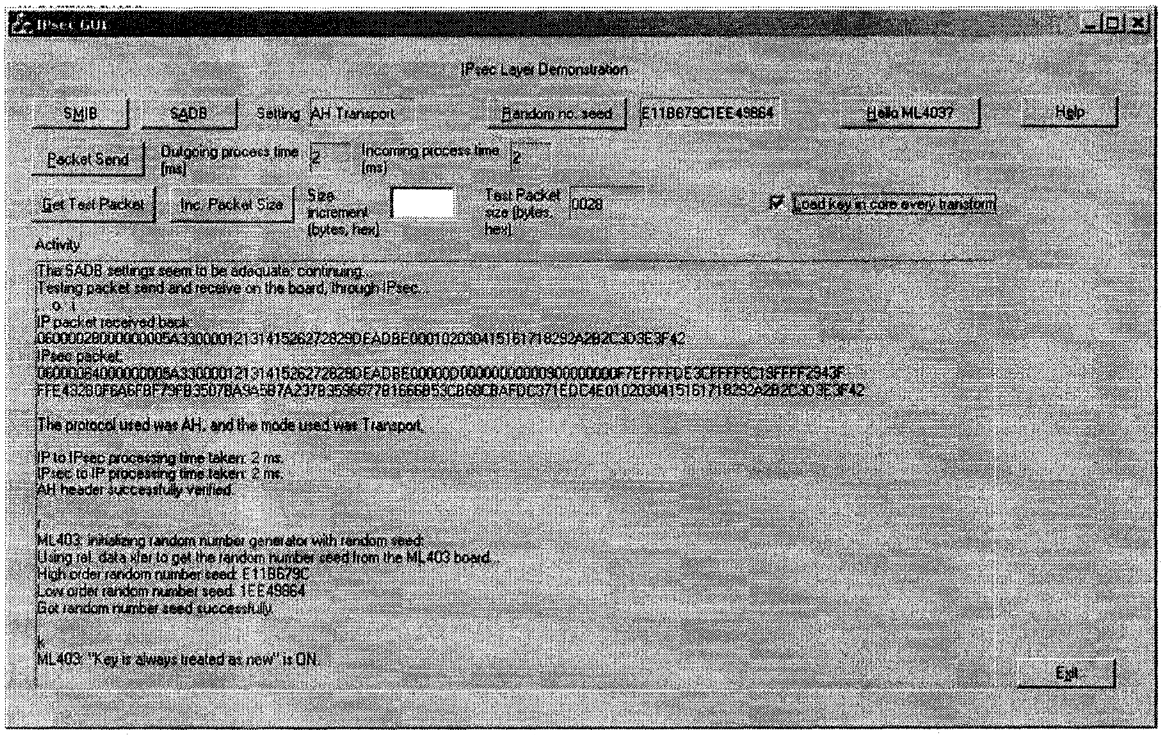
ver. 8: A transform pulse was added to follow the service call immediately, for accurate timing (see section 3.5., "Design of a CLI Version, 'IPsecLoop,' to Facilitate Testing," after). The ESP service was modified to create only the payload portion to hold the payload image to be transformed, and to remove some redundant header-setting code. The AH service was revised to simply use the IPsec datagram created for hashing as the datagram out when sending, instead of copying it to a new one. The encrypted authentication data produced when sending was placed into a temporary holding area for transfer to the IPsec datagram after the mechanism call. Some redundant header-setting code was removed. Both the HMAC and the Enc. mechanisms were revised to only get the key from the void pointer in the SADB when it is needed. Initialization was only necessary in one mechanism, which saves a little code space. Some common code across an if-else that was left when the SADB was simplified to one key and one IV was amalgamated. The mechanisms were revised to not retrieve the IV from the SADB's pointer if *NewIVNeeded*. The local key and IV variables were removed from the mechanisms. The mechanism-level key status tracker (L4AES128Mode) was revised to track the key in use via the SPI to uniquely identify the SADB and is now called

L4AES128SPI. The RNG was added as a field in the mechanisms layer in the SMIB. *MaxIPsecPacketSize* was added to the policy struct in the SMIB.

3.4. Design of a Test and Demonstration GUI, "IPsecGUI"

The GUI, named "IPsecGUI", and also written in MSVC++V6, supports all the features that were coded into the board: "hello", "key is new", random no. seed, SMIB and SADB receive and send. As in "AESfile", there is an activity window to display scrolling messages to the user. Text appearing in this activity window is also saved to a file, "ActivityLog.txt" for self-record-keeping. The SMIB and SADB settings can be read in and saved via file I/O. A test packet can be read in via file I/O and its total size is validated and corrected, if necessary. A test packet size increment can be set and used to increment the test packet size. Help information is shown in a "modeless" window that can be left open while working, for reference. The "Packet Send" button initiates the *PacketProcessing* of the OSI layers in the board, and the called functions in the GUI measure the processing speeds of the service calls in the board.

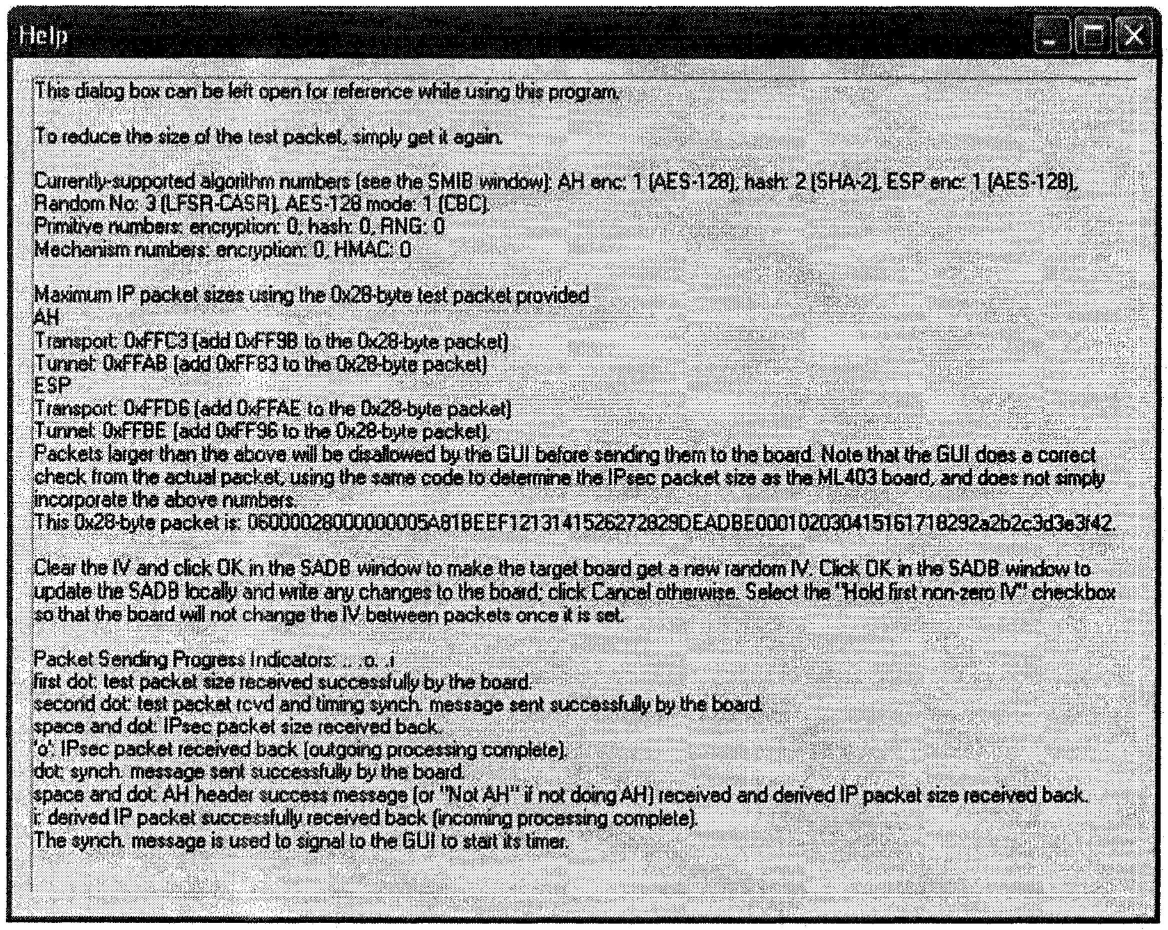
Figures 35 and 36 show the operation of the *KeyIsAlwaysTreatedAsNew*, i.e., "Load key in core every transform," the seeding of the RNG, and the help window.



Setting: AH Transport
Random no. seed: E11B679C1EE49864
Outgoing process time (ms): 2
Incoming process time (ms): 2
Size increment (bytes, hex):
Test Packet size (bytes, hex): 0028
[✓ Load key in core every transform]

```
The SADB settings seem to be adequate; continuing...
Testing packet send and receive on the board, through IPsec...
  o  i
IP packet received back:
06000028000000005A3300001213141526272829DEADBE000102030415161718292A2B2C3D3E3F42
IPsec packet:
06000064000000005A3300001213141526272829DEADBE00000D000000000000900000000F7EFFFFDE3CFFFF8C19FFFF2943F
FFE43280F6A6FBF79FB3507BA9A5B7A237B3596677B1666B53CB68CBAFDC371EDC4E0102030415161718292A2B2C3D3E3F42

The protocol used was AH, and the mode used was Transport.

IP to IPsec processing time taken: 2 ms.
IPsec to IP processing time taken: 2 ms.
AH header successfully verified.

r
ML403: initializing random number generator with random seed:
Using rel. data xfer to get the random number seed from the ML403 board...
High order random number seed: E11B679C
Low order random number seed: 1EE49864
Got random number seed successfully.

k
ML403: "Key is always treated as new" is ON.
```

Figure 35. Operation of "KeyIsAlwaysTreatedAsNew" and the seeding of the RNG in "IPsecGUI"



This dialog box can be left open for reference while using this program.

To reduce the size of the test packet, simply get it again.

Currently-supported algorithm numbers (see the SMIB window): AH enc: 1 (AES-128), hash: 2 (SHA-2), ESP enc: 1 (AES-128), Random No: 3 (LFSR-CASR), AES-128 mode: 1 (CBC).
Primitive numbers: encryption: 0, hash: 0, RNG: 0
Mechanism numbers: encryption: 0, HMAC: 0

Maximum IP packet sizes using the 0x28-byte test packet provided
AH
Transport: 0xFFC3 (add 0xFF9B to the 0x28-byte packet)
Tunnel: 0xFFAB (add 0xFF83 to the 0x28-byte packet)
ESP
Transport: 0xFFD6 (add 0xFFAE to the 0x28-byte packet)
Tunnel: 0xFFBE (add 0xFF96 to the 0x28-byte packet).
Packets larger than the above will be disallowed by the GUI before sending them to the board. Note that the GUI does a correct check from the actual packet, using the same code to determine the IPsec packet size as the ML403 board, and does not simply incorporate the above numbers.
This 0x28-byte packet is: 060000280000000005A81BEEF1213141526272829DEADBE000102030415161718292a2b2c3d3e3f42.

Clear the IV and click OK in the SADB window to make the target board get a new random IV. Click OK in the SADB window to update the SADB locally and write any changes to the board; click Cancel otherwise. Select the "Hold first non-zero IV" checkbox so that the board will not change the IV between packets once it is set.

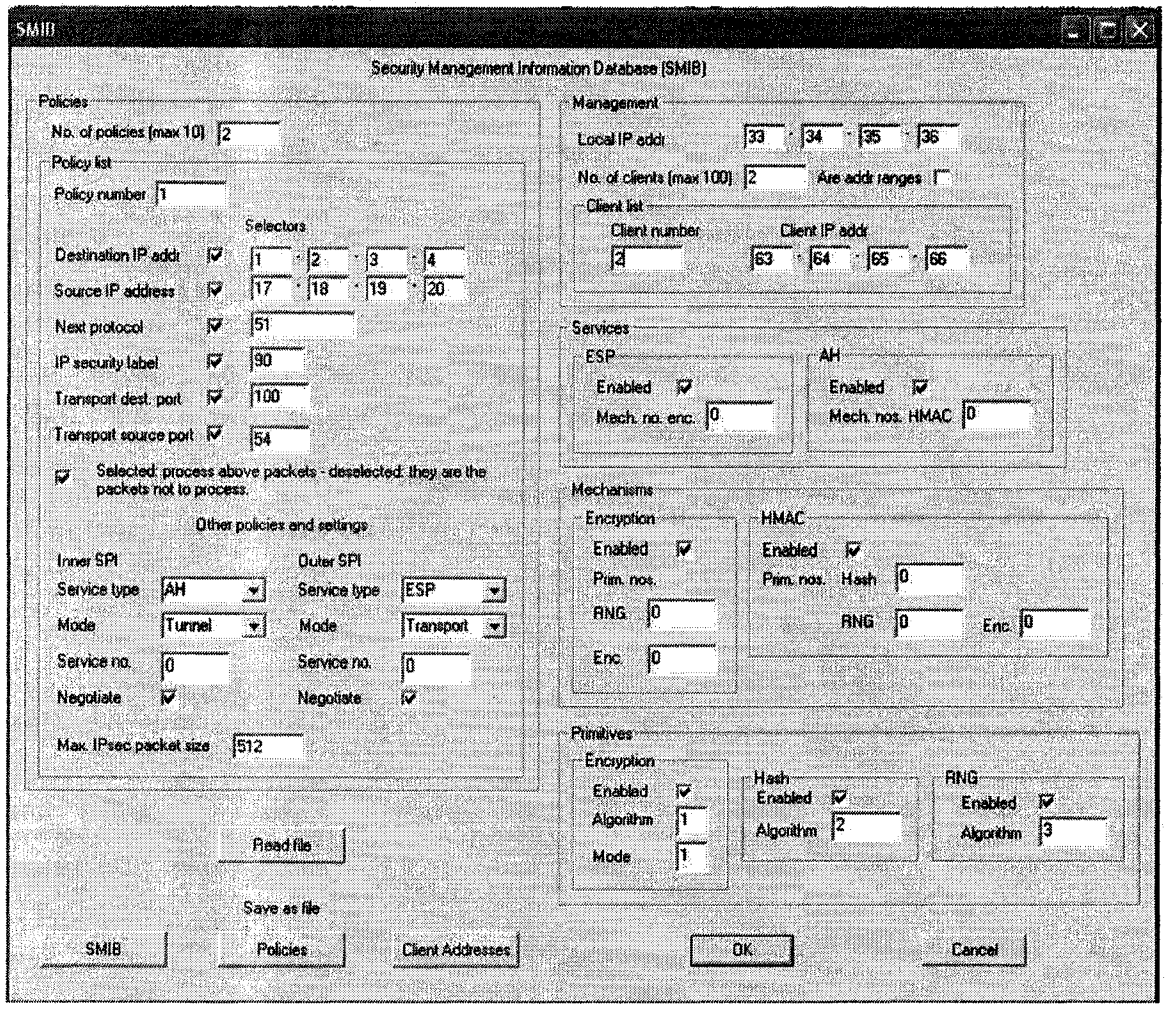
Packet Sending Progress Indicators: .. .o. .i
first dot: test packet size received successfully by the board.
second dot: test packet rcvd and timing synch. message sent successfully by the board.
space and dot: IPsec packet size received back.
'o': IPsec packet received back (outgoing processing complete).
dot: synch. message sent successfully by the board.
space and dot: AH header success message (or "Not AH" if not doing AH) received and derived IP packet size received back.
i: derived IP packet successfully received back (incoming processing complete).
The synch. message is used to signal to the GUI to start its timer.

Figure 36. The "IPsecGUI" help window

When the GUI launches, it first gives the 'h' command to check whether the board is responding. If "Hello" is not received back, an error message is displayed to the user. If the board is responding, reliable data transfer is used to get the SMIB and the SADB from the board. In this way, update of context to the PC is automatic. If the user chooses to access the SADB dialog, the SADB is also acquired from the board, including the key and IV pointed to, since the board changes the IV – and changing the key is forseen, in future work, when key exchange is added.

Multiple policies are supported by the GUI. A GUI field is provided for the user to enter the number of policies, and another is provided for the user to select the particular policy to view. When the former field is changed, the space allocated is changed to hold the required number of policies, in an array of policy structures. When the latter field is changed, the policy fields are updated with the data of the policy number selected. When a data field is changed, the entry in its policy structure is updated. If "OK" is clicked in the SMIB window, those policy structures are incorporated as the official SMIB array of policy structures, and the former array of policy structures is discarded. If the SMIB window is exited without clicking "OK", such as by clicking "Cancel", that is not done, leaving the official array of policy structures previously entered, unchanged. In addition, if "OK" is clicked, the SMIB in the board is updated via reliable data transfer and the policy that was selected to view in the SMIB dialog window is sent to the board. The same technique is used to implement the entry of multiple clients in the management layer of the SMIB.

Figure 37 shows the SMIB window, or dialog box.


SMIB

Security Management Information Database (SMIB)

Policies
No. of policies (max 10) 2
Policy list
Policy number 1
Selectors
Destination IP addr ✓ 1 . 2 . 3 . 4
Source IP address ✓ 17 . 18 . 19 . 20
Next protocol ✓ 51
IP security label ✓ 90
Transport dest. port ✓ 100
Transport source port ✓ 54
✓ Selected: process above packets - deselected: they are the packets not to process.
Other policies and settings
Inner SPI
Service type AH
Mode Tunnel
Service no. 0
Negotiate ✓
Outer SPI
Service type ESP
Mode Transport
Service no. 0
Negotiate ✓
Max. IPsec packet size 512

Management
Local IP addr 33 . 34 . 35 . 36
No. of clients (max 100) 2 Are addr ranges ☐
Client list
Client number 2
Client IP addr 63 . 64 . 65 . 66

Services
ESP
Enabled ✓
Mech. no. enc. 0
AH
Enabled ✓
Mech. nos. HMAC 0

Mechanisms
Encryption
Enabled ✓
Prim. nos.
RNG 0
Enc. 0
HMAC
Enabled ✓
Prim. nos. Hash 0
RNG 0
Enc. 0

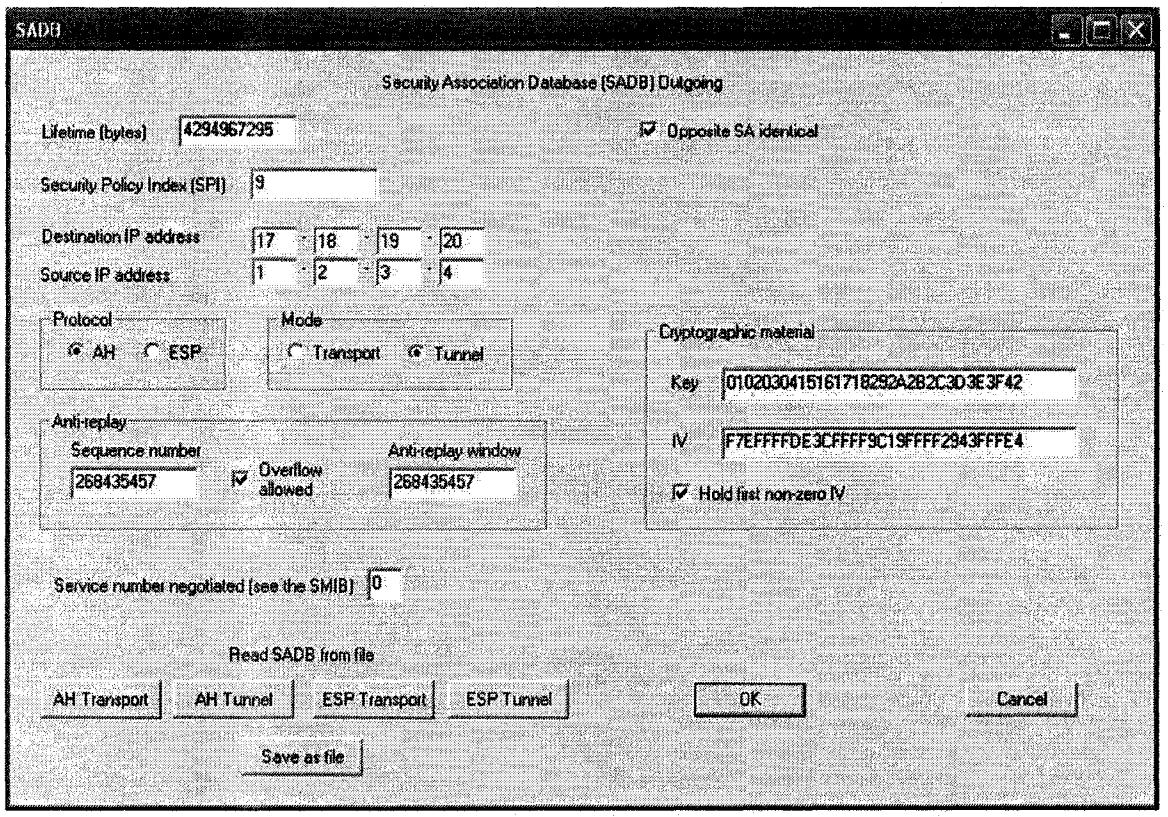
Primitives
Encryption
Enabled ✓
Algorithm 1
Mode 1
Hash
Enabled ✓
Algorithm 2
RNG
Enabled ✓
Algorithm 3

Read file
Save as file
SMIB Policies Client Addresses
OK Cancel

Figure 37. The "IPsecGUI" SMIB dialog

Similarly, when the user exits the SADB, it is updated to the board via reliable data transfer if "OK" was clicked.

Figure 38 shows the SADB dialog box, with the first non-zero IV produced by the PRNG from a seed of all "1"s (or "F"s, in hex.).


SADB

Security Association Database (SADB) Outgoing

Lifetime (bytes) 4294967295

[✓] Opposite SA identical

Security Policy Index (SPI) 9

Destination IP address 17 . 18 . 19 . 20

Source IP address 1 . 2 . 3 . 4

Protocol: [● AH] ○ ESP

Mode: ○ Transport [● Tunnel]

Cryptographic material

Key 0102030415161718292A2B2C3D3E3F42

IV F7EFFFFDE3CFFFF9C19FFFF2943FFFE4

[✓] Hold first non-zero IV

Anti-replay

Sequence number 268435457

[✓] Overflow allowed

Anti-replay window 268435457

Service number negotiated (see the SMIB) 0

Figure 38. The "IPsecGUI" SADB dialog

There is a checkbox provided for setting the *IV Constant* field in the SADB, "Hold first non-zero IV." See the explanation before, in section 3.3., "Design of an IPsec Implementation, 'IPsecImp' [etc.]," for the use of this field.

In sending these structures as a block, it should be noted that the GNU C code in the board uses big-endian data orientation, whereas the MSVC++V6 code running on the PC uses little-endian. The GUI is responsible for correcting the data orientation after receiving from the board and before sending back to the board, by switching the order of the bytes within each variable affected; a series of functions were written to do this.

When the "Packet Send" button is clicked, the GUI verifies whether the SMIB exists and has been set, and whether the SADB exists, including the key and IV, and has been set. The existence of the test packet is checked, and its header is checked for the minimum size. Using the same code that the board uses to obtain the IPsec packet size, the GUI checks to make sure that the resulting IPsec packet would not be larger than 64kB-1 bytes, or 0xFFFF (65,535) bytes, and also that it would not be larger than the maximum packet size set in the policy. If everything is verified satisfactorily, the 'p' command is issued to the board to start packet processing. Reliable data transfer is used to send the test packet to the board when the IP layer requires it. The synchronization message, "Synch", is received, using reliable data transfer, in order to synchronize with the beginning of the service call. The time count variables are cleared, and a timer that counts milliseconds is turned on. Following this, execution occurs in the timer message-handling routine. Figure 39 illustrates the design of this function.

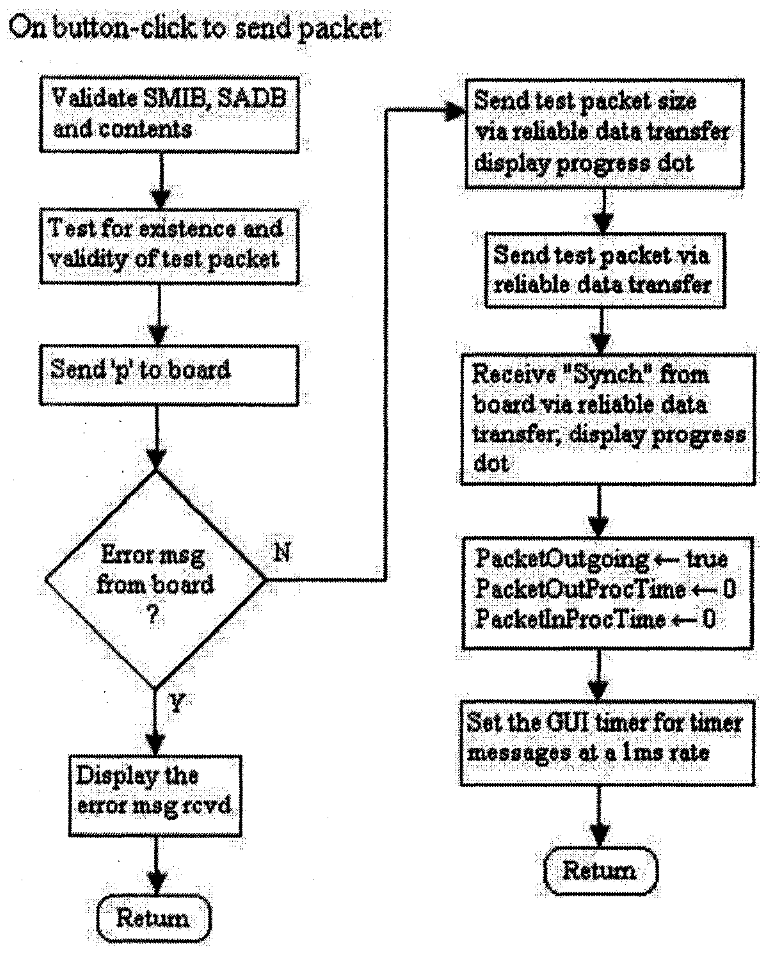

On button-click to send packet
Validate SMIB, SADB and contents
Test for existence and validity of test packet
Send 'p' to board
Error msg from board ?
N
Y
Display the error msg rcvd
Return
Send test packet size via reliable data transfer display progress dot
Send test packet via reliable data transfer
Receive "Synch" from board via reliable data transfer; display progress dot
PacketOutgoing ← true
PacketOutProcTime ← 0
PacketInProcTime ← 0
Set the GUI timer for timer messages at a 1ms rate
Return

Figure 39. "IPsecGUI" packet processing setup flowchart

In the timer message-handling routine, the quickest possible check is made for any character received via RS232. The time count variables are updated, and the routine exits if no character was received. This is done since timer messages are ignored if the MSVC++V6 program is busy, and missing timer messages would result in a low time count. If a character was received, the timer is stopped, and reliable data transfer is used to get the IPsec packet. The synchronization message is received again, and the same process is used to time the "incoming", or "receiving", service call, in which the IPsec packet is converted back to an IP packet. During this process, minimal echoing is shown in the activity window, in order not to cause difficulties in synchronizing the GUI with the board, which could cause problems such as communication lockups. Packet Sending Progress Indicators shown are: ".. .o. .i", which have the following meanings:

first dot: test packet size received successfully by the board.

second dot: test packet rcvd and timing synch. message sent successfully by the board.

space and dot: IPsec packet size received back.

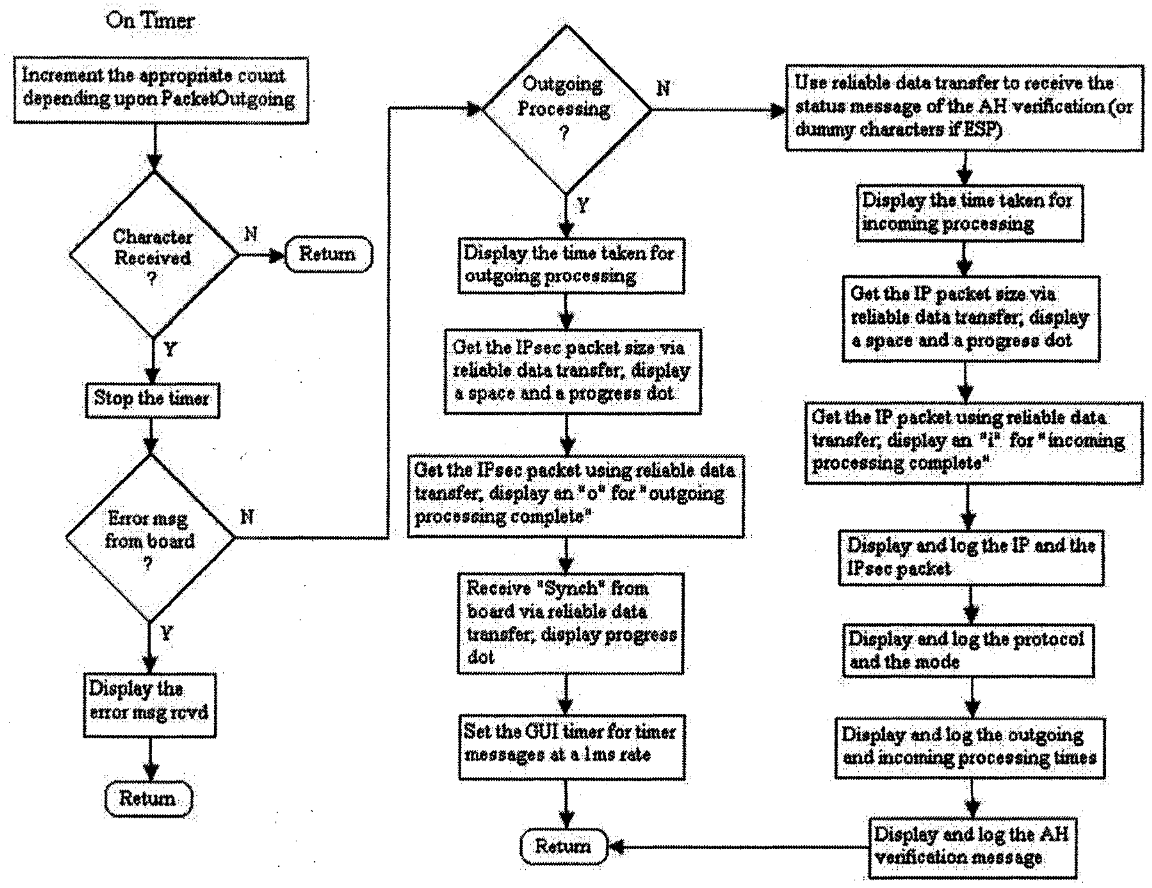
'o': IPsec packet received back (outgoing processing complete).

dot: synch. message sent successfully by the board.

space and dot: AH header success message (or "Not AH" if not doing AH) received and derived IP packet size received back.

'i': derived IP packet successfully received back (incoming processing complete).

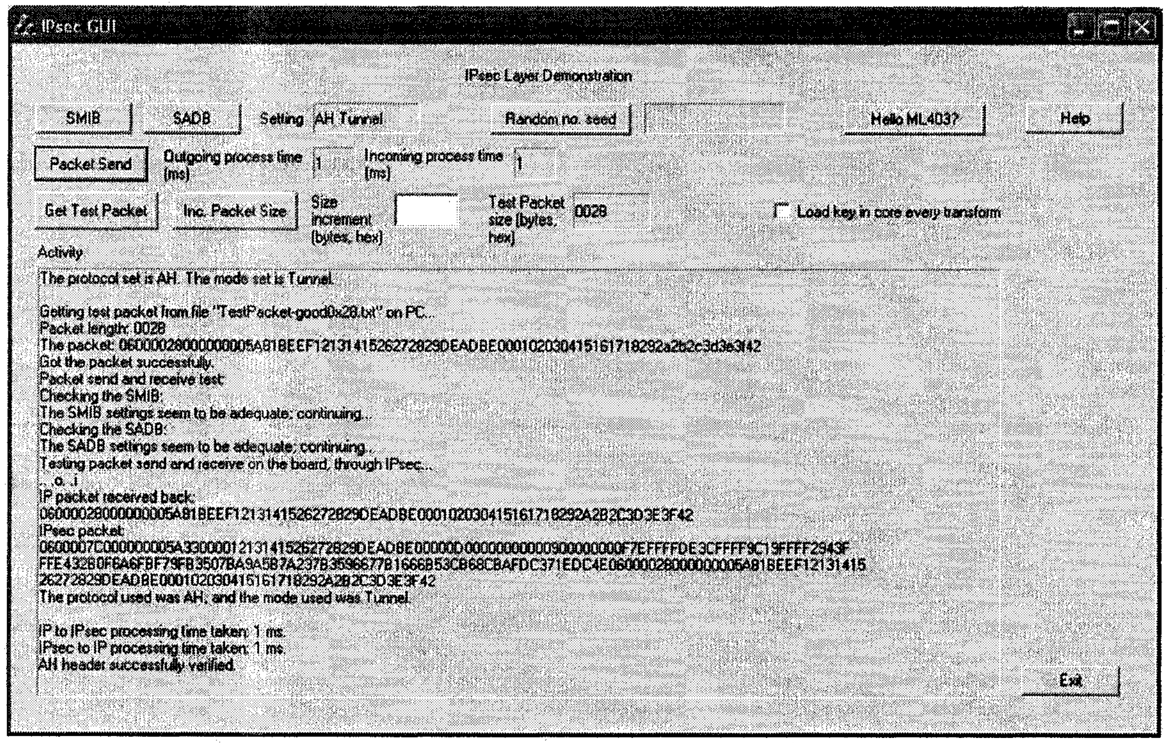
Figure 40 illustrates the design of this function.



On Timer
Increment the appropriate count depending upon PacketOutgoing
Character Received ?
N
Return
Y
Stop the timer
Error msg from board ?
N
Y
Display the error msg rcvd
Return
Outgoing Processing ?
N
Y
Display the time taken for outgoing processing
Get the IPsec packet size via reliable data transfer; display a space and a progress dot
Get the IPsec packet using reliable data transfer; display an "o" for "outgoing processing complete"
Receive "Synch" from board via reliable data transfer; display progress dot
Set the GUI timer for timer messages at a 1ms rate
Return
Use reliable data transfer to receive the status message of the AH verification (or dummy characters if ESP)
Display the time taken for incoming processing
Get the IP packet size via reliable data transfer; display a space and a progress dot
Get the IP packet using reliable data transfer; display an "i" for "incoming processing complete"
Display and log the IP and the IPsec packet
Display and log the protocol and the mode
Display and log the outgoing and incoming processing times
Display and log the AH verification message

Figure 40. "IPsecGUI" packet processing timer processing flowchart

Figure 41 shows a "screenshot" of an example of the IPsecGUI's operation.



Setting: AH Tunnel
Outgoing process time (ms): 1
Incoming process time (ms): 1
Size increment (bytes, hex):
Test Packet size (bytes, hex): 0028
☐ Load key in core every transform

Activity

```
The protocol set is AH. The mode set is Tunnel.

Getting test packet from file "TestPacket-good0x28.txt" on PC...
Packet length: 0028
The packet: 06000028000000005A81BEEF121314152627282 9DEADBE000102030415161718292a2b2c3d3e3f42
Got the packet successfully.
Packet send and receive test:
Checking the SMIB:
The SMIB settings seem to be adequate; continuing...
Checking the SADB:
The SADB settings seem to be adequate; continuing...
Testing packet send and receive on the board, through IPsec...
...o...i
IP packet received back:
06000028000000005A81BEEF1213141526272829DEADBE000102030415161718292A2B2C3D3E3F42
IPsec packet:
0600007C000000005A3300001213141526272829DEADBE00000D0000000000000900000000F7EFFFFDE3CFFFF9C19FFFF2943F
FFE4328 0F6A6FBF79FB3507BA9A5B7A237B3596677B1666B53CB68CBAFDC371EDC4E06000028000000005A81BEEF12131415
26272829DEADBE000102030415161718292A2B2C3D3E3F42
The protocol used was AH; and the mode used was Tunnel.

IP to IPsec processing time taken: 1 ms.
IPsec to IP processing time taken: 1 ms.
AH header successfully verified.
```

Figure 41. A "screenshot" of the operation of "IPsecGUI"

See also the pseudo-code for the "IPsecGUI" packet processing functions, in Appendix A.

3.4.1. Test Methodology

Testing the time required for the service calls to complete was done using the IPsecGUI packet processing functions described before. The times were echoed to the activity window, and thus to the activity log file. A standard 40-byte (0x28) test packet with a six-32-bit-word (twenty-four bytes) header was used, and incremented in units of 0x100 and 0x200 bytes up to 0x4028 (approximately 16k) bytes. This range was used in testing all four protocol-mode combinations: AH protocol, Transport and Tunnel modes, and ESP protocol, Transport and Tunnel modes. The packet used was as follows (hex., with spaces added for readability):

```
        06000028 00000000 5A81BEEF 12131415 26272829 DEADBE00 01020304
15161718 292a2b2c 3d3e3f42
```

A packet of the smallest size possible was tested just to verify correct operation – note the one-byte payload and the five-long-word header:

```
        05000015 00000000 5A81BEEF 12131415 26272829 A5
```

Also, a packet with the largest possible header, of fifteen, 0xF, "long words" (32-bit words), and a one-byte payload, was also tested, just to verify correct operation:

```
        0F00003D 00000000 5A81BEEF 12131415 26272829 DEADBE00 10000001
10000001 10000001 10000001 10000001 10000001 10000001 10000001 10000001
A5
```

The typical size of a packet is about 500 bytes, 576 bytes to be exact ([RFC0791] pg.12), and "Datagrams are rarely larger than 1,500 bytes" ([KUR2000], 3rd ed., pg. 326). In an example from the literature, IPsec was tested only up to a packet size of about 8k (0x2000) bytes [KER1997]. The minimum packet size includes a one-byte payload ([RFC0791] pg 34). In testing in this work, operation with an IP packet of the minimum size and up to an IPsec packet of the maximum size of 0xFFFF bytes was tested. However, "Such long datagrams are impractical for most hosts and networks" ([RFC0791] pg.12). The resulting millisecond counts were graphed.

Testing was done to verify that the AES encryption was appearing correctly. ESP Transport mode was used to encrypt a packet payload consisting, in part, of the block "01020304 15161718 292A2B2C 3D3E3F42," using the first IV produced by the RNG from a seed of all "F"s (hex.), which is F7EFFFFD E3CFFFF9 C19FFFF2 943FFFE4. The key value was arbitrarily chosen as the same as the plaintext. To follow the process of Cipher-Block Chaining (CBC mode), these were XORed using Windows Calculator to obtain F6EDFCF9 F6D9E8E1 E8B5D4DE A901C0A6. To predict the ciphertext that should appear, the Styer Javascript example [STY2006] was used, which predicted "F1E73B95 E690F3BA 45CF3F0B DDD92594." An ESP Transport test was done using the 40-byte test packet, which contained that block as a payload, first leaving the IV in the SADB set to zero, and a freshly-initialized board that had the RNG seed set to all "F"s (hex.). Additionally, the comprehensive CLI demonstration program done to demonstrate the operation of the AES core described in section 3.1.8.1., "CLI and Simpler Programs," was also used to verify the output of these "higher-level" programs.

In addition, the “IPsecGUI” (as well as “IPsecLoop”) was used many times to encrypt and decrypt packets using all four protocol-mode combinations, and the resulting IPsec and received-back IP packets were verified for correctness.

3.5. Design of a CLI Version, “IPsecLoop,” to Facilitate Testing

Since it was unknown how accurate the method of testing described before using “IPsecGUI” would be, oscilloscope timing measurements were planned, with the idea that the service calls would be iterated. There was no point in developing a GUI to operate such a program, therefore “IPsecLoop” has no GUI, and is a CLI program, operated via Hyperterminal. Also, it only differs in the top-level loop and OSI layers; all of L3 (Service) to L5 (Primitive) layers are identical – the same C files are used as in “IPsecImp”.

Referring to the pseudo-code (see Appendix A), “IPsecLoop” differs from “IPsecImp” in having a pre-set SADB set in initialization, before the main command loop. Also, the test packet is built-in by allocating memory for it and setting its contents during program initialization. In the command loop, ‘r’ to seed the PRNG, ‘z’ and ‘s’ to send and receive the SADB, and ‘m’ and ‘i’ to send and receive the SMIB, are not needed and are removed. Instead, the following commands were added: ‘o’, to toggle testing between outgoing (sending) and incoming (receiving), ‘t’, to change the protocol and mode to be tested, ‘i’, to change the increment by which to increase the test packet size, ‘s’, to increment the test packet size or revert to the original size, and ‘n’, to toggle *NeedNewIV*. Commands retained are ‘h’, for “Hello” from the board, ‘k’ to toggle *KeyIsAlwaysTreatedAsNew*, and ‘p’, to start packet processing, i.e., launch the test.

In the IPsec sublayer function, an infinite loop was placed around each of the services, which activates if its protocol and processing direction are set. A message is echoed to the user to indicate that outgoing, or incoming, processing is occurring. *Slv_reg8* of the AES core of the opposite transform from that employed by the service call for that direction is written in order to cause autoload and create the diagnostic pulse previously described, in order to mark the timing of the loop and make it readable by an oscilloscope. Since the service call contains many lines of calculation and memory allocation, and there are two more layers below it, it was expected that the duration of the transform pulse would be about three orders of magnitude shorter than the duration of the service call, leading to a precise reading. Moreover, a reading to the nearest 220 ns due to the transform pulse is about three and a half orders of magnitude more precise than the IPsecGUI measurement, which is to the nearest millisecond.

Included in the service call loop are (as noted in Appendix A):

a dummy transform to read via oscilloscope to mark the loop (the opposite one to the one used in the service call)

the service call

a dummy transform to read via oscilloscope to mark the end of the service call (added in *ver. 8* of "IPsecLoop")

test for error

set *KeyIsNew* from top-level user selection, *KeyIsAlwaysTreatedAsNew*

check for a keystroke via RS232 and exit the loop if so

check for the setting of the "Outgoing" vs. incoming test selection to exit the loop after only one execution if testing the other direction.

delete the transformed packet if looping so as not to use up the memory

Since the extra instructions required for continuous looping would cause some delay, *ver. 8* of "IPsecLoop" was revised to add an extra dummy transform immediately after the service call, for accurate timing.

3.5.1. Test Methodology

The test methodology to use "IPsecLoop" is fairly straightforward. Oscilloscope probes are connected to pins 2 and 64 of J6 so that the decryption core "loadtodone" signal (pin 2), is connected to channel 1 (the upper trace in Figure 42) and the encryption core "loadtodone" signal (pin 64) is connected to channel 2 (the lower trace in Figure 42). Hyperterminal is used to operate the program running in the ML403 board. 'p' is pressed to begin the first test. The reading is read from the oscilloscope, in units of screen divisions, in order to work carefully from the raw data. The "units per division" setting is carefully noted. A key is pressed to exit the loop. The processed packet is echoed, following that (if incoming processing is being tested, the IPsec packet is echoed before the program pauses during the incoming processing). The 'o' command is used to toggle the test to the opposite direction. The 'i' command is used to select the packet increment amount and the 's' command is used to increment the packet size. The packet size, starting from the standard 40 (0x28)-byte packet (see section 3.4.1., "Test Methodology" – "IPsecGUI") is incremented until a range of packet sizes from 0x28 to 0x4028 bytes for that protocol and mode are tested, then the process is repeated for the other three protocol-mode combinations.

To show the method used (without the additional marker pulse following the service call, which would be close to the second marker pulse for the beginning of the next loop), Figure 42 shows an oscilloscope measurement of AH Transport outgoing processing time with the 40 (0x28 hex.)-byte packet, using IPsecLoop *ver. 6*. The oscilloscope used was the Tektronix TDS1002 Two-Channel Digital Storage Oscilloscope [TEKTDS]. Tektronix TDSPCS1 "Open Choice" PC Communications Software, Version 1.10, was installed on the PC and a 9-pin RS232 full "cross-cable" or "null modem" (the RS232 handshaking lines were also crossed) cable was used to connect an RS232 serial port on the PC to the RS232 port on the oscilloscope.

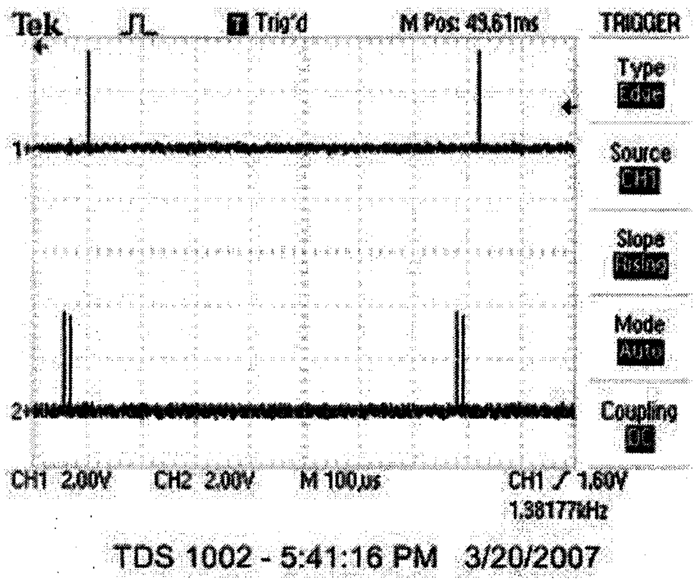


Figure 42. Oscilloscope measurement of AH Transport outgoing processing time with a 40-byte packet

The ‘k’ and ‘i’ commands only need to be toggled on once or twice, preferably when the packet size being tested is the smallest, so that the additional processing time to load the key into the core or get the IV from the PRNG can be most easily determined from the total processing time read from the oscilloscope. Since the IV is in the data in the incoming (receiving) processing, setting *NeedNewIV* does not cause a new IV to be acquired for that computation, of course. The time taken to load the key and/or the IV was not included in the series of measurements of packet processing time.

CHAPTER IV

ANALYSIS OF RESULTS

4.1. The AES Accelerator

4.1.1. Simulation Results

Referring to the large "screenshots" in Appendix B, section B.1., Figures 61, 62, 63 and 64 in section B.1.1.1. show simulation of an encryption transform. Using the key shown, of value 01020304 15161718 292A2B2C 3D3E3F42 (hex.), and *text_in*, of 41747461 636B2061 74206461 776E2020 (hex.) – which is the ASCII for "Attack at dawn " (note the two spaces at the end that make up the sixteen bytes of a 128-bit block), a ciphertext, or *text_out*, of 03211ECA A144E6D0 7FF9F6D9 1801D80C (hex.) was produced, and was verified (see section 3.1.6.1., "Simulation Test Methodology"). The "load pulse" process formerly used can be seen. *w0-w3* are the key expand values, beginning with the unchanged value of the key itself. The *sa* variables are the results of the XOR of the state (beginning with *text_in*), the *sa_sub* variables are the results of the sub-byte operation, and the *sa_mc* variables are the results of the mix columns step.

In section B.1.1.2., verification with autoload can be seen in Figure 65. Note that *start_load* starts when *slv_reg8* is written.

In section B.1.1.3., two encryptions using CBC mode can be seen in Figure 66. During the time of the *donepulse*, the ciphertext in *slv_regs9-12* is copied into *iv0-3*. The results of XOR with the plaintext can be seen where *text_in* changes. The differing ciphertext can be seen, as produced by the chained plaintext.

The "screenshot" in section B.1.2., "Decryption" (Figure 67) shows the inverse transform from the chained ciphertext in *slv_regs5-8*. The previous ciphertext can be seen, just prior, in *slv_regs5-8*, being used to load *iv0-3* during *ivload*. When *slv_reg8* is written, the transform commences (which was designed not to happen when *ivload* is "1"), and the familiar ASCII for "Attack at dawn " can be seen in *slv_regs9-12*, occurring during *donepulse*, when the decrypted output is XORed with the IV to form the plaintext. During *done2pulse*, the previous ciphertext is seen being transferred to the IV. In *slv_regs1-4*, the last (in encryption order) key expand value loaded to the decryptor can be seen, being the first used for decryption in *w0-3*. (*slv_regs1-4* correspond to *w3-0*). The last key expand value used in *w0-3* can be seen to be the key itself, opposite from encryption order. *Slv_reg0* bit 6 can be somewhat discerned as being the source of *ivload* (0x40 when the only bit set in *slv_reg0*).

4.1.2. FPGA Usage

Table 5 shows the FPGA device utilization summary when both the encryptor and the decryptor core, with all of the features described before, were included.

```
      Number of BUFGs                         7 out of 32      21%
      Number of DCM_ADVs                      2 out of 4       50%
      Number of ILOGICs                      33 out of 320     10%
      Number of External IOBs                68 out of 320     21%
         Number of LOCed IOBs                68 out of 68     100%
      Number of JTAGPPCs                      1 out of 1      100%
      Number of OLOGICs                      63 out of 320     19%
      Number of PPC405_ADVs                   1 out of 1      100%
      Number of RAMB16s                      36 out of 36     100%
      Number of Slices                     5470 out of 5472    99%
         Number of SLICEMs                  666 out of 2736    24%

      Number of Slices containing only related logic:   4,635 out of
5,470   84%
      Number of Slices containing unrelated logic:        835 out of
5,470   15%
```

Table 5. XC4VFX12 device utilization summary (most recent build – Mar. 12, 2007)

The most recent build was only required due to the change to “Daylight Saving Time” (DST), which was held early in 2007, when the Windows XP PC Operating System (OS) changed the EDK/ISE project file times retroactively to those of the changed “time zone”. Previous builds were done on Dec 10 and Nov 18, 2006.

As can be seen, usage has moved into using slices for unrelated logic, 99% of the slices being used. Place and route required 54 minutes; that time begins to grow rapidly as the FPGA is filled and slices have to be used for unrelated logic.

4.1.3. AES Performance Results

4.1.3.1. AES Core Performance using Small Software Test Programs

Please refer to the methodology that can be found in sections 3.1.7.2.4., “Timing Diagnostic Output for Test,” and 3.1.8.1., “CLI and Simpler Programs”.

When the original method of pulsing “load” to begin the transform was used, the load pulse in the encryptor was observed to take 510 ns and “loadtodone” was observed to take 730 ns. 730 - 510 = 220 ns, which is eleven 20 ns clock pulses as expected. Clearly, turning a bit on and off via software over the bus is slow.

Table 6 shows some of the most relevant observations that were made. The calculated SW transform rate was calculated by multiplying 128 by the SW loop rate (or by dividing it by the period).

Observed AES core performance used by software and in-FPGA								
Date (2006)	Load (go) method or test description	Core	SW loop speed		(Calculated) SW transform rate	FPGA transform time		
			Period (µs)	Rate (kHz)	(128-bit blocks) (Mbps)	Load pulse (ns)	Load-to-Done (ns)	Transform time (ns)
10-Oct	Pulse	Encryptor	2.26	442	56.637168	510	730	220
21-Nov	Autoload	Decryptor	1.78	559	71.910112	NA	200	200
11-Dec	Full write and read back	Encryptor	5.3	189	24.150943	NA	NA	220
	Full write and read back	Decryptor	5.76	174	22.222222	NA	NA	200
	Full write and read back	Both	10.18	98.2	12.573674	NA	NA	NA
	Full write and read back (D)	Both	8.94	111.9	14.317673	NA	NA	NA
Pulse load: tested with OP to a pin while the load pulse was high while also OP to a pin until done is high.								
Autoload: test by only outputting to a pin from time of internal load (ld_r) until done asserted								
note-Autoload used in all subsequent tests								
(D): Not even testing for the Done bit in software, since the FPGA is so fast								
NA: Not Applicable								

Table 6. Observed AES performance in-core and as used by software

"Full write and read back" means that the time taken to write all four input and read all four output slave registers via software was included in the test loop. The test of both means that the output of the encryptor was fed into the input of the decryptor by reading and writing the slave registers via software.

Note that if the time taken to turn the load bit on and then off is 510 ns, then the time for one write is 255 ns. If the time taken to read a slave register is the same, that means that it is hardly worth using a software loop to check for the *done* bit in *slv_reg0*, since the transform takes only 200-220 ns.

By initializing the step counter in the decryptor and using a different timing plan, the clock cycles required were reduced from eleven to ten.

220 ns to encrypt 128 bits implies 128/220 = 582 Mbps, and 200 ns to decrypt implies 128/200 = 640 Mbps as an intrinsic core transform rate; however, some method, such as DMA, will be needed to transfer data to and from the core as fast as it can be processed. Xilinx offers a technology called FSL (Fast Simplex Link) that can import data from user IP cores to its soft-core processors, but this method is not available when a hard-core processor, as in this work, is used.

Processing via software intervention is slow. Even if the time required to write the input slave registers, wait for the transform to finish and read the output slave registers could be reduced by removing the wait loop on the grounds that the transform happens so quickly compared to software operation that a wait loop is not needed, that would only save about 600 ns from one of the single transform times noted in Table 6 (removing it when it was used twice when both transforms were done, saved 1.24 μs, as can be seen from Table 6). If the encryption time, for example, were thus reduced to 4.7 μs, that would still imply a processing rate of only 27 Mbps, 21 times slower than the core's capacity of 582 Mbps.

Table 7 compares the intrinsic core transform rates of this work with those of others' found in the literature.

AES core rates achieved, by implementation (Mbps)						
	This work	[MCL2002]	[DAN2000]	[KIM2004]	[BEL2002]	[LUJ2005]
Encryption	582	310	353	390	887	1197
Decryption	640					

This work (Fall 2006): Xilinx Virtex-4 FPGA, XC4VFX12, speed grade 10 (slowest)

[DAN2000]: all on an FPGA (make, model unspecified)

[BEL2002]: Xilinx Virtex 1000 FPGA

[MCL2002]: XCV1000E

[KIM2004]: FPGA not specified

[LUJ2005]: Xilinx Virtex-II Pro 100 XC2VP100

Table 7. AES core rates achieved in this work and in the literature

Since the FPGA used in this work is the lowest speed grade [XILRIV], it may be possible to double its processing rates by running the core at the full 100 MHz clock speed (see section 3.1.7.1., "Core Design and other Modifications").

4.1.3.2. AES Performance with the "AESfile" GUI

Please refer to the test methodology described in section 3.1.8.2.2., "The PC Demonstration GUI, 'AESfile'."

"AESfile" encrypts a 36kB text file in 24s and decrypts it (75 kB of ciphertext) in 36s. 36kB/24s = 1.5 kBps = 1.5 x 8 = 12 kbps (57600 bps serial connection). 75k/36 = 25k/12 = 2.1 kBps = 2.1 x 8 = 17 kbps. The serial communication speed, display delays in the PC application, as well as the software calls in the board for each 128-bit block, all contribute to this slowdown. The simple calculation of the encryption and decryption rates here may not be precisely meaningful, because in both processes, there is a transmission of 16 bytes of plaintext and 32 bytes of hex-ASCII-encoded ciphertext for each block transformed, meaning that both processing times should be the same. The longer time taken for decryption is explained by the extra display echoing included in the decryption process (see the figure in section 3.1.8.2.2., "The PC Demonstration GUI, 'AESfile'").

Here is an example of the ciphertext output from "AESfile" (notes added on the right, afterward). A key and IV of zero were used to encrypt the plaintext:"Attack at dawn Attack at dawn ". Two spaces were added to each phrase to make each phrase a block of 16 bytes; the quotes were not included in the plaintext.

```
66E94BD4EF8A2C3B884CFA59CA342B2E - IV encrypted in ECB
06362F5ED752BD8A1A2D8AFF2D887988 - "Attack at dawn  "
5049804CC352A02E3B6E2BB6EB55E548 - "Attack at dawn  "
8A8BC72E24DCFE7DFF6F89065BE13599 - 0x03 (ETX) padded with 0's.
```

This output format looks regular and pleasing to the eye, somewhat reminiscent of the "PGP" style of output [PGPI], and is suitable in the same way, for convenient copying and pasting for emailing, since it is ASCII text.

4.2. The LFSR-CASR PRNG

Please refer to section 3.2.3., "Test Methodology and Use in this Work", for a description of how the graphs that follow were produced (the section just preceding that one refers to tests already done on this PRNG as documented in a corporate paper). The four graphs that follow (Figures 43, 44, 45 and 46) show that the distribution of the pseudo-random numbers produced by the LFSR-CASR PRNG is reasonably uniform, at least for the quantities of numbers produced.

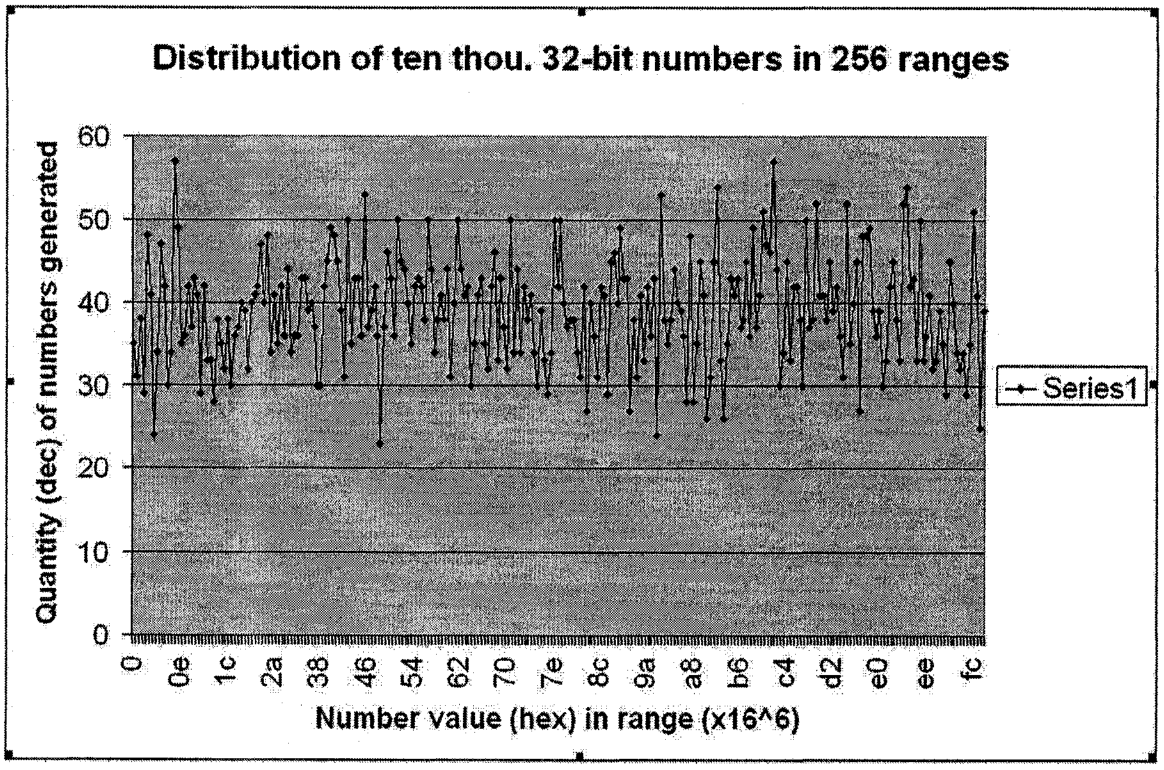


Figure 43. Distribution of ten thou. 32-bit numbers from the LFSR-CASR PRNG in 256 ranges

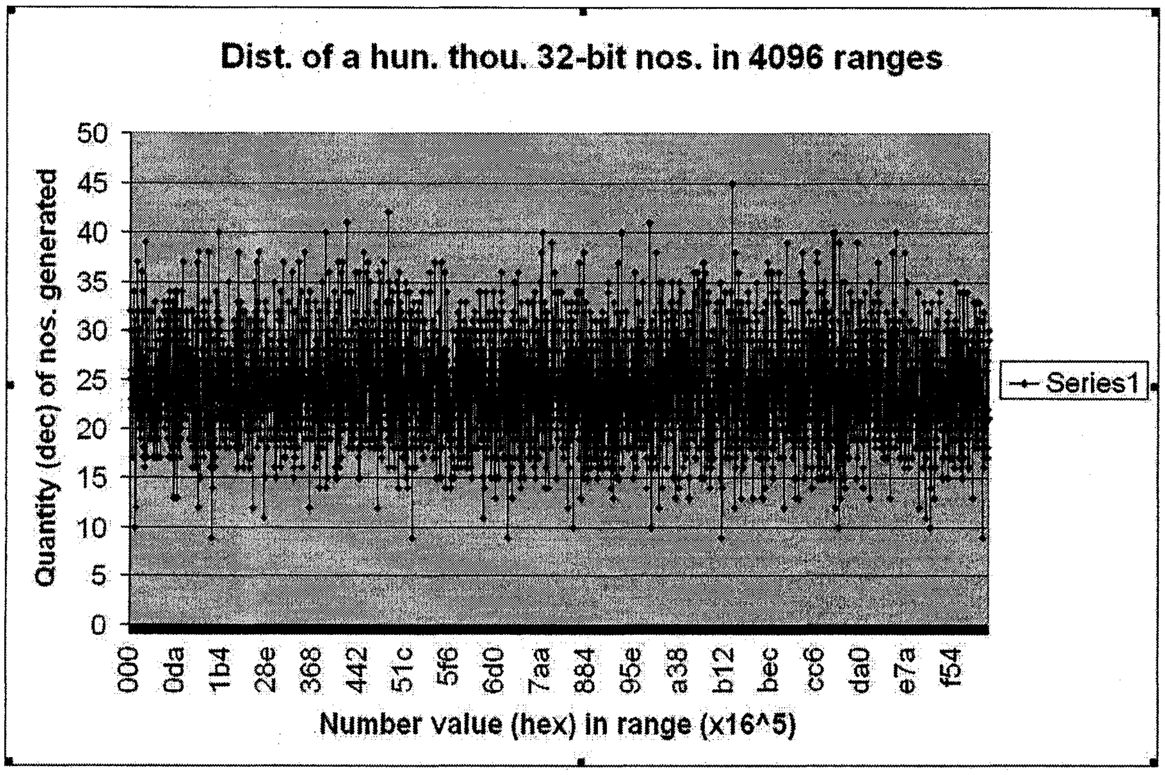


Figure 44. Distribution of a hun. thou. 32-bit numbers from the LFSR-CASR PRNG in 4096 ranges

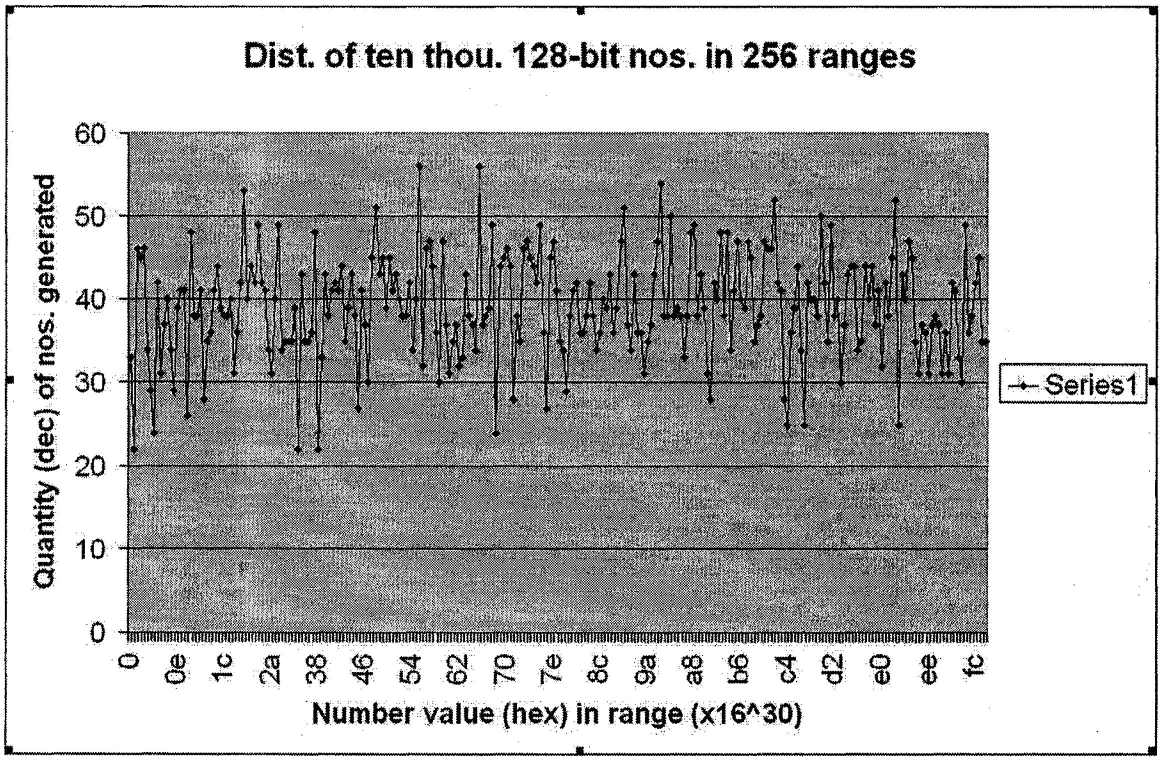


Figure 45. Distribution of ten thou. 128-bit numbers from the LFSR-CASR PRNG in 256 ranges

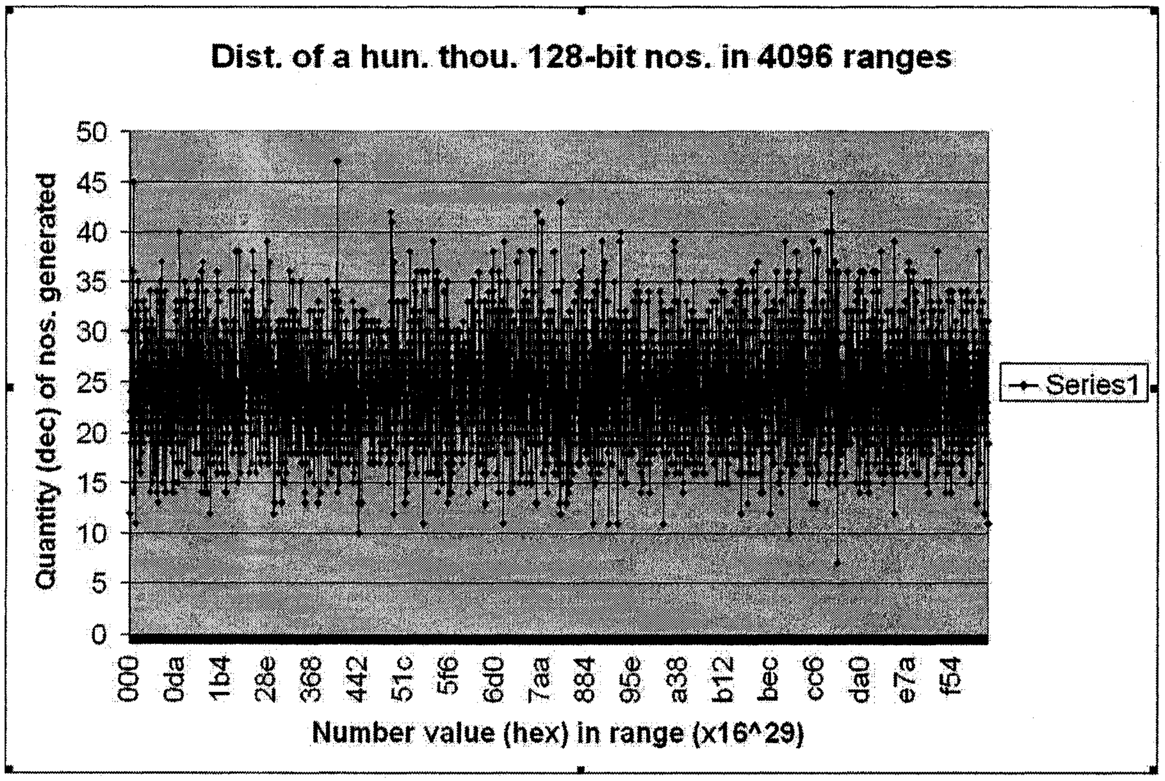


Figure 46. Distribution of a hun. thou. 128-bit numbers from the LFSR-CASR PRNG in 4096 ranges

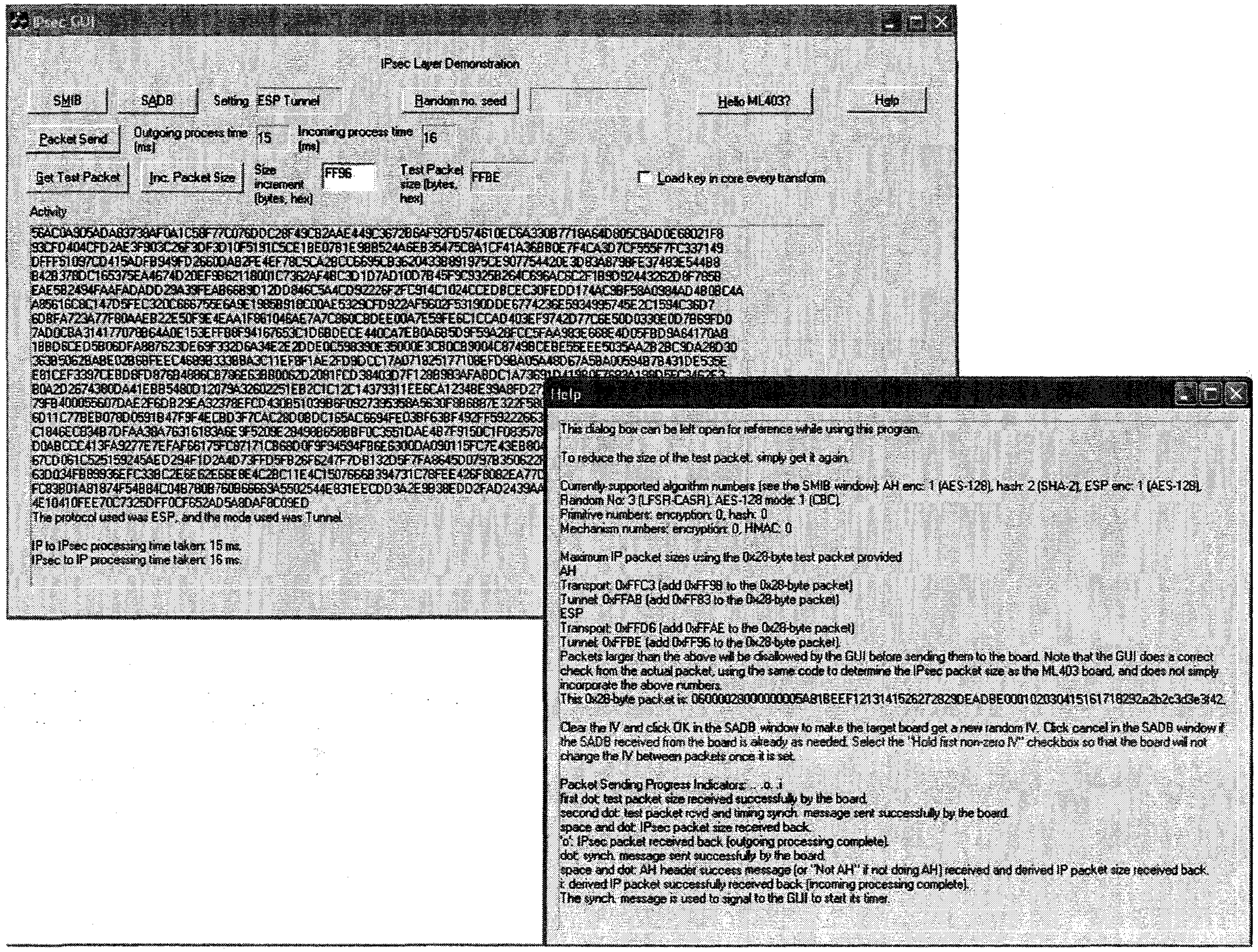
4.3. Performance Results from "IPsecImp", "IPsecGUI" and "IPsecLoop"

Please refer to sections 3.3.10., "Versions", 3.4.1., "Test Methodology" ("IPsecGUI"), and 3.5.1., "Test Methodology" ("IPsecLoop").

In a related note, final memory usage in the XC4VFX12 chip, out of the designed (using BSB) 0 to 0x7FFF BRAM memory space, was 0x7ab6 for "IPsecLoop" and 0x6fea for "IPsecImp". IPsecLoop required more space due to the built-in SADB and test packet, including its copying, changing the test packet size, and the commands to toggle the test direction, change the protocol and mode, setting the test packet size increment, and toggling "NeedNewIV", in spite of removal of the send and receive code for the SADB and SMIB and the removal of the reliable data transfer code.

4.3.1. Demo. of Processing the Largest Possible Packet

Figure 47 shows successful packet processing using the largest IPsec packet possible in ESP, Tunnel mode. An IP packet size of FFBE created by increasing the payload size of the standard 40 (0x28)-byte test packet was the maximum possible using this protocol and mode, since it resulted in an IPsec packet size of 0xFFFF.



Setting: ESP Tunnel

Outgoing process time (ms): 15

Incoming process time (ms): 16

Size increment (bytes, hex): FF96

Test Packet size (bytes, hex): FFBE

☐ Load key in core every transform

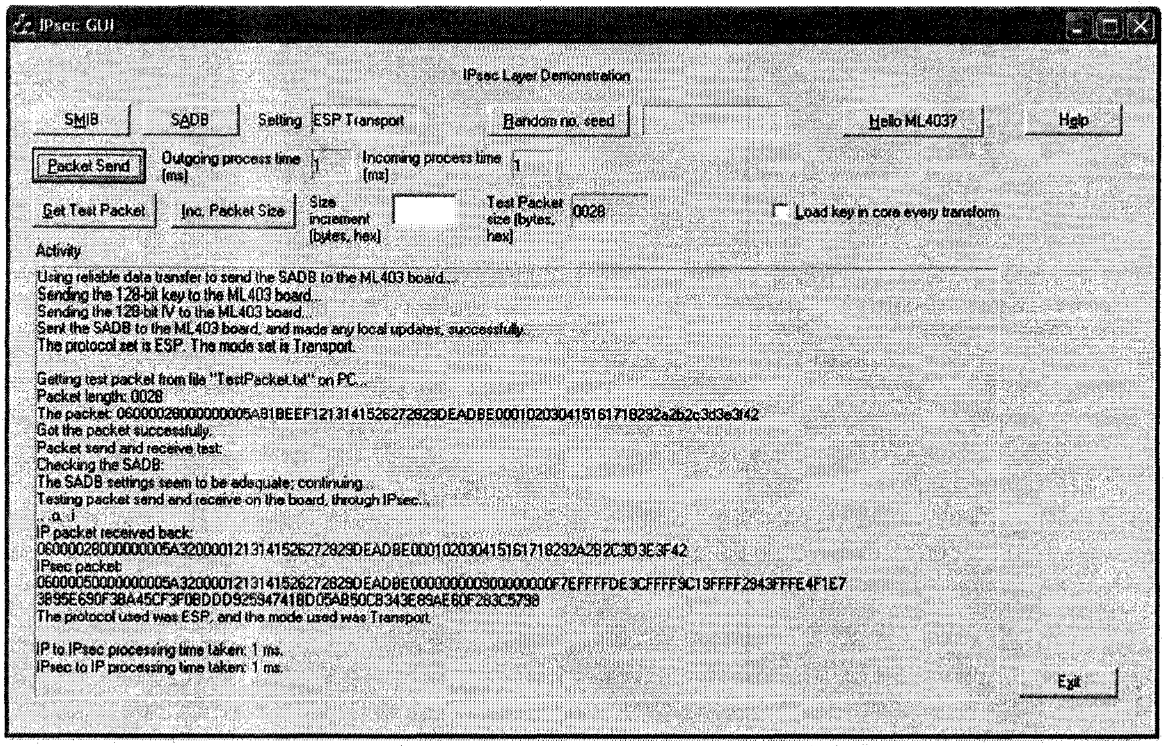
```
56AC0A305ADA83738AF0A1C58F77C076DDC28F49C82AAE449C3672B6AF92FD574610EC6A330B7718A64D805C8AD0E68021F8
93CF0404CFD2AE3F903C26F3DF3D10F5191C5CE1BE0781E98B524A6EB35475C8A1CF41A368B0E7F4CA3D7CF555F7FC337149
DFFF51097CD415ADFB949FD2660DAB2FE4EF78C5CA28CC6695CB3620433B891975CE907754420E3D83A879BFE37483E544B8
B428378DC165375EA4674D20EF9B62118001C7362AF48C3D1D7AD10D7B45F9C9325B264C696AC8C2F1B9D92443262D8F7858
EAE5B2494FAAFADADD29A39FEAB66B9D120D846C5A4CD92226F2FC914C1024CCED8CEC30FEDD174AC3BF58A0984AD4B08C4A
A85616C8C147D5FEC320C666755E6A9E19858918C00AE5329CFD922AF5602F53190DDE6774236E5934995745E2C1594C36D7
6D8FA723A77F80AAEB22E50F9E4EAA1F861046AE7A7C860C8DEE00A7E59FE6C1CCA0403EF9742D77C6E50D0330E0D7B69FD0
7AD0C8A314177078B64A0E153EFFB8F94167653C1D6BDECE440CA7EB0A685D9F59A28FCC5FAA983E668E4D05FBD9A64170A8
18BD6CED5B06DFA887623DE69F332D6A34E2E2DDE0C598390E35000E3C80C89004C87498CEBE55EEE5035AA2B28C9DA28D30
363B5062BABE02B69FEEC4689B3338BA3C11EF8F1AE2FD9DCC17A071825177108EFD98A05A48D67A58A00594B7B431DE535E
E81CEF3397CEBD8FD876B4886C8796E63BB0062D2081FCD38403D7F1298983AFA8DC1A73691D41980E7682A189D5EC346253
80A2D2674380DA41EBB5480D12079A3260225 1EB2C1C12C1437931 1EE6CA12348E39A8FD27
79FB400055607DAE2F6DB29EA32378EFCD430B51039B6F0927395358A5630F9B6887E322F58
6D11C778EB078D0591B47F9F4ECBD3F7CAC28D0BDC165AC6694FE038F638F492FF5922263
C1846EC834B7DFAA38A763161836E9F5209E2849886588BF0C3551DAE487F9150C1F083578
D0ABCCC413FA9277E7EFAF66175FC87171CB68D0F9F94594FB6E6300DA090115FC7E43EB804
67CD061C525159245AED294F1D2A4D73FFD5FB26F6247F7DB132D5F7FA8645D0797B350622
63D034FB89936EFC338C2E6E62E68E8E4C28C11E4C15076668394731C78FEE426F8082EA77C
FC83B01A81874F54B84C04B780B760B66669A5502544E831EECDD3A2E9B38EDD2FAD2439AA
4E10410FEE70C7325DFF0CF652AD5A8DAF8C09ED
The protocol used was ESP, and the mode used was Tunnel.

IP to IPsec processing time taken: 15 ms.
IPsec to IP processing time taken: 16 ms.
```

```
This dialog box can be left open for reference while using this program.

To reduce the size of the test packet, simply get it again.

Currently-supported algorithm numbers (see the SMIB window): AH enc: 1 (AES-128), hash: 2 (SHA-2), ESP enc: 1 (AES-128),
Random No: 3 (LFSR-CASR), AES-128 mode: 1 (CBC).
Primitive numbers: encryption: 0, hash: 0
Mechanism numbers: encryption: 0, HMAC: 0

Maximum IP packet sizes using the 0x28-byte test packet provided
AH
Transport: 0xFFC3 (add 0xFF9B to the 0x28-byte packet)
Tunnel: 0xFFA8 (add 0xFF83 to the 0x28-byte packet)
ESP
Transport: 0xFFD6 (add 0xFFAE to the 0x28-byte packet)
Tunnel: 0xFFBE (add 0xFF96 to the 0x28-byte packet).
Packets larger than the above will be disallowed by the GUI before sending them to the board. Note that the GUI does a correct
check from the actual packet, using the same code to determine the IPsec packet size as the ML403 board, and does not simply
incorporate the above numbers.
This 0x28-byte packet is: 0600002800000005A81BEEF1213141526272829DEADBE0001020304151617182 92a2b2c3d3e3f42.

Clear the IV and click OK in the SADB window to make the target board get a new random IV. Click cancel in the SADB window if
the SADB received from the board is already as needed. Select the "Hold first non-zero IV" checkbox so that the board will not
change the IV between packets once it is set.

Packet Sending Progress Indicators: .. .o. .i
first dot: test packet size received successfully by the board.
second dot: test packet rcvd and timing synch. message sent successfully by the board.
space and dot: IPsec packet size received back.
'o': IPsec packet received back (outgoing processing complete).
dot: synch. message sent successfully by the board.
space and dot: AH header success message (or "Not AH" if not doing AH) received and derived IP packet size received back.
i: derived IP packet successfully received back (incoming processing complete).
The synch. message is used to signal to the GUI to start its timer.
```

Figure 47. Packet processing using the largest IPsec packet possible in ESP, Tunnel mode

4.3.2. Demo. of Correct AES Encryption in "IPsecGUI"

Figures 48 and 49 show correct AES encryption in "IPsecGUI" demonstration, and the SADB settings used. Note the predicted block of ciphertext in the IPsec packet (copied below the GUI window shown in Figure 48). Note also that the IP packet was correctly received back.

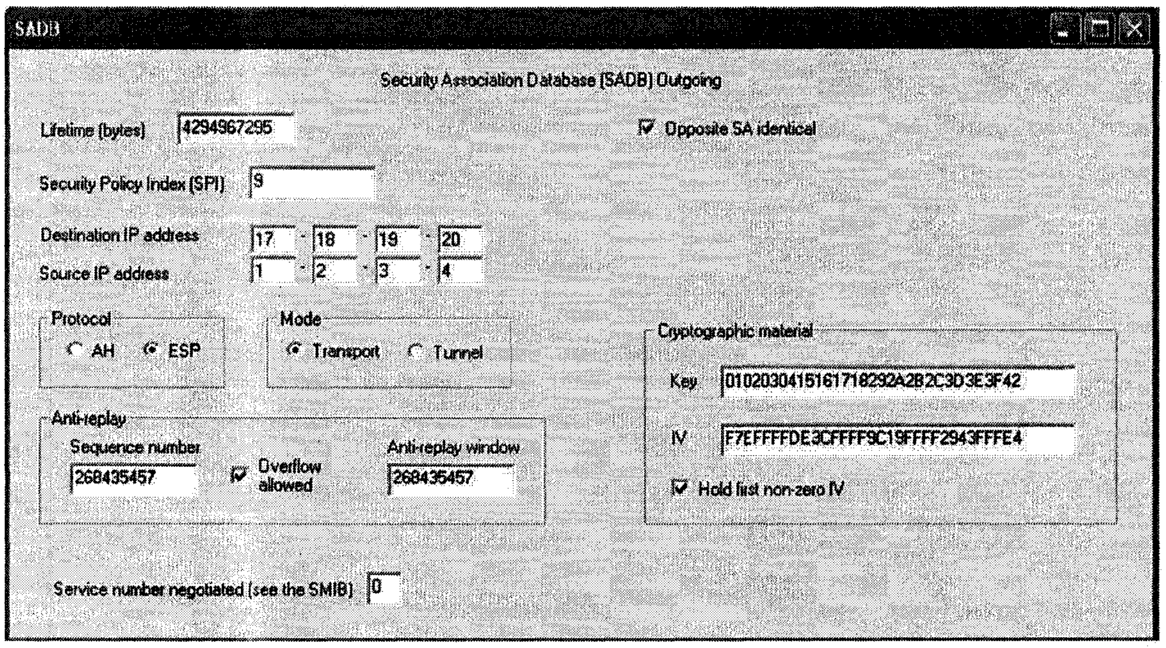


Setting: ESP Transport
Outgoing process time (ms): 1
Incoming process time (ms): 1
Size increment (bytes, hex):
Test Packet size (bytes, hex): 0028

```
Using reliable data transfer to send the SADB to the ML403 board...
Sending the 128-bit key to the ML403 board...
Sending the 128-bit IV to the ML403 board...
Sent the SADB to the ML403 board, and made any local updates, successfully.
The protocol set is ESP. The mode set is Transport.

Getting test packet from file "TestPacket.txt" on PC...
Packet length: 0028
The packet: 06000028000000005A81BEEF1213141526272829DEADBE0001020304151617182 92a2b2c3d3e3f42
Got the packet successfully.
Packet send and receive test:
Checking the SADB:
The SADB settings seem to be adequate; continuing...
Testing packet send and receive on the board, through IPsec...
..a..i
IP packet received back:
06000028000000005A3200001213141526272829DEADBE000102030415161718292A2B2C3D3E3F42
IPsec packet:
06000050000000005A3200001213141526272829DEADBE0000000003000000000F7EFFFFDE3CFFFF9C19FFFF2943FFFE4F1E7
3B95E690F38A45CF3F08DDD92594741BD05AB50CB343E89AE60F283C5798
The protocol used was ESP, and the mode used was Transport.

IP to IPsec processing time taken: 1 ms.
IPsec to IP processing time taken: 1 ms.
```

```
F1E73B95 E690F38A 45CF3F08 DDD92594
```

Figure 48. "IPsecGUI" "screenshot" showing correct encryption



SADB

Security Association Database (SADB) Outgoing

Lifetime (bytes) 4294967295

[✓ Opposite SA identical]

Security Policy Index (SPI) 9

Destination IP address 17 . 18 . 19 . 20

Source IP address 1 . 2 . 3 . 4

Protocol: AH [✓ ESP]

Mode: [✓ Transport] Tunnel

Anti-replay

Sequence number 268435457

[✓ Overflow allowed]

Anti-replay window 268435457

Cryptographic material

Key 0102030415161718292A2B2C3D3E3F42

IV F7EFFFFDE3CFFFF9C19FFFF2943FFFE4

[✓ Hold first non-zero IV]

Service number negotiated (see the SMIB) 0

Figure 49. SADB settings used in the previous figure

4.3.3. “IPsecLoop” Results

In Figure 42, in section 3.5.1., “Test Methodology” (“IPsecLoop”), two encryption pulses are visible, due to the two (128-bit) blocks of hash being signed (encrypted), since the SHA-2 hash (supplied by the function that returns a dummy hash) is a 256-bit hash.

In testing with version 8, it was found that the extra loop-managing overhead after each service call takes 8.4μs, which is not a significant amount of error, but was included in measurements prior to *ver. 8* (see section 3.5., “Design of a CLI Version, ‘IPsecLoop,’ to Facilitate Testing”).

Using “IPsecLoop”, the time required to run the LFSR-CASR PRNG in software for four 32-bit numbers to get the 128-bit IV was determined to be 1.3ms. The time required to load the key into the encryptor was determined to be 15μs, and 20μs when version 8 was tested (see section 3.3.10., “Versions”). The time required to load the key into the encryptor, do key expand in the encryptor and load the key expand values into the decryptor was determined to be 130μs, and 120μs when version 8 was tested. It is not quite clear why the time required to load the key expand values into the decryptor should decrease when the time required to load the key into the encryptor increased, which may have occurred due to the slightly more complex code used to track the key loaded into the core according to the SPI being used.

The figures in sections 4.3.3.1. to 4.3.3.4., inclusive, show the packet processing times graphed, for the range of packet sizes, protocol-mode combinations, and software versions tested (see the test methodology sections: 3.4.1., "IPsecGUI", and 3.5.1., "IPsecLoop"; see also the tabulated data in Appendix B, section B.2.). All lines graphed are closely linear, beginning at the origin.

4.3.3.1. Version 4, with "IPsecLoop" and "IPsecGUI" Comparison

In the graph shown in Figure 50, since it is expected that the line symbols will be difficult to discern without the benefit of colour, it should be explained that the upper line is "Incoming" and the middle line is "Outgoing". The lowest four lines are the "IPsecGUI" results, "GUI in L," "GUI in P," "GUI out L," and "GUI out P," in order from top to bottom, at least on the right-hand end, at a packet size of 16,424 (0x4028) bytes; their total spread there is only about 8 ms.

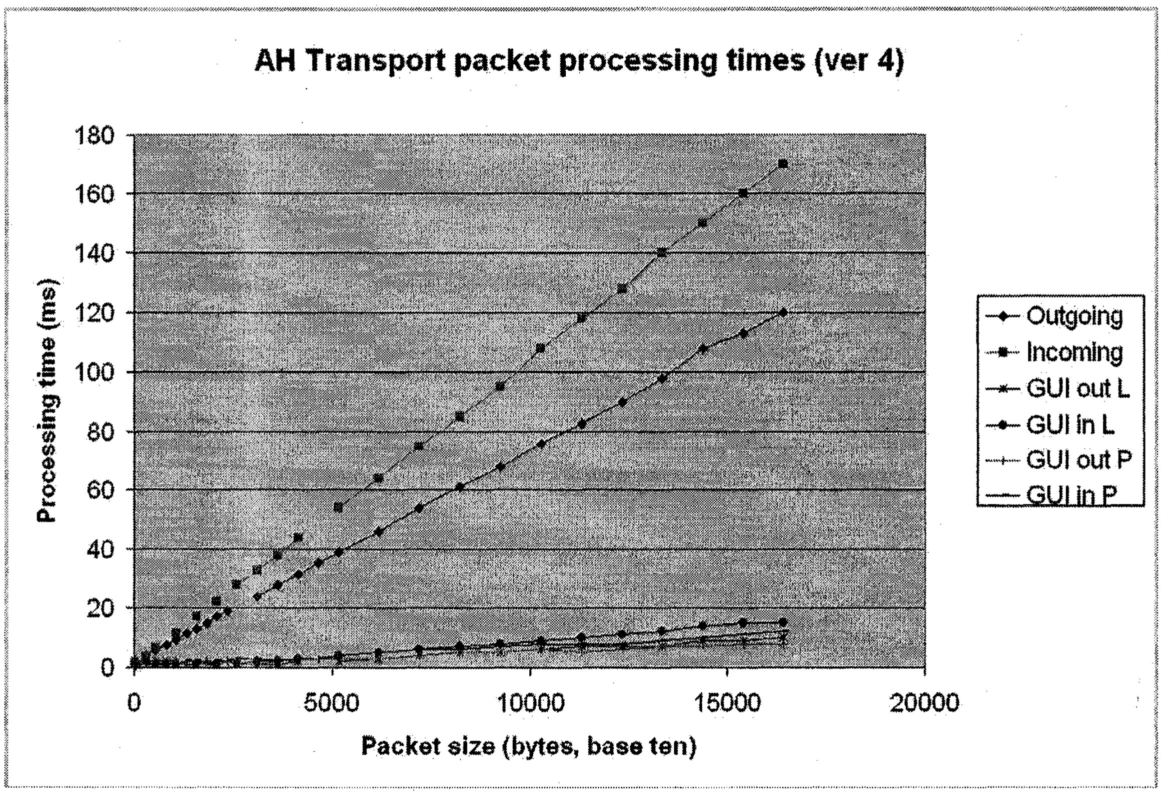


Figure 50. AH Transport packet processing times (ver 4), comparing measurements made using "IPsecLoop" to those made using "IPsecGUI"

Note the low count given by the GUI ("L" stands for the Laptop and "P" stands for the PC). There must have been some activity going on in the "IPsecGUI" code which caused it to miss most of its millisecond time-count event messages. It is concluded that it is a better idea to use good, commercially-available test equipment, than to attempt to build one's own.

The processing times are close to being linear, meaning a fixed per-byte processing rate; the header size of twenty-four bytes which imposes a fixed processing overhead, does not impose an overhead sufficient to appear in these graphs. Differences in processing times in these graphs that are proportional to the packet size must be explained by processing done to each unit of payload.

The incoming (receiving) processing may have taken longer in the results shown in Figure 50, due to two additional comparison operations used in the "for"-loop that copies the datagram without the authentication data in order to hash it with that field cleared. In the outgoing (sending) code (the "if" of the "if-else"), the authentication data section was skipped by using a separate "for"-loop from the one that copied the packet header portion.

4.3.3.2. Version 6 Compared to Ver. 4

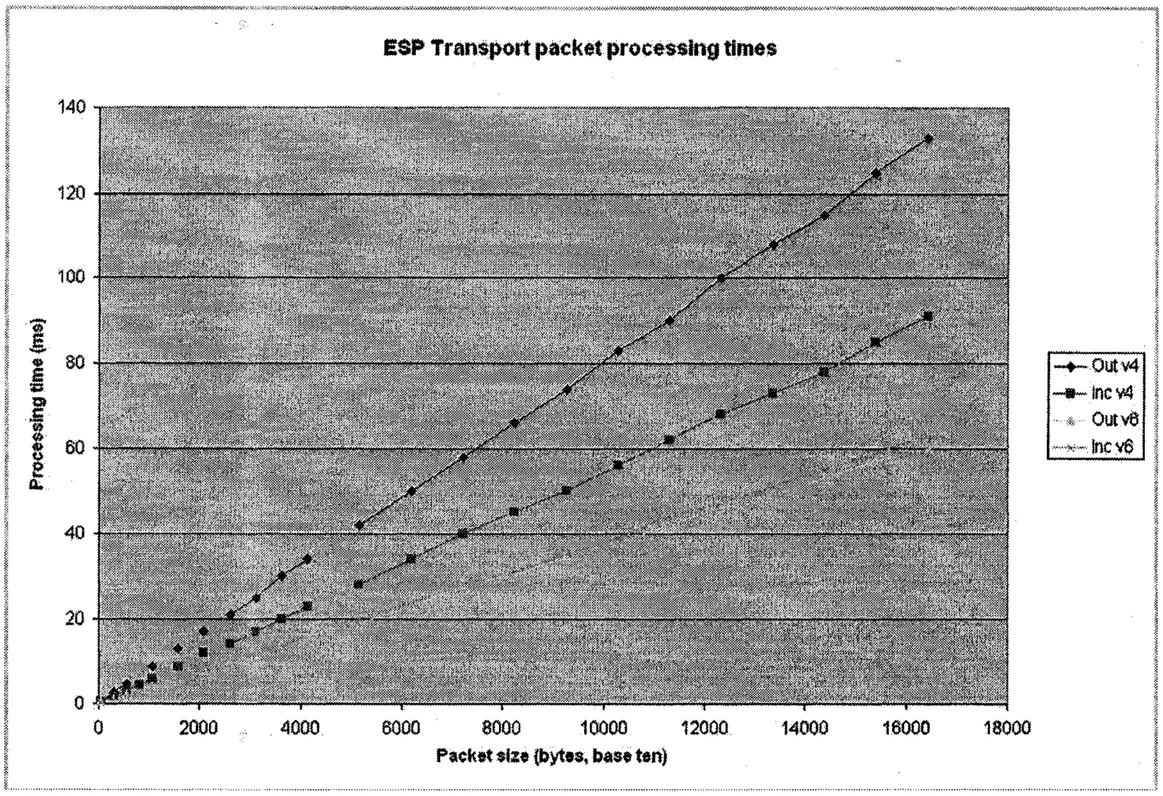


Figure 51. ESP Transport packet processing times – ver. 6 vs. ver. 4

In the graph shown in Figure 51, the lines are "Out v4," "Inc v4," Inc v6," and "Out v6," from top to bottom. "Inc v6" is only above "Out v6" by 3 ms at the right-hand end, and the two are centered on 60ms there.

Note that no change was made to the AH service in *ver. 6*, however the ESP service was modified to remove "for" loops that cleared the entire datagram, which would cause delays proportional to the packet size. Outgoing processing required more time in *ver. 4* in ESP due to copying the entire datagram to be encrypted including the padding as well as to the datagram to be produced, which was not necessary; see also the analogous graph for ESP Tunnel mode, in Figure 52.

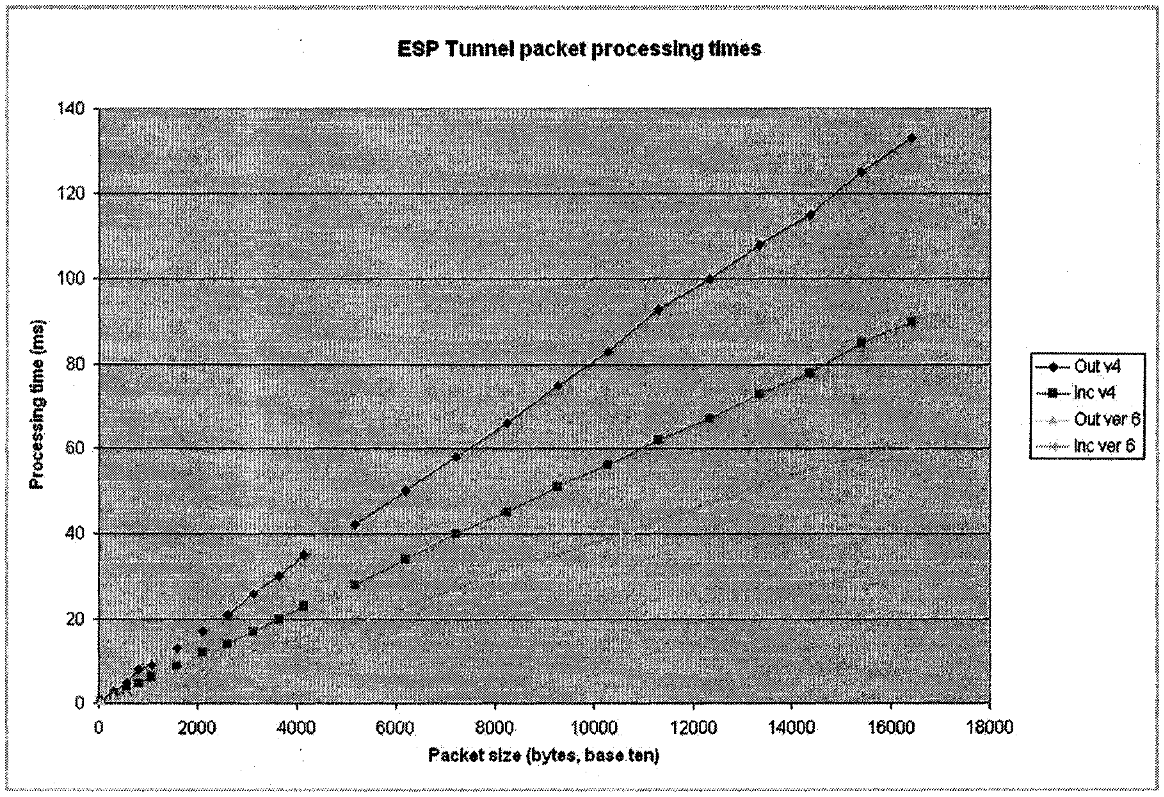


Figure 52. ESP Tunnel packet processing times – ver. 6 vs. ver. 4

In the graph shown in Figure 52, the lines are “Out v4,” “Inc v4,” Inc ver 6,” and “Out ver 6,” from top to bottom. “Inc ver 6” is only above “Out ver 6” by 2 ms at the right-hand end, and the two are centered on 60ms there.

4.3.3.3. Version 7 Compared to Ver. 6 and 4

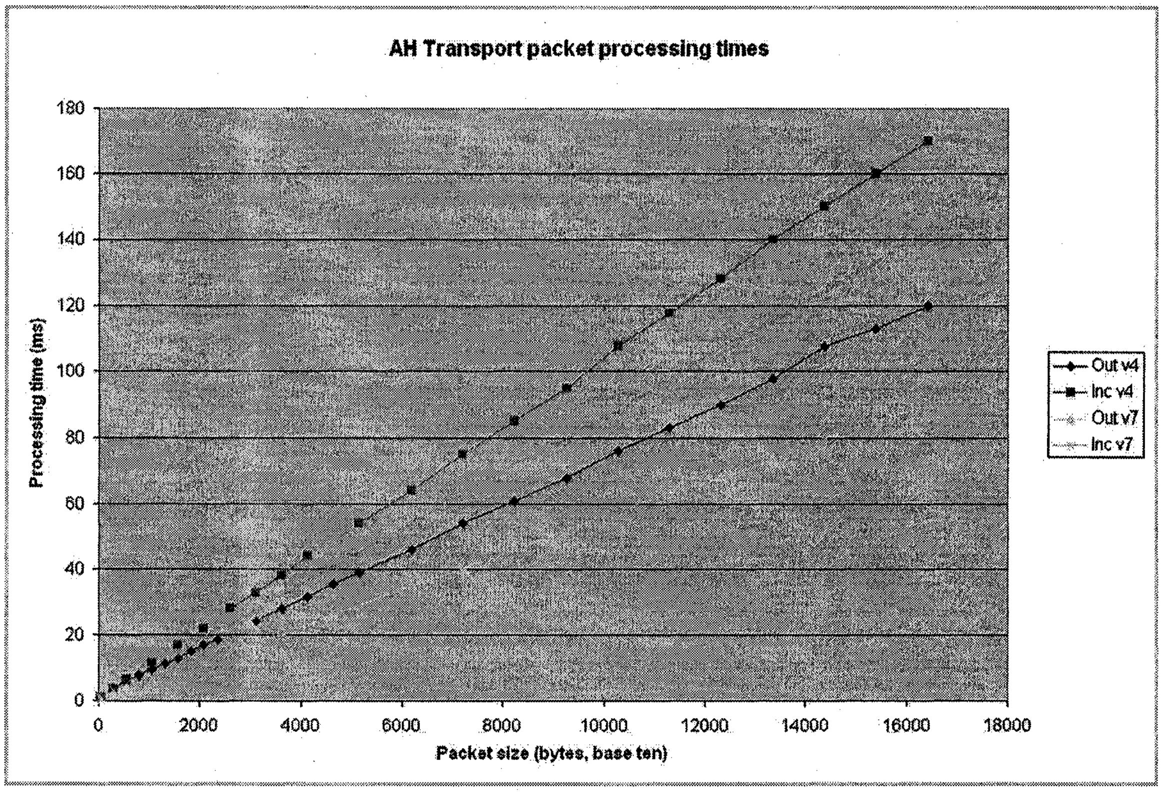


Figure 53. AH Transport packet processing times – ver. 7 vs. ver. 4

In the graph shown in Figure 53, the lines are "Inc v4," "Inc v7," "Out v4," and "Out v7," from top to bottom. "Inc v7" reaches 143 ms at the right-hand end, and "Out v7" reaches 90 ms.

Note that *ver. 4* is compared to *ver. 7* here, since no change was made to AH in *ver. 6.* As noted in section 3.3.10., "Versions," AH was sped up by removing a "for" loop that was being used to pre-clear the entire datagram; instead, only portions of the outgoing or incoming datagram being prepared that needed to be, were cleared. This applied to both modes; see also the graph of AH Tunnel mode in Figure 54.

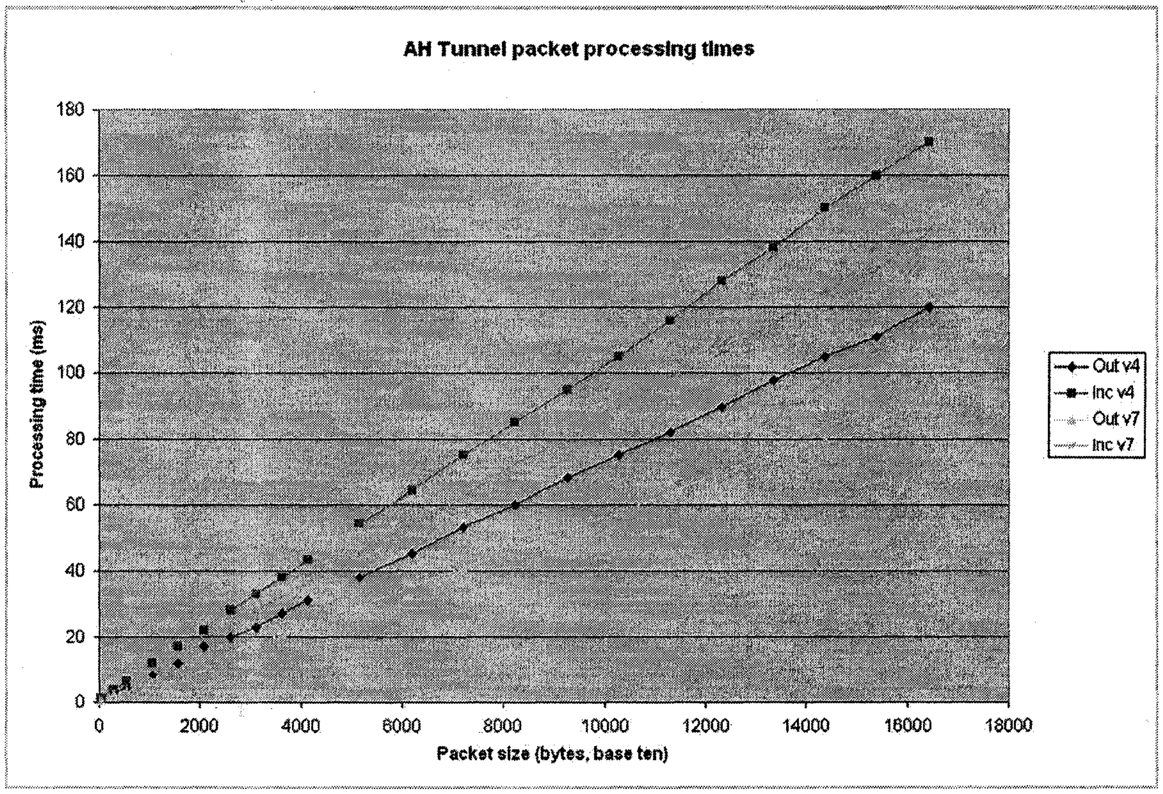


Figure 54. AH Tunnel packet processing times – ver. 7 vs. ver. 4

In the graph shown in Figure 54, the lines are "Inc v4," "Inc v7," "Out v4," and "Out v7," from top to bottom. "Inc v7" reaches 143 ms at the right-hand end, and "Out v7" reaches 93 ms.

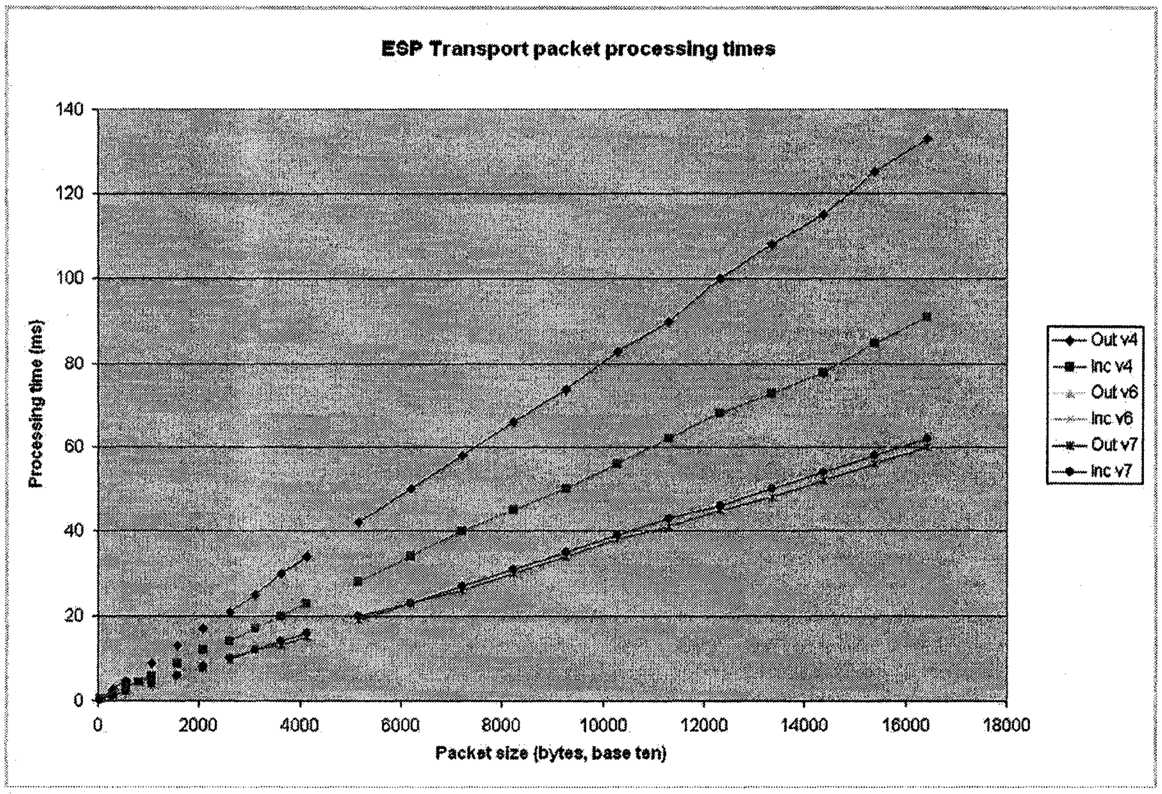


Figure 55. ESP Transport packet processing times – ver. 7 vs. ver. 6 and 4

In the graph shown in Figure 55, the lines are "Out v4," "Inc v4," "Inc v6," "Inc v7," "Out v6," and "Out v7;" those last four looked at, at the right end, since their total spread there is only about 2 ms.

Note that no processing time penalty was incurred in the ESP protocol in going from *ver. 6* to *ver. 7*, showing that processing not done to each unit of the packet paylod is not significant; see also the graph of ESP Tunnel mode in Figure 56.

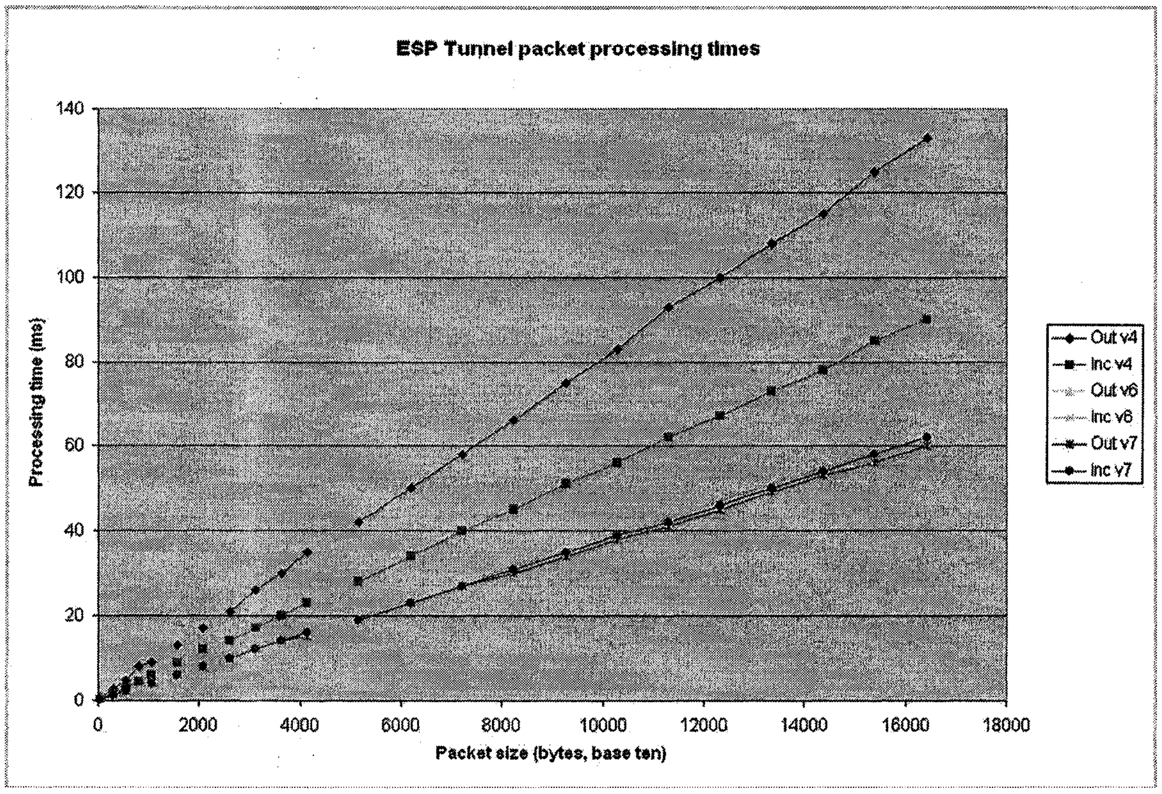


Figure 56. ESP Tunnel packet processing times – ver. 7 vs. ver. 6 and 4

In the graph shown in Figure 56, the lines are "Out v4," "Inc v4," "Inc v6," "Inc v7," "Out v6," and "Out v7," although, at the right end, "Inc v6" is covered by "Inc v7" and "Out v6," the lowest line there, is covered by "Out v7"; the total spread of the last four there is only about 2 ms.

4.3.3.4. Version 8 Compared to Ver. 7

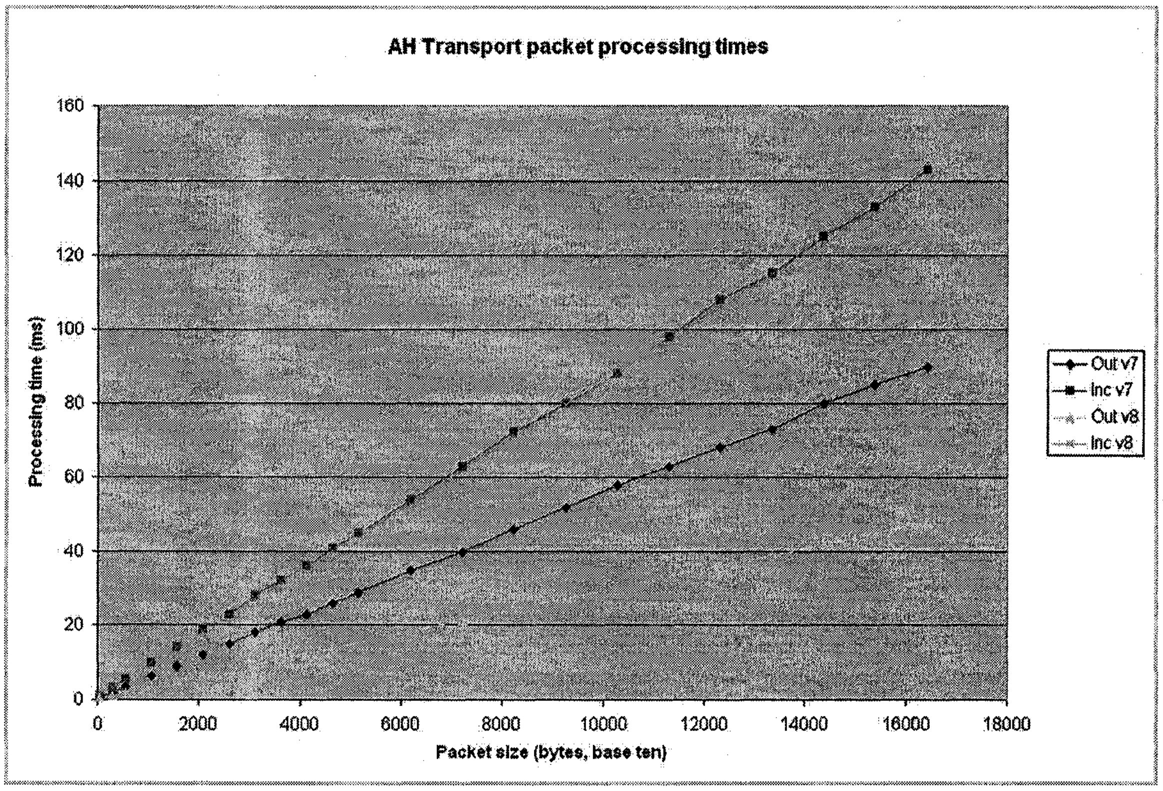


Figure 57. AH Transport packet processing times – ver. 8 vs. ver. 7

In the graph shown in Figure 57, "Inc v7" is very slightly above "Inc v8," at the right end, by only about 1 ms, virtually superimposed, then "Out v7" follows, then "Out v8," from top to bottom. "Out v8" reaches 48 ms at the right-hand end.

The reuse of the datagram prepared for hashing as the outgoing datagram, as noted in section 3.3.10., "Versions," succeeded in almost halving the outgoing packet processing time in the AH protocol in *ver. 8* as compared to *ver. 7*; see also the graph of AH Tunnel mode, in Figure 58. The incoming packet processing time in this protocol could be similarly reduced by clearing the mutable fields of the received IPsec datagram and sending it to the Mechanisms layer for hashing (also removing and saving the block of authentication data), which is advisable, since although the incoming IPsec packets are passed to the Service layer by pointer (i.e., memory address) and thus retain any modifications made to them, they are simply discarded by the security layer above the Service layer, following the service call.

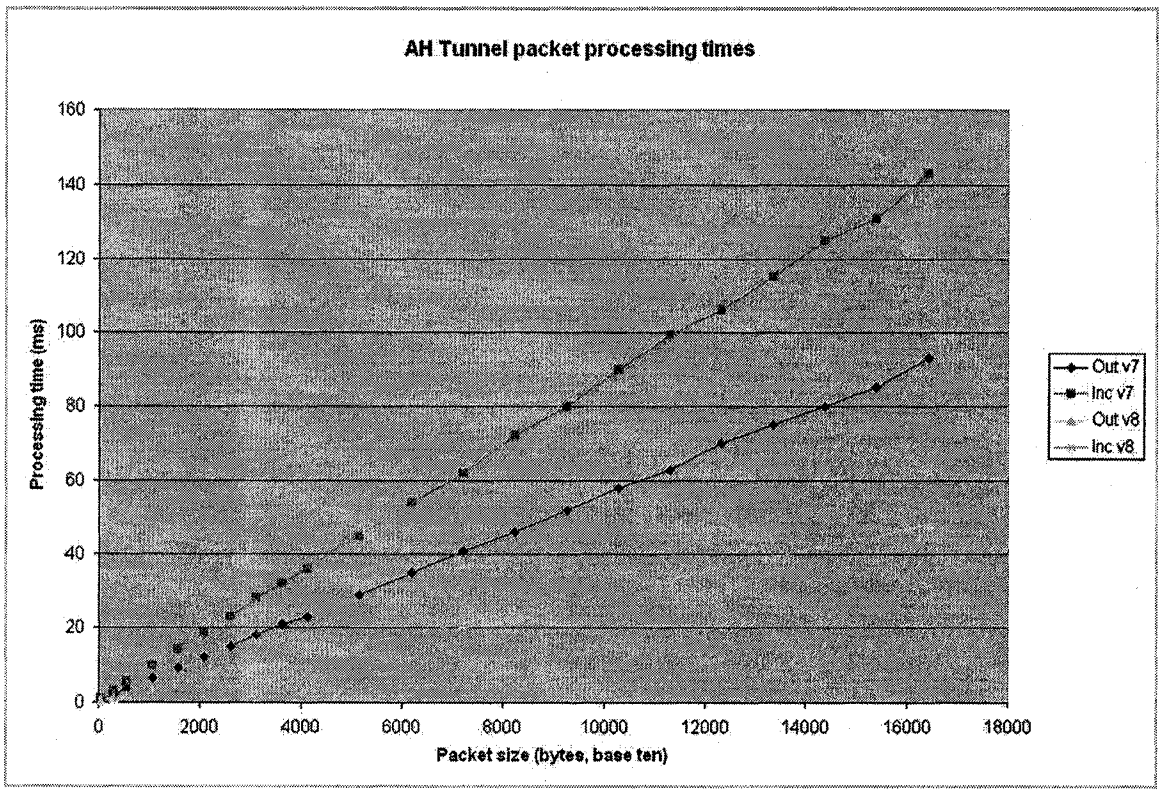


Figure 58. AH Tunnel packet processing times – ver. 8 vs. ver. 7

In the graph shown in Figure 58, "Inc v7" is very slightly above "Inc v8," at the right end, by only about 1 ms, slightly less than in the previous graph, virtually superimposed, then "Out v7" follows, then "Out v8," from top to bottom. "Out v8" reaches 48 ms at the right-hand end.

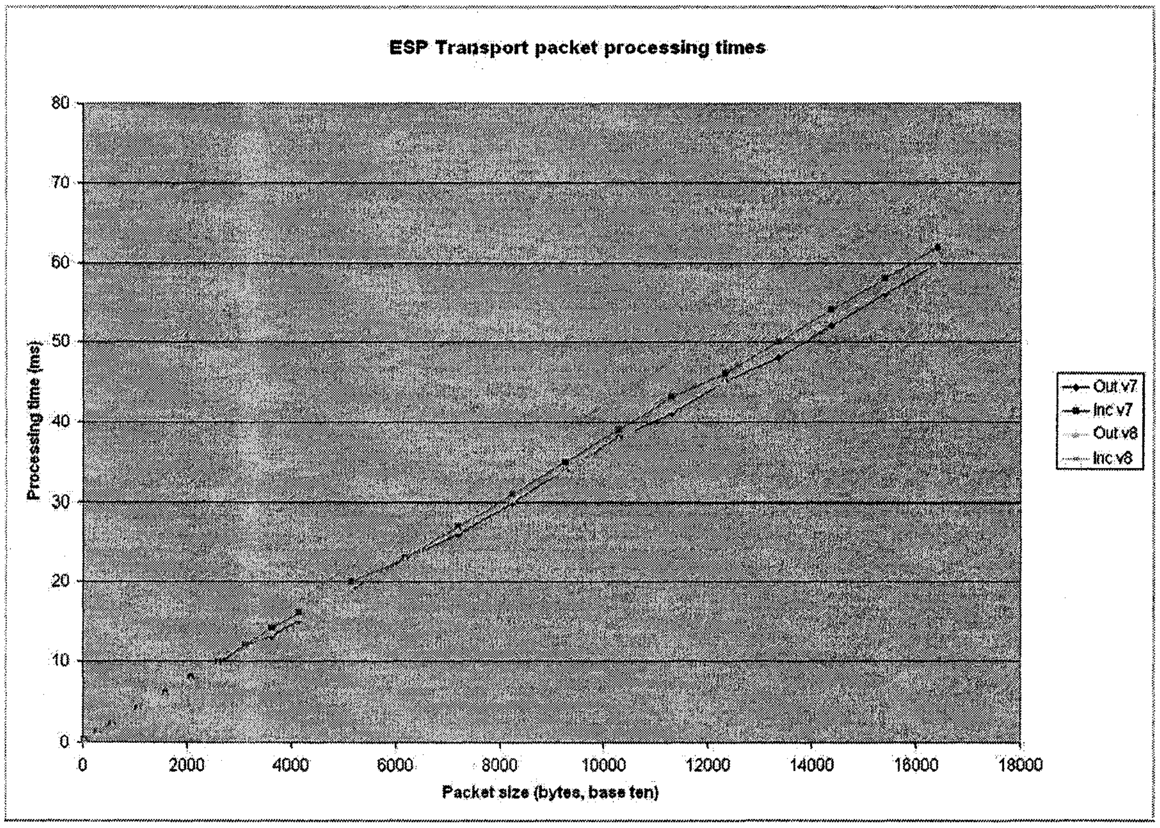


Figure 59. ESP Transport packet processing times – ver. 8 vs. ver. 7

In the graph shown in Figure 59, "Inc v8" is the highest line, reaching 69 ms at the right-hand end, followed by "Inc v7," which reaches 62 ms, and "Out v8" is slightly higher than "Out v7," virtually superimposed.

Here again it can be seen that introducing efficiencies not tied to each unit of payload has no significant effect; modifying for the efficiency of not including the datagram header in the plaintext to be encrypted didn't help at all in the outgoing case and introduced some extra processing in the incoming case. Since the index values of the payload data items alone were no longer the same as those of the entire datagram, an additional index variable was used in the code following the mechanism call, that copies the decrypted data to the IP datagram being built. Incrementing that extra index variable is probably responsible for the extra time taken, since it was the only additional processing added. The outgoing (sending) case was not affected, because it already used an additional loop variable to be incremented, due to the changed positioning of the packet payload in the IPsec packet due to the addition of the ESP header. The receiving case is a little simpler because the payload always goes right after the IP header (and the mechanism processing is not able to place it because the padding size is unknown before and during decryption in the ESP protocol). This is also the case in Tunnel mode, as shown in Figure 60.

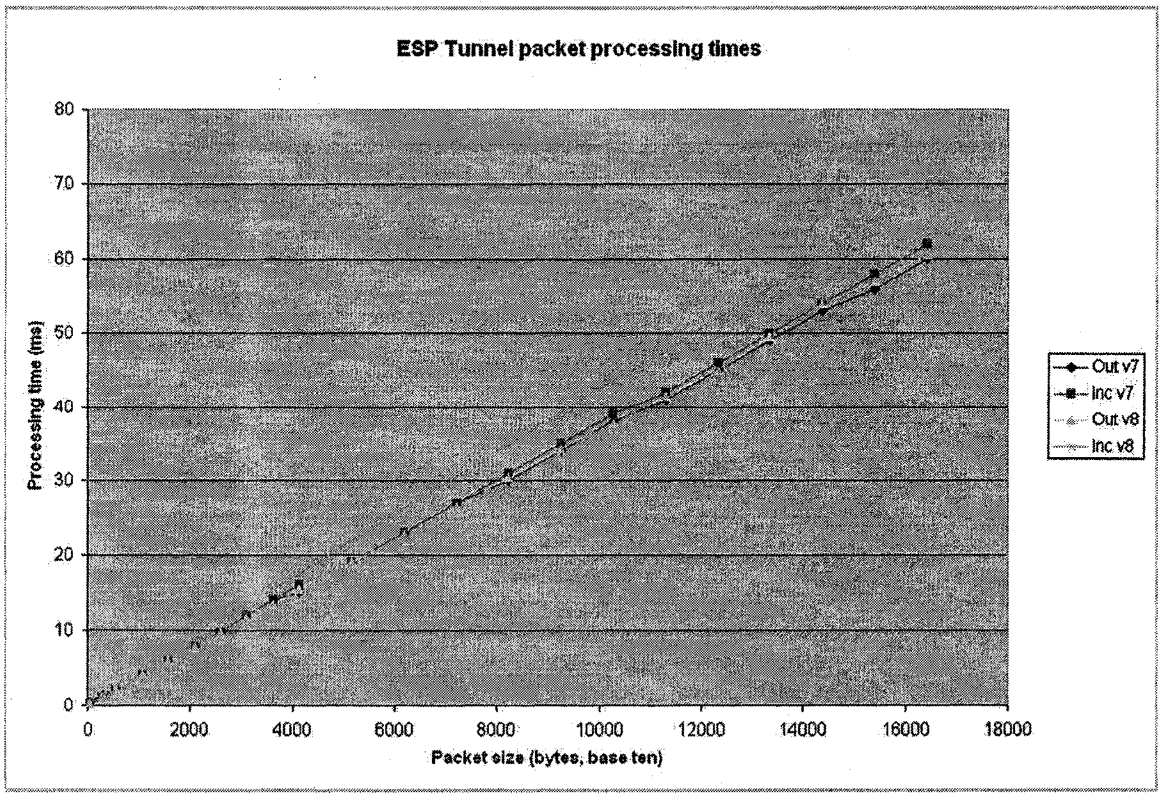


Figure 60. ESP Tunnel packet processing times – ver. 8 vs. ver. 7

In the graph shown in Figure 60, "Inc v8" is the highest line, reaching 69.5 ms at the right-hand end, followed by "Inc v7," which reaches 62 ms, and "Out v8" is slightly higher than "Out v7," virtually superimposed.

4.3.4. Comparisons to Results from the Literature

Table 8 shows a comparison of the best processing times achieved in this work against an available report of processing times found in the literature [KER1997]. The paper reported on a 3DES implementation, which is notoriously slow, all done in software, which is not difficult to beat. The paper's times are for combined ESP with

ESP's own authentication, and are the latencies. At the largest packet size compared, the sum of the largest AH and ESP processing reported in this work (which would be two different protocols, nested), become slightly larger than the 3DES-MD5 implementation. For example, at the 8kB packet size (approx. 2028 hex.), if the incoming AH Transport and ESP Transport mode from this work were to be used together, the latency would be 103 ms vs. 100 ms reported from the 3DES-MD5 implementation, although the AH incoming processing time in this work can be halved, as explained before. However, at the small packet sizes, the worst combined times from this work become approximately five times less than those of the 3DES-MD5 implementation. This work could be added to, to add the ESP's own authentication service, which would be more efficient.

IPsec packet processing times comparison										
Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (AR)	Incoming proc. time (ms) (AR)	Outgoing proc. time (ms) (AU)	Incoming proc. time (ms) (AU)	Outgoing proc. time (ms) (ER)	Incoming proc. time (ms) (ER)	Outgoing proc. time (ms) (EU)	Incoming proc. time (ms) (EU)	[KER1997] 3DES-MD5 (ms)
28	40	0.435	0.84	0.51	0.975	0.38	0.315	0.44	0.255	
128	296	1.19	3.1	1.25	3.2	1.3	1.3	1.4	1.2	
228	552	1.94	5.3	2	5.4	2.2	2.2	2.3	2.2	40
328	808									
428	1064	3.43	9.6	3.5	9.8	4.1	4.1	4.2	4.1	42
528	1320									
628	1576	4.9	14	4.95	14	5.9	6	6	6	
728	1832									
828	2088	6.4	19	6.45	19	7.8	8	7.8	7.9	
928	2344									
a28	2600	7.9	23	7.95	23	9.6	9.9	9.7	9.8	
c28	3112	9.25	28	9.38	28	12	12	12	12	
e28	3624	10.9	32	10.9	32	13	14	14	14	
1028	4136	12.4	36	12.8	36	15	16	15	16	65
1228	4648	13.8	41	13.9						
1428	5160	15.3	45	15.4	45	19	20	19	19	
1828	6184	18.3	54	18.3	54	23	23	23	23	
1c28	7208	21.3	63	21.3	62	26	27	27	27	
2028	8232	24.3	72	24.3	72	30	31	30	31	100
2428	9256	27.3	80	27.3	80	34	35	34	35	
2828	10280	30.3	88	30.3	90	38	39	38	39	
2c28	11304	33.3	98	33.3	99	41	43	41	42	
3028	12328	36.3	108	36.3	106	45	46	45	46	
3428	13352	39	115	39.3	115	48	50	49	50	
3828	14376	42	125	42.3	125	52	54	53	54	
3c28	15400	45	133	45	131	56	58	56	58	
4028	16424	48	143	48	143	60	62	60	62	

(Packet sizes in bytes - 8 bits in the ML403 board)
This work: IPsecLoop ver. 7, March 21, 2007 AR: Auth. Trans.; AU: Auth Tunn.; ER: ESP Trans.; EU: ESP Tunn.
Using IPsecLoop ver. 8 results, April 9, 2007, for AH outgoing, both modes: AR and AU
SW, 100MHz PPC405 with FPGA AES accelerator
(A: Authentication Header; E: ESP: Encapsulating Security Payload; Trans.: Transport Mode; Tunn.: Tunnel Mode.)
[KER1997]: Keromytis, 1997: latency - a SW, Linux, implementation on a 166MHz Pentium w. 100Mbps Ethernet

Table 8. Packet processing times compared to a result from the literature [KER1997].

In Table 9, shown as a group of tables, the processing-time results from the previous table are expressed as rates, calculated by dividing the packet size processed by the time taken, averaging them for each category and expressing them in units of Megabits per second (Mbps). A comparison with three results found in the literature is made.

IPsec packet processing rates (Mbps)							
Outgoing proc. rate (AR)	Incoming proc. rate (AR)	Outgoing proc. rate (AU)	Incoming proc. rate (AU)	Outgoing proc. rate (ER)	Incoming proc. rate (ER)	Outgoing proc. rate (EU)	(cont) Incoming proc. rate (EU)
2.5	0.88	2.5	0.87	2.1	2.0	2.1	2.1

These results are calculated averages across 40 to 16kB packet sizes (since the rate remained roughly constant due to linearity of the processing-time results)

Literature reports (Mbps)		
[BEL2002] AES	[DAN2000] AES	[CHE2002] 3DES
50	353	53

[BEL2002]: SW, Linux, w AES acceleration on Xilinx Virtex 1000 FPGAs
[DAN2000]: all on an FPGA (make, model unspecified)
[MCL2002]: Did not test packet processing rates
[KIM2004]: Did not test packet processing rates
[CHE2002]: Free S/WAN using a DES accelerator on a custom platform, "Pilchard"
[LUJ2005]: Did not test packet processing rates

Table 9. (group): Packet processing rates and comparison with the literature

The processing rates reported from this work are lower than those found in the literature due to reading core results from registers via the bus rather than using DMA, FSL, an all-FPGA implementation or some other such fast access method.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

In this work, an IPsec implementation, including five design layers, from top to bottom: Policy, Management, Service, Mechanism, and Primitive, an SMIB (Security Management Information Database), an SPDB (Security Policy Database), an SADB (Security Association Database), code to support testing and a GUI, was developed and tested. The portions designed and the results are discussed as follows.

5.1. The AES Implementation

A 128-bit AES implementation was done in VHDL in this work, working from a published Verilog core design that accurately implemented the published AES standard, using a Xilinx Virtex-4 FPGA, part number XC4VFX12, in the Xilinx ML403 development board. The cipher and inverse cipher were implemented as separate modules, an encryptor and decryptor, using 99% of the FPGA fabric, and unrelated logic occupying 15% of the slices; even so, the key expand module was not added to the decryptor and a protocol is provided to obtain the key expand, or permutation, values from the decryptor and store them in the decryptor, a process that requires 120 μs, whereas loading the key to the encryptor required 20 μs. The speed grade of the FPGA was 10, the slowest, and the clock used in the modules was divided by two from the 100 MHz board clock. The encryptor performs its transform in eleven clock cycles, i.e., 220 ns; the decryptor in ten, i.e., 200 ns, due to a better timing plan, which could be adopted for the encryptor. These times imply processing rates of 582 Mbps and 640 Mbps, respectively. However, the access method to obtain the results is only via software via 32-bit registers via the OPB, which reduces maximum processing rates to about 22 to 27 Mbps in this implementation.

In addition to the ability to generate and read the key expand values from the encryptor and store them in the decryptor via software, the key expand values stored in the decryptor can be read via software. Both peripherals have CBC mode built in, so that the plaintext input to the encryptor is always XORed with the stored and automatically-updated "IV" register, and the internally decrypted block in the decryptor is always XORed with its similarly-designed "IV" register. The "IV" register can be loaded into each peripheral, and a readback method is provided in the decryptor. The peripherals each automatically perform their transform when their input data registers are written. A "programming model" is provided in this work showing the bit positions in the peripheral slave registers for the commands to accomplish these operations and giving the operational protocol required. Connections to pins on the ML403 board provide two signals, each of which is high when its corresponding peripheral is performing its transform, allowing oscilloscope measurements to be taken for research, development, and evaluation purposes.

A version of the decryptor was produced that can output the value of each step of each round, and did so in testing. A bug fix was added that was verified in the main version of the decryptor, to the source code of the "stepper" and is expected to induce the "stepper" version to begin always with the correct output.

Numerous small software programs were written in C and C++ to test and demonstrate the AES cores, culminating in a full CLI version that demonstrates all features of the cores in individual block processing, and a C++ GUI that runs on a PC and uses the ML403 board to encrypt and decrypt an ASCII text file on the PC. It was found that the IV should not be encrypted when CBC mode is used, since an attacker could use knowledge of that design to tell when the first block of plaintext happens to be all zeroes. The processing rate of the "AESfile" GUI using the ML403 board in this way was found to be in the 12-17 kbps range, given a 57,600 bps RS232 link and a good deal of time-consuming processing display echoing to the user in the GUI.

It is hoped that this section of this work will contribute a commercially-useful AES implementation, with the features necessary for it to be used in practice, and contribute knowledge of the capabilities of the relatively recently-developed Xilinx Virtex 4 FPGA, results from which have not yet been seen in journal and conference papers.

5.2. The LFSR-CASR PRNG Implementation

A LFSR-CASR PRNG implementation was done in C++ and C in this work, working from a published Verilog design. Distributions of samples of its output were graphed in this work, showing that its output seems to have a good, uniform distribution. Using the IPsec implementation done in this work, it produces and stores a 128-bit random number in 1.3ms, as a concatenation of four of its output words (a rate of 128/1.3 = 98 kbps). According to its original specifications, the basic design has a cycle length of 2^{80}-2^{43}-2^{37}+1 and a bias of 2^{-80}. Its final XOR and the use of only the lower 32 bits of each of its state variables conceals the states from cryptanalysis. The original design came supplied with a report from the literature that showed that it did well in the "Diehard" series of statistical tests [TKA2002].

However, AES itself makes a better PRNG, having a cycle length of two to the exponent of its block size, and being carefully and successfully designed for extreme non-linearity. From a design comparison, it is estimated that AES contains up to two orders of magnitude more non-linearity than does this LFSR-CASR PRNG. Since it is already implemented in the FPGA, it can produce random numbers much more quickly, at a rate of 27 Mbps in this work. Although an FPGA implementation of the LFSR-CASR PRNG would probably generate pseudo-random numbers faster than AES, it remains merely a good PRNG, while AES makes a great one [HEL2003].

5.3. The IPsec Implementation

Working from the five-layer framework for design of a security system established in previous work, a partial IPsec implementation was designed, written in C and tested. Content from each of the five layers was included. The AES and PRNG implementations discussed before were used as the primitives, and a dummy SHA-2 hash routine was included. Two mechanisms, an HMAC and an encryption mechanism, and two service routines, AH and ESP, were implemented. IPv4 was supported. The HMAC mechanism performs its signing by using the AES symmetric key to encrypt the hash – a key exchange service was not implemented. The ciphertext in both the HMAC and the transformed ESP datagram begin with the IV itself, sent unencrypted along with the datagram. The mechanisms are designed to show the technique of selecting the primitives and to show the technique of selecting different mechanisms; new mechanisms added are intended to be added as additional cases to the existing mechanism files. The anti-replay service was not implemented. Attention was given to possible security leaks, such as by making sure that the padding area in the ESP plaintext is cleared. The SMIB is designed to be used to specify and select the mechanism from the service and the primitives from the mechanism. The idea of having an array of services, mechanisms and primitives is indicated. The SMIB is designed in its policy layer to support doubly-nested SAs.

At the management layer, two versions were implemented, one which supports a remote GUI, running on a PC, and a CLI (Command-Line Interface) version more suitable for use in performing laboratory measurements. In the GUI version, the IP addresses of multiple clients can be entered and the Service, Mechanism and Primitive layers are configurable. The board itself supplies the basic configuration of its implemented capabilities. In the SADB, the IV can be set constant, or to be regenerated by the RNG for every packet sent. The PRNG can be seeded from a free-running counter. The GUI demonstrates packet transform and reverse transform: IP to IPsec and back.

At the policy layer, multiple policies were implemented, making the GUI running on the PC a policy server. Only the selected policy is downloaded to the board. It is intended that the service number as an index of the array of those available be negotiated in SA setup; the SADB is designed to hold that selection. The SADB and SMIB can be saved to files on the PC from the GUI, the SMIB in three different files, one for each of the policies, the clients' IP addresses, and the base SMIB.

The implementation was verified to successfully process packets from the smallest to largest possible size of a five-32-bit-word header and one-byte payload, to a six-32-bit-word header and a total size of 65,535 bytes, respectively, in both the GUI and the CLI versions. A benefit of layering and modularity is that all files at the layers below the management were the same in both versions.

The typical packet processing rate achieved in this implementation was found to be 2.0 Mbps, measuring the rate at which the Service calls could be repeated, although outgoing (IP to IPsec) AH processing reached 2.5 Mbps and incoming AH processing sank to 0.87 Mbps. The packet processing times for all four protocol-mode combinations were always linear, meaning a roughly constant packet processing rate. Operations that do not need to be done over the packet payload do not significantly affect processing time. The time required, 120 μs, to load the key expand values into the decryptor, is not a serious delay, considering that it usually only has to be done once after the SA is set up with a particular key. It could cause more delay if the board serves more than one incoming SA. The time required to get a random number using the software implementation of the LFSR-CASR PRNG, 1.3 ms, is significant, especially given the processing times of the most common packet size of about 500 bytes, of about 2 ms. Processing rates reported in this work are lower than those reported in [BEL2002], [DAN2000] and [CHE2002] (see section 4.3.4., "Comparisons to Results from the Literature"), due to reading core results from registers via the bus rather than using some fast access method such as DMA, FSL (Fast Simplex Link) or an all-FPGA implementation. However, it is not difficult to beat a software implementation of 3DES.

Some different versions of the service calls and lower were produced in development, to carefully adhere to the design concept of using the SMIB to select the modules used in the lower layers from those in the upper layers, and to remove inefficiencies and speed up processing. Four different versions were involved in testing.

As briefly noted before in this section, a huge benefit resulted from the modularity of layering when a second version, "IPsecLoop", was needed for testing: only the management layer file needed to be modified; eight code files and their "header" files for three lower layers could simply be reused. A drawback of layering was noticed in which data present at a lower layer had to be read at a higher layer, that is, the Primitive layer information on the size of data created at the Primitive layer had to be used at the Services layer in order to determine packet sizes; strict separation of layers had to be discarded in order to have a successful implementation. However, the design of the management layer is still not entirely rigorous.

5.4. Recommendations for Future Work

5.4.1. The AES Implementation

The timing plan of the encryptor could be revised to match that of the decryptor, "shaving" a clock cycle off of the time required.

A higher speed grade of the XC4VFX12 Virtex-4 could be used to determine if the core transform rate could be doubled. This could be attempted in simulation with available tools, without having to buy an actual chip; once post-PAR simulation succeeds, operation in the actual chip is virtually guaranteed. However, one must pay attention to test techniques; it is easy to set unrealistically-short input pulse times in the testbench, for example, when real-world input pulse durations may trigger unintended effects in the design and cause it to fail.

The debugged source code of the "stepper" version of the decryptor could be built, tested and verified.

The preceeding three ideas would be good initial exercises to perform to become familiar with the technology. An ambitious investigation would be to see if the encryptor and decryptor could be implemented in the same module.

Programming the Flash EEPROM could be done so that the code in the board would be non-volatile, making demonstration easier, especially with larger code sizes that require the 1MB of SRAM, since the debugger to load the executable code separately may not be available when the board is programmed "in the field" or in demonstrations, using Xilinx “iMPACT”, due to the full Xilinx software package being cumbersomely large for installation on a laptop PC.

5.4.2. The LFSR-CASR PRNG Implementation

AES should be used as the PRNG in this work, saving the code space required for the software implementation of the LFSR-CASR PRNG that has a far shorter cycle length and is probably much less non-linear, and speeding up random-number generation by over two orders of magnitude using technology established in this work.

5.4.3. The IPsec Implementation

As with most work, the amount of work that can be done in the future vastly exceeds the amount of work completed.

The project could be redone without the layering in order to compare the effectiveness of the layering technique, but that would be tedious, since it is not expected to produce useful, modifiable code and is therefore not recommended. Instead, it is recommended to investigate the use of DMA to speed up processing; then the results obtained could be expected to be improved, and could be recompared with those from the literature. An all-FPGA implementation would be an ambitious undertaking, and would require the purchase of considerable amounts of new hardware. FSL could perhaps be attempted using the Xilinx “Microblaze” boards available in this department, if available in the version installed, which is 6.1. Otherwise that might require purchase of new hardware, design software and Xilinx IP cores.

As a conclusion from this work and a recommendation, it is a better idea to use good, commercially-available test equipment than to attempt building one’s own.

An easy and forseen improvement that can be made is to double the speed of the AH incoming processing using the method described (see section 4.3.3.4., “Version 8 Compared to Ver. 7”). The ESP Service routine should be reverted from *ver. 8* to *ver. 7.*

As was done in this portion of this work, the IV should not be encrypted if included with ciphertext encrypted in CBC mode, since an attacker could use knowledge of such a feature to determine whether the first block of plaintext is all zeroes.

Additional primitives, mechanisms, and thus services could be added, implementing the arrays in the SMIB by making the contents of the constituent lower three layers into pointers to the contained structures. The reading of the SMIB to determine the services, mechanisms and primitives used would then have to be fully "fleshed out". The "Enabled" fields that were designed into the SMIB Services, Mechanisms and Primitives layers, could be actually used. The SHA-2 hash routine, which contains only dummy code as a result of this work, could be implemented. The unimplemented IPsec services and portions thereof, such as anti-replay and ESP authentication (which would be more efficient than nesting ESP and AH SAs – see section 4.3.4., "Comparisons to Results from the Literature"), could be added. Support for IPv6 could be added. A challenging investigation would be to make the design of the management layer more rigorous.

Header checksumming support could be added, as well as support for other IP header and AH header fields. This work now supports the IP header total length field (in octets), the datagram total length field, the AH or ESP protocol field, the TTL (Time To Live) field, the AH header Payload length, the ESP padding length, and the AH and ESP headers' SPI. All other fields were left for future work.

An *RxIVConstant* variable can be defined for the SADB and used if the particular mechanism calls for the IV not to be included with the ciphertext. More criteria could be added as policy selectors. More ideas could be incorporated into the SMIB from [KEN1994].

A function should be used for the AES-128 encryption mechanism, since a similar block of code is used in both the HMAC and Encryption mechanisms. In general, functions should be used for any mechanism component that may be added in the future, that is used in a similar way in more than one mechanism.

More sophisticated error recovery techniques could be added, such as "freeing" all "malloc"ed pointers in the event of heap overflow.

Since the reliable data transfer routines sometimes "lock up" and seem, during the initial transfer of a session, particularly, to take many repetitions for a successful transmission, their operation should be debugged, using a protocol analyzer. Display echoing to the activity was used in debugging, which might continue to be useful; the echoing commands affect the timing, which is "touchy", and are left "commented out", in the source code.

As with all complex software, the software written in this work should be viewed with scepticism and should be continually tested, particularly in areas of operation that do not receive common use, or in areas of unintended operation permitted by the software that could cause the program to fail.

Key Exchange could be added, as well as support for bidirectional SAs; in this work, the incoming SA is set to use the same SA as the outgoing. To implement multiple SAs, the SADB declaration could be made a pointer to an array of pointers that each point to an SADB. Then space for SADBs can be "malloc"ed at will, as can the space for an array of pointers as the number of SADB instances change. To work with the GUI, an intermediate dialog could be added in the GUI that just gets the different SPIs from the board, allowing the user to select the SPI of the SADB that he wishes to view. It does not seem likely that support for the incoming SADB being identical to the outgoing would need to be retained.

In outgoing processing in the IPsec sublayer, the SA should be chosen from the future array of SAs based on the selectors in the datagram. In incoming processing, the SAs should all be searched for the SPI that matches that in the datagram.

A challenge would be to make the implementation in this work communicate with a different existing implementation. A full OSI stack would have to be included. The physical OSI layer could be Ethernet and use the RJ11 connector on the ML403 board. The application layer can be the interface to the GUI on the PC or perhaps the LCD on the ML403 board, or any other of the many IO devices and output ports on the ML403 board.

If AH IPsec packets fail to authenticate, that should at least be made an auditable event, for the system administrator to check on, and should conceivably be passed up to the application layer to prevent further communication and inform the user of that.

APPENDICES

APPENDIX A

A. Pseudo-Code

Note that "//" or "/*" indicates a comment

A.1. IPsecImp

A.1.1. The SMIB

A.1.1.1. The Overview and Top Two Layers

```
structure SMIB {
   structure PolicyLayer {
      NumberOfPolicies
      pointer to array of policy structures }
   structure ManagementLayer {
            Local IP Address
            NumberOfClients
            boolean AreAddressRanges   // If so, there should be an even
NumberOfClients
            pointer to array of client IP addresses }
   structure ServicesLayer {}
   structure MechanismsLayer {}
   structure PrimitivesLayer {} }
```

A.1.1.2. The Policies

```
structure Policy {
         // First, the selectors, and whether they are used
         boolean Next1ItemUsed      // If False, all destination addrs are selected.
         DestinationAddress
         boolean Next2ItemUsed
         SourceAddress
         boolean Next3ItemUsed
         NextProtocol
         boolean Next4ItemUsed
         IPSecurityLabel            // Level 1-16 (future work: more criteria)
         boolean Next5ItemUsed
         TransportDestinationPort
         boolean Next6ItemUsed
         TransportSourcePort
```

```
                              // If none of the above are used, all packets are
selected
              boolean Process              // True if these packets are to be processed
                              // False if these are the packets NOT to be
processed.
              // The negotiation goals
              // The policy data is to be used by the key exchange module to negotiate
       SAs.
              32-bit int SPI1Inner         // SPI of the inner SA set up - 0 if none
              32-bit int SPI2Outer         // SPI of the outer SA set up - 0 if none
              ProtocolInner       // Protocol type of the inner SA to be set up
              ModeInner           // The mode wanted: 1 for transport; 2 for tunnel
              ServNumInner        // Array index in the service structure
              boolean NegotiateInner              // if True, try different service
choices if selected one not accepted
              ProtocolOuter       // Protocol type of the outer SA to be set up
              ModeOuter
              ServNumOuter                 // Array index in the service structure
              boolean NegotiateOuter              // if True, try different service
choices if selected one not accepted
              MaxIPsecPacketSize
              … }
```

A.1.1.3. The Lower Three Layers (within the Overview)

```
       structure SMIB {
         structure PolicyLayer {}
         structure ManagementLayer {}
         structure ServicesLayer {
           structure AH {
             Enabled
             HMAC mech. no. }
           structure ESP {
             Enabled
             enc. mech. no. } }
         structure MechanismsLayer {
           structure Encryption {
             Enabled
             RNG prim. no. }
             enc. prim. no. }
           structure HMAC {
             Enabled
             RNG prim. no. }
             Hash. prim. no.
             enc. prim. no. } }
         structure PrimitivesLayer {
```

```
structure Encryption {
  Enabled
  enc. algorithm no.
  chaining mode }        // such as CBC
structure Hash {
  Enabled
  hash algorithm no. }
structure RNG {
  Enabled
  RNG algorithm no. } } }
```

In future work, each substructure of the Services to Primitives layers can be generalized to an array of the structures in order to select one at the higher layer. For now, each service, mechanism and primitive number is zero for each type because there is only one of each.

A.1.2. The SADB

```
structure SADB {
      FromIPAddress                // Set to all 255s for all
      ToIPAddress
      Protocol                     // 1=AH; 2=ESP
      Mode               // Same for AH and ESP 1=Transport 2=Tunnel
      32-bit int SPI     // default 1
      32-bit int SequenceNumber
      32-bit int AntiReplayWindow
      boolean SequenceNumberOverflow
      32-bit int LifeTime          // Number of bytes
      L3ServiceNo        // The SMIB service number negotiated
      Pointer to (address of) the key
      Pointer to the IV
      boolean IVConstant;          // - true if IV constant for this SA
      boolean OppositeSAIdentical;         // The SA for the other direction --
TRUE if so }
```

A.1.3. Top-Level Loop

```
Initialization
  -call the security Services layer with initialization command code.
  -fill in lower three SMIB layers to reflect programmed capabilities.
  -"key is always new" ← FALSE;
  -packet processing ← FALSE;
Loop forever {
  increment 64-bit counter for RNG seed (dec high 32 bits; inc low 32 bits)
  if (packet processing) {      // process the OSI stack
```

```
        IP Layer            // Note: each of these layers is called in sequence and each
checks the dispatcher for an incoming packet
        IPsec sublayer
        Link Layer }
      if (Command character received via RS232) {
        case of command character {
          h: board sends "Hello" characters via RS232
          r: board seeds the RNG with the 64-bit count and sends the count via
RS232
          z: board sends its SADB out via RS232
          s: board receives its SADB via RS232
          m: board sends its SMIB out via RS232
          i: board receives its SMIB via RS232
          k: toggle setting to initialize key in core every packet transform: "key is
always new"
          p: toggle packet processing } } }
```

Note: blocks of data in the r,z,s,m and i commands are communicated using two reliable data transfer routines: a send routine and a receive routine. These commands are not used in the test version (IPsecLoop).

A.1.4. Packet Processing, or OSI Layers

A.1.4.1. IP Layer

```
IP Layer: somewhat of a dummy layer {
  if (Start of Test) {
    Start of Test ← FALSE
    get the memory to receive the test packet via RS232
    receive the test packet via RS232 using reliable data transfer
    dispatch the packet to the next layer }

  check the dispatcher for a received packet
  if (Packet received) {
    send the packet received, out via RS232 using reliable data transfer
    delete the memory containing the packet received back
    packet processing ← FALSE – passed to top-level loop } }
```

A.1.4.2. IPsec Layer

```
IPsec Layer {
  if (not already a packet being sent to the Link layer) {
    check the dispatcher for a packet from the IP layer }
```

```
if (packet received from the IP layer) {      // Showing the packet sending code
    KeyIsNew ← KeyIsNew or KeyIsAlwaysTreatedAsNew or ((not
LastTimeWasSending) and KeysDiffer)
    depending upon the protocol in the SADB {
        use the reliable data transfer to synchronize with the GUI
        Level 3 AH or ESP Service(SADB, SMIB, SENDING, KeyIsNew,
NeedNewIV, IP packet in, IPsec packet out)
        KeyIsNew ← FALSE }
    echo the IPsec packet via RS232 using reliable data transfer - alerts the GUI
    dispatch the IPsec packet to the link layer
    delete the IP packet }
```

The receive code is analogous, with RECEIVING set in the service call, except that the protocol is read from the incoming packet and the SPI from the incoming packet is checked against the available incoming SADB(s). If the protocol is AH, a header verification message is sent via RS232. }

A.1.4.3. Link Layer

```
Link Layer: a dummy layer {
    check the dispatcher for a packet from the IPsec layer
    if there is a packet, put it back in the dispatcher to send it back }
```

A.1.5. Reliable Data Transfer

These are for the IPsecImp demonstration only. Each of the board and the GUI have a send and a receive routine

A.1.5.1. Send Algorithm

```
Send algorithm (pointer to character buffer, length) {
        Receive characters until STX from the recipient is received
        for each count of length, encode each of the two digits of the character
byte as hexadecimal ASCII
                and send the two digits
                -increment the checksum from the original byte
        read and clear any hanging characters sent from the recipient, to clear any
left-over STXes
        send the four-byte checksum the same way.
        if nothing received back from recipient, send ETX until something is
        while characters received back from recipient, echo back ACK or NAK
                -this ensures that the send code knows that the receiver got the
handshake, when it stops sending
```

```
                -quit and continue if recipient sent STX for another block of data
            if the transmission was ACKed, quit, otherwise repeat the above to try
again }
```

A.1.5.2. Receive Algorithm

```
    Receive algorithm (pointer to character buffer, length) {
            read any hanging characters and discard; stop if ETX received
            if ETX not received, send STX until a character is received
            get two characters for every count of length
                -each of the two is a hex digit; strip off the ASCII encoding and
make one byte
                -increment the checksum using the determined byte
            get the four-byte checksum the same way
            send ACK or NAK depending upon whether the checksums match
            get the handshake from the Sending entity
                -if the handshake is not ACK or NAK, resend the ACK or NAK
until the handshake is received.
            read any hanging characters and discard
            if the checksums matched, quit, otherwise repeat the above }
```

A.1.6. The AH Service

```
    L3AHServ (in: SADB, SMIB, Initialize, KeyIsNew, Sending, NewIVNeeded,
DatagramIn, out: DatagramOut, Verifies) {
        if (Initialize) L4MACMech(Initialize) and return.
        case SMIB→Hash and encryption primitive {
            get the hash size needed
            get the size of any extra space needed for signing (encryption) such as for
including the IV }
        get memory for the generated hash
        if (not Sending) get memory for the decrypted hash, for verification
        get size of and memory for the transformed datagram, depending upon
SADB→mode: Transport or Tunnel
        point to the location of the signed hash to write if Sending or to read if
receiving (pAuthData)
        get size of and memory for a copy of the datagram to hash (pPayloadToL4)
        fill the copy of the datagram to hash, leaving off the mutable fields – set to zero
        fill the transformed datagram out, skipping the location of the signed hash if
sending
        L4MACMech(In: SMIB→HMAC Mechanism number, SADB → key and IV,
SMIB, Initialize, KeyIsNew, Sending, NewIVNeeded, pPayloadToL4, Out (In if
receiving): pAuthData, Out: DecryptedHash, GeneratedHash)
        delete the space at pPayloadToL4
        if (not Sending) check the decrypted vs. generated hash and set Verifies accordingly
        delete the decrypted and generated hash memory }
    -Future work: use SADB service no → SMIB→Service layer→Mechanism layer and →Primitives layer when arrays added
```

A.1.7. The ESP Service

```
    L3ESPServ (in: SADB, SMIB, Initialize, KeyIsNew, Sending, NewIVNeeded, DatagramIn, out: DatagramOut) {
        if (Initialize) L4EncMech(Initialize) and return.
        case SMIB→ encryption primitive {
            get the encryption block size
            specify whether an IV is included }
        if (Sending) calculate the padding size required to make the payload a natural number multiple of the block size
        calculate the transformed DatagramOut size, depending on SADB → mode: Transport or Tunnel
        get the memory for an unencrypted/decrypted copy of the datagram, DatagramOutUnEnc, with all the padding; also if receiving, the padding has to be decrypted before the padding size can be retrieved.
        if (Sending) get the memory for the DatagramOut
        if (Sending) set pTransformedData in DatagramOut, else set pTransformedData in DatagramOutUnEnc
        if (Sending) set pPayloadToL4 to the payload location in DatagramOutUnEnc, else set it to the payload location in DatagramIn
        Set the header data in DatagramOutUnEnc
        if (Sending) set the data to be transformed in DatagramOutUnEnc and copy the header data to DatagramOut
        L4EncMech(In: SMIB → Enc Mechanism number, SADB → key and IV, SMIB, Initialize, KeyIsNew, Sending, NewIVNeeded, pPayloadToL4, Out: pTransformedData)
        if (not Sending) get the memory and copy DatagramOutUnEnc to DatagramOut without the padding
        delete DatagramOutUnEnc}
    Future work: use SADB service no → SMIB→Service layer→Mechanism layer and →Primitives layer when arrays added)
```

A.1.8. The HMAC Mechanism

```
    L4MACMech(In: SMIB→HMAC Mechanism number, SADB → key and IV, SMIB, Initialize, KeyIsNew, Sending, NewIVNeeded, pPayloadToL4, Out (In if receiving): AuthHdrStorage, Out: DecryptedHash, GeneratedHash) {
        if (Initialize) L4AES128Mode(NONE) return
        case SMIB→SMIB_Prims_layer.SMIB_Hash_Prims.Algorithm {
            SHA2: set the hash size
            L5SHA2 (pPayloadToL4, Out: GeneratedHash) }
        case SMIB→SMIB_Prims_layer.SMIB_Enc_Prims.Algorithm {
          AES128: get the key, if (Sending) get the IV
              determine whether the key needs to be set in the core from KeyIsNew
and L4AES128Mode
              if so, set the core use to MAC using L4AES128Mode() and call
L5AESI28UpdateKey
              if (NewIVNeeded and Sending) {
                Get the RNG state, Use the RNG to create a random number for the
IV, save the RNG state
                Pass back the new IV using its pointer }
              if (Sending) {
                put the IV into the output datagram at AuthHdrStorage
                L5AES128UpdateIVs() - put the IV into the AES core
                L5AES128EnDecrypt(ENCRYPT, GeneratedHash, HashSize,
(AuthHdrStorage+sizeof(IV)), HashSize) }
              else {
                get the IV from the input datagram at AuthHdrStorage
                L5AES128UpdateIVs() - put the IV into the AES core
                L5AES128EnDecrypt(DECRYPT, (AuthHdrStorage+sizeof(IV)),
HashSize, DecryptedHash, HashSize) } } }
```

A.1.9. The Encryption Mechanism

```
    L4EncMech(In: SMIB→Encryption Mechanism number, SADB → key and IV,
SMIB, Initialize, KeyIsNew, Sending, NewIVNeeded, pPayloadToL4, Out:
TransformStorage) {
        if (Initialize) L4AES128Mode(NONE), reset and save the RNG state, return
        case SMIB→SMIB_Prims_layer.SMIB_Enc_Prims.Algorithm {
          AES128: get the key, if (Sending) get the IV
              determine whether the key needs to be set in the core from KeyIsNew
and L4AES128Mode
              if so, set the core use to ENC using L4AES128Mode() and call
L5AESI28UpdateKey
              if (NewIVNeeded and Sending) {
                Get the RNG state, Use the RNG to create a random number for the
IV, save the RNG state
                Pass back the new IV using its pointer }
              if (Sending) {
                put the IV into the output datagram at TransformStorage
                L5AES128UpdateIVs() - put the IV into the AES core
                L5AES128EnDecrypt(ENCRYPT, pPayloadToL4+sizeof(IV), Size,
(TransformStorage+sizeof(IV)), Size) }
              else {
                get the IV from the input datagram at pPayloadToL4
                L5AES128UpdateIVs() - put the IV into the AES core
```

```
            L5AES128EnDecrypt(DECRYPT, pPayloadToL4+sizeof(IV), Size,
TransformStorage, Size) } } }
```

A.1.10.L4 (Mechanism) SPI Tracking Storage

```
    L4AES128SPI(Setting, SomethingSet, →SPI, →Decrypting) {
        static IsSomethingSet              // Whether anything is stored
        static SPIState
        static DecryptingState

        if (Setting) {
            IsSomethingSet = SomethingSet
            SPIState = →SPI
            DecryptingState = →Decrypting}
        else {
            →SPI = SPIState
            →Decrypting = DecryptingState}
        return(IsSomethingSet)}
```

A.1.11.The Hash Primitive

Dummy Hash

L5SHA2(MsgIn, MsgLen, 32-bit Out0 - 32-bit Out7) { set each of the eight 32-bit words to (hex) 5A5A5A5A }

A.1.12.The RNG Primitive

First, the state is stored using static variables within a function – this function must be used to retrieve and then to save the state before and after the Generate function.

A.1.12.1. Get or Set the State

```
    L5RngCoreValues(int Set, 32-bit HOCASR, 32-bit LOCASR, 32-bit HOLFSR,
32-bit LOLFSR) {
      if (Set) store HOCASR, LOCASR, HOLFSR, LOLFSR (the state) in local
static variables
      else retrieve the state }
```

A.1.12.2. Reset the State (to all "1"s)

```
    L5RngReset(32-bit HOCASR, 32-bit LOCASR, 32-bit HOLFSR, 32-bit
LOLFSR) {
      set the state to all binary 1s, i.e., hex FFFFFFFF for each }
```

A.1.12.3. Seed the RNG

```
L5RngSeed(32-bit HOSeedIn, 32-bit LOSeedIn, 32-bit HOCASR, 32-bit LOCASR, 32-bit HOLFSR, 32-bit LOLFSR) {
        HOCASR ← HOSeedIn, LOCASR ← LOSeedIn, HOLFSR ← HOSeedIn, LOLFSR ← LOSeedIn }
```

A.1.12.4. Generate a Random Number

```
L5RngGenerate(32-bit HOCASR, 32-bit LOCASR, 32-bit HOLFSR, 32-bit LOLFSR) {
    generate the random number as given previously and update the state
    return the 32-bit random number }
```

A.1.13. The AES-128 Primitive

A.1.13.1. Key Load to the Core

```
L5AES128UpdateKey(32-bit Key0, 32-bit Key1, 32-bit Key2, 32-bit Key3, Decryptor) {
    write the encryptor core slave registers 1-4 with Key 0 to 3
        if (Decryptor) {
      use the algorithm detailed previously to load the key expand values from
        the encryptor core to the decryptor core }
        The encryptor's IV was changed by loading the key expand values to the
          decryptor; just clear it even if not loading the decryptor. The calling
          routine will be responsible for setting it:
        L5AES128UpdateIVs(0, 0, 0, 0) }
```

A.1.13.2. IV Load to the Core

```
L5AES128UpdateIVs(32-bit IV0, 32-bit IV1, 32-bit IV2, 32-bit IV3) {
    set IVLoad in the encryptor core
    write the encryptor core slave registers 5-8 with IV0 to 3
    clear IVLoad in the encryptor core
    set IVLoad in the decryptor core
    write the decryptor core slave registers 5-8 with IV0 to 3
    clear IVLoad in the decryptor core }
```

A.1.13.3. The Encrypt or Decrypt Function

```
L5AES128EnDecrypt(int Encrypt, 32-bit CurrentBlockIn, BlockInLength, 32-bit CurrentBlockOut, BlockOutLength) {
        Validate user entry - BlockInLength must be a multiple of sixteen bytes, and BlockOutLength must be at least the same size - return with error if not the case.
        for each 128-bit (16-byte) block {
                // Note big-endian data orientation. Note that SlaveReg8 has to be written last for the transform to proceed; use the first 32-bit words as higher-order data. In the core, SlaveReg5 is highest order.
                if (Encrypt) {
                        write the encryptor core slave registers 5-8 with the four 32-bit words of the block in
                        Wait for the encryptor core to finish, although doing this seems somewhat ridiculous given how fast the core works (220 ns).
                        read the encryptor core slave registers 9-12 to the four 32-bit words of the block out }
                else {
                        do the same except using the decryptor core slave registers
} } }
```

A.2. IPsecLoop

Differences from IPsecImp are shown. Note that IPsecLoop has no GUI, since it is a CLI program. Also, it only differs in the top-level loop and OSI layers; all of L3 (Service) to L5 (Primitive) layers are identical.

A.2.1. Top-Level Loop

Differences from IPsecImp:

Initialization
- -set the SADB
- -set a specific embedded test packet

commands

r, z, s, m, and i are not needed

added:

o: toggle the loop testing between outgoing (encryption) and receiving back (decryption)

t: change the protocol and mode to be tested, in the SADB

i: change the increment by which to increase the test packet size

n: toggle "need new IV"; if set, a new IV is acquired via the RNG every packet transform

s: increment the packet size or revert to the starting size

A.2.2. Packet Processing, or OSI Layers

Differences from IPsecImp:

A.2.2.1. IP Layer

```
IP Layer:
   Start of test:
      copy the embedded test packet (so that the IPsecLayer won't delete the original)
   if packet received:
      echo the packet via plain RS232
```

A.2.2.1. IPsec Layer

```
IPsec Layer:

   The service calls are put into a loop containing:
      a dummy AES transform (the opposite one to the one used in the service call) to read via oscilloscope to mark the loop
      the service call
      a dummy transform to read via oscilloscope to mark the end of the service call
      test for error
      set "KeyIsNew" from top-level user selection
      check for a keystroke via RS232 and exit the loop if so
      check for the setting of the "Outgoing" vs. incoming test selection to exit the loop after only one execution if testing the other direction.
      delete the transformed packet if looping
```

A.3. IPsecGUI

The key packet processing functions are shown below.

A.3.1. Packet processing test setup

```
OnButtonSendPacket {
   initial validation of SMIB, SADB and their contents
   test for existance and validity of the test packet - header size, minimum 1-byte payload size
   verify that the size of the IPsec test packet produced will be no larger than (hex) FFFF
```

```
        send 'p' to start packet processing in the board
        if no error message from the board, use reliable data transfer to send the test
packet
          - display a progress dot after sending the packet size
        use reliable data transfer to receive "Synch" from the board to synchronize to
the board's actual service call
          - display a progress dot
        PacketOutgoing ← true
        PacketOutProcTime ← 0, PacketInProcTime ← 0
        Set the GUI timer for timer messages at a 1ms rate }
```

A.3.2. Packet processing timer processing

```
OnTimer {
  increment the appropriate count depending upon PacketOutgoing
  check for character received; exit if nothing
  stop the timer
  if error message from the board, display it and exit
  if (PacketOutgoing) {
    display the time taken for the outgoing processing for display purposes
    get the IPsec packet size via reliable data transfer; display a space and a progress dot
    get the IPsec packet using reliable data transfer; display an "o" for "outgoing
processing complete"
    use reliable data transfer to receive "Synch" from the board to synchronize to the
board's actual service call
    display a progress dot
    Set the GUI timer for timer messages at a 1ms rate }
  else {
    use reliable data transfer to receive the status message of the AH verification (or
dummy characters if ESP)
    display the time taken for the incoming processing for display purposes
    get the IP packet size via reliable data transfer; display a space and a progress dot
    get the IP packet using reliable data transfer; display an "i" for "incoming processing
complete"
    display the IP and the IPsec packet
    echo the protocol and mode for record-keeping
    echo the outgoing and incoming processing times for record-keeping
    echo the AH verification message for record-keeping } }
```

APPENDIX B

B. Experimental Data

B.1. Simulation Results

B.1.1. Encryption

B.1.1.1. Encryption – with All Intermediate Step Results

At the time these "screenshots" were taken, the "load pulse" method was used to initiate "ld_r" and the transform.

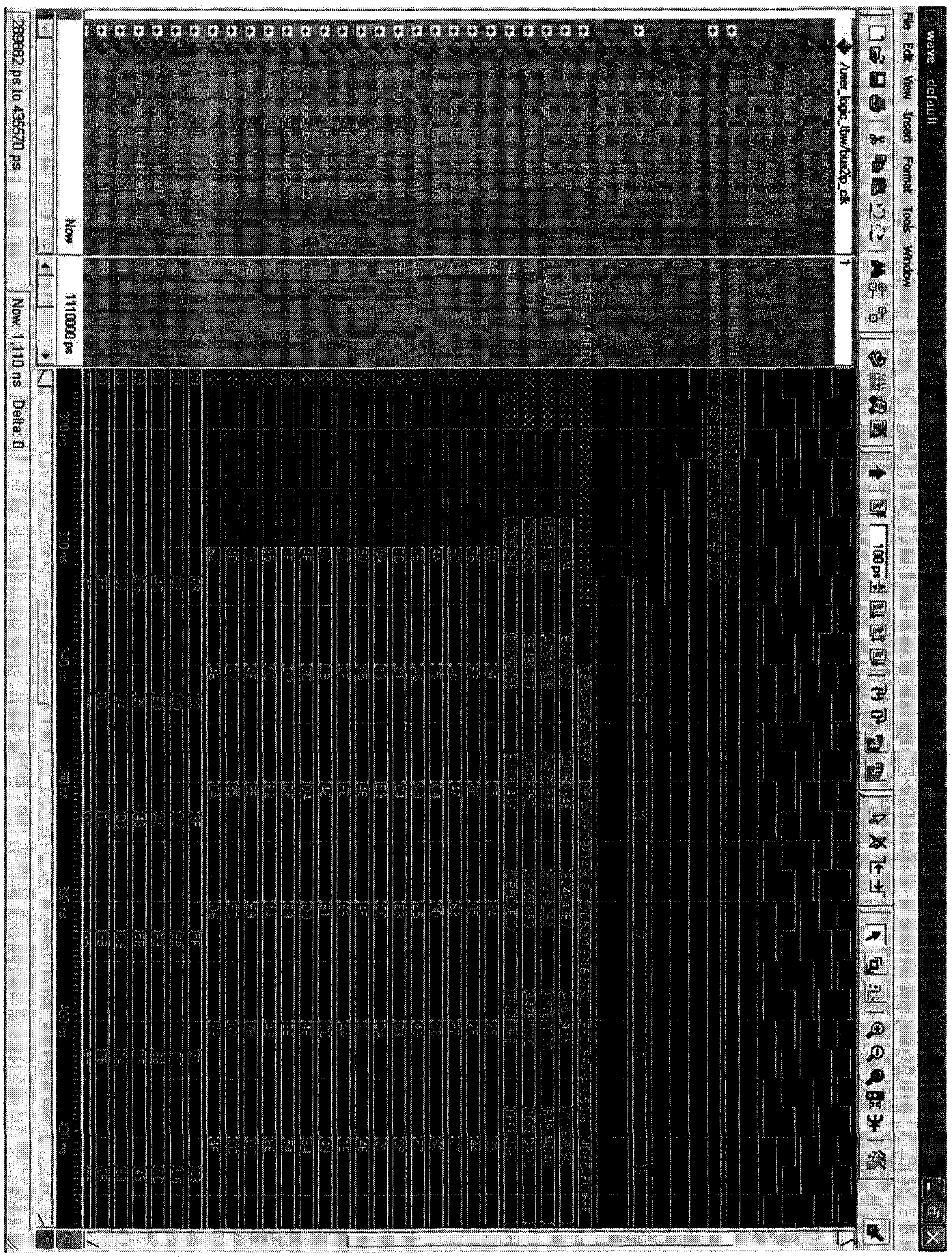


Figure 61. Encryption simulation "screenshot" – upper left quadrant of view

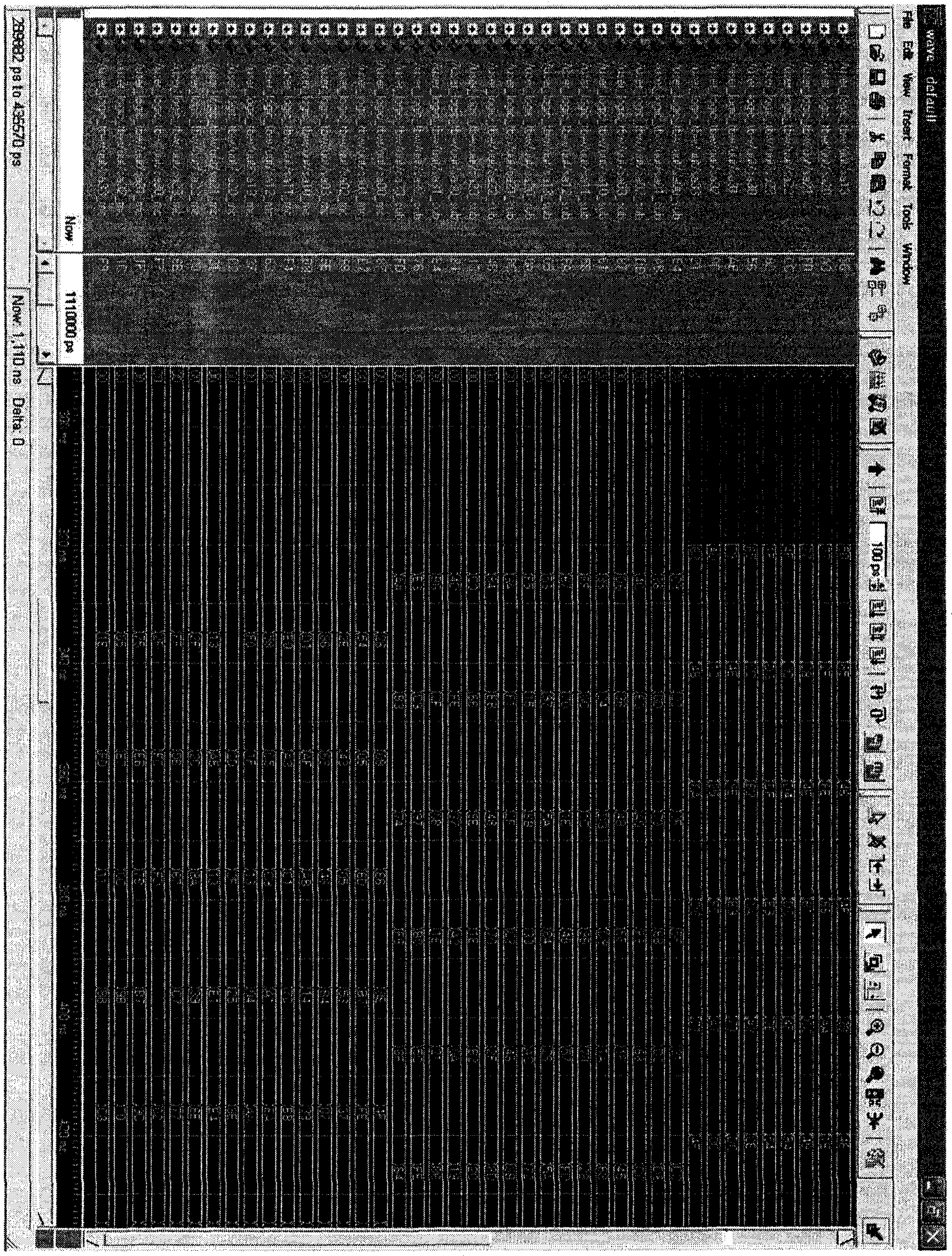


Figure 62. Encryption simulation "screenshot" – lower left quadrant of view

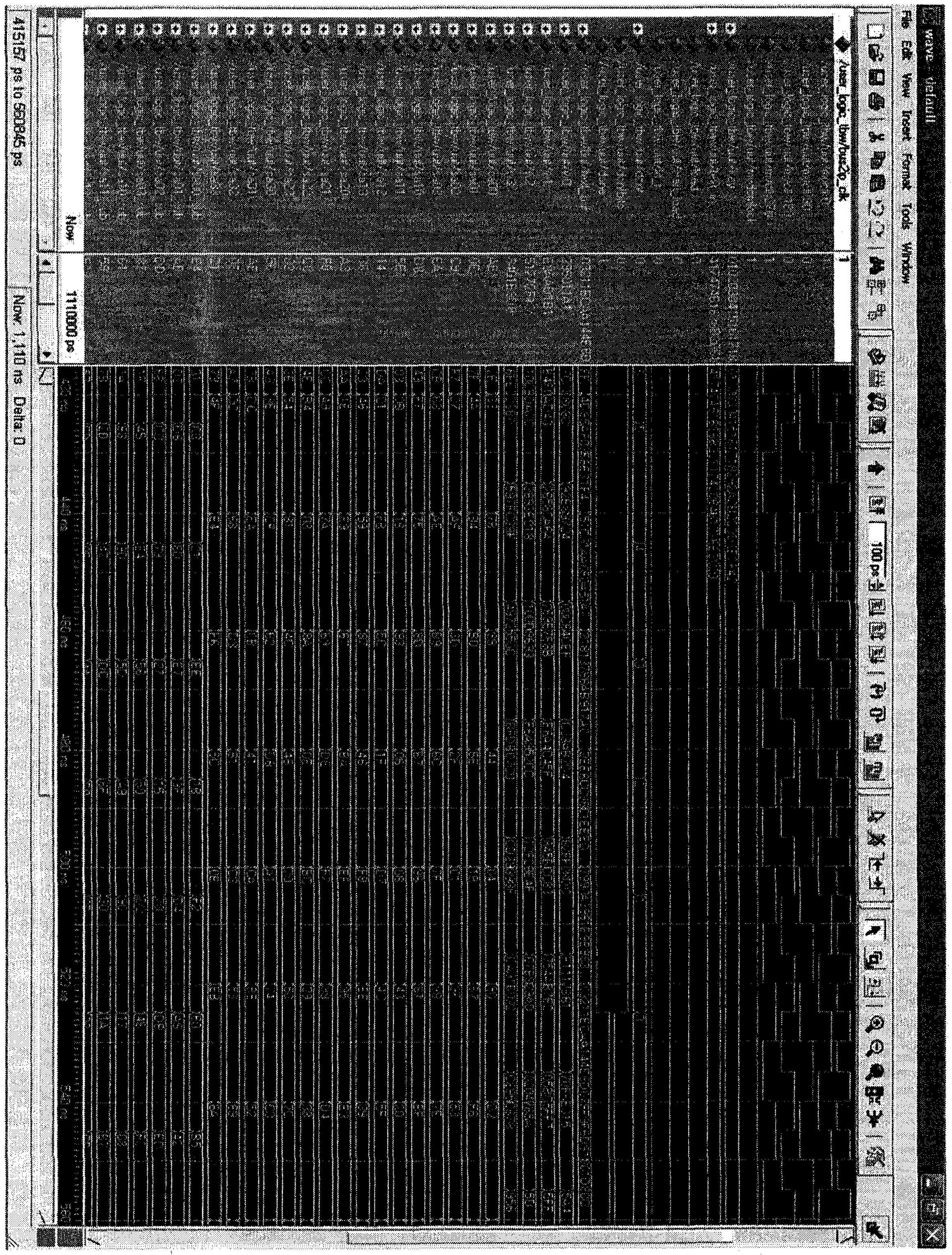


Figure 63. Encryption simulation "screenshot" – upper right quadrant of view

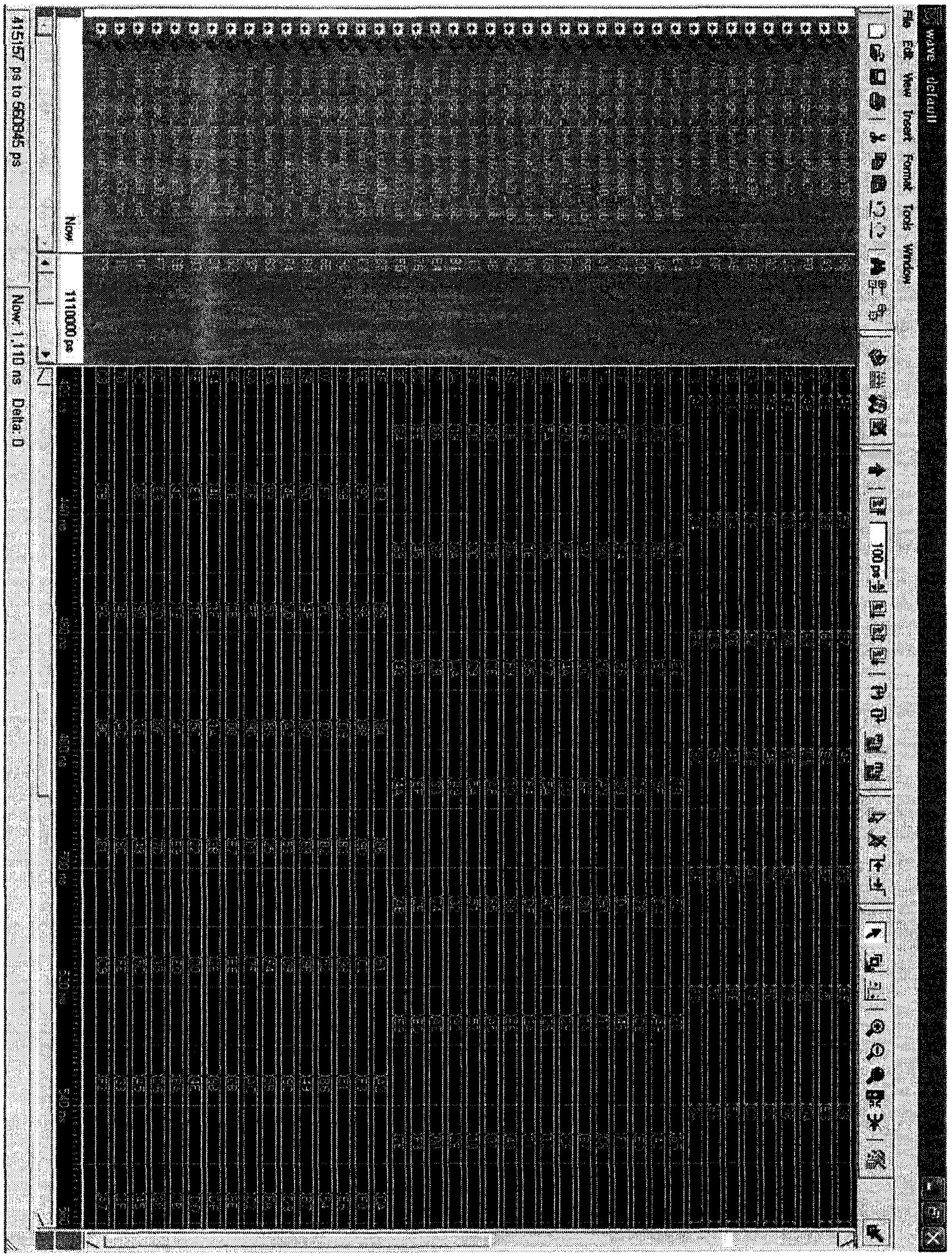


Figure 64. Encryption simulation "screenshot" – lower right quadrant of view

B.1.1.2. Autoload

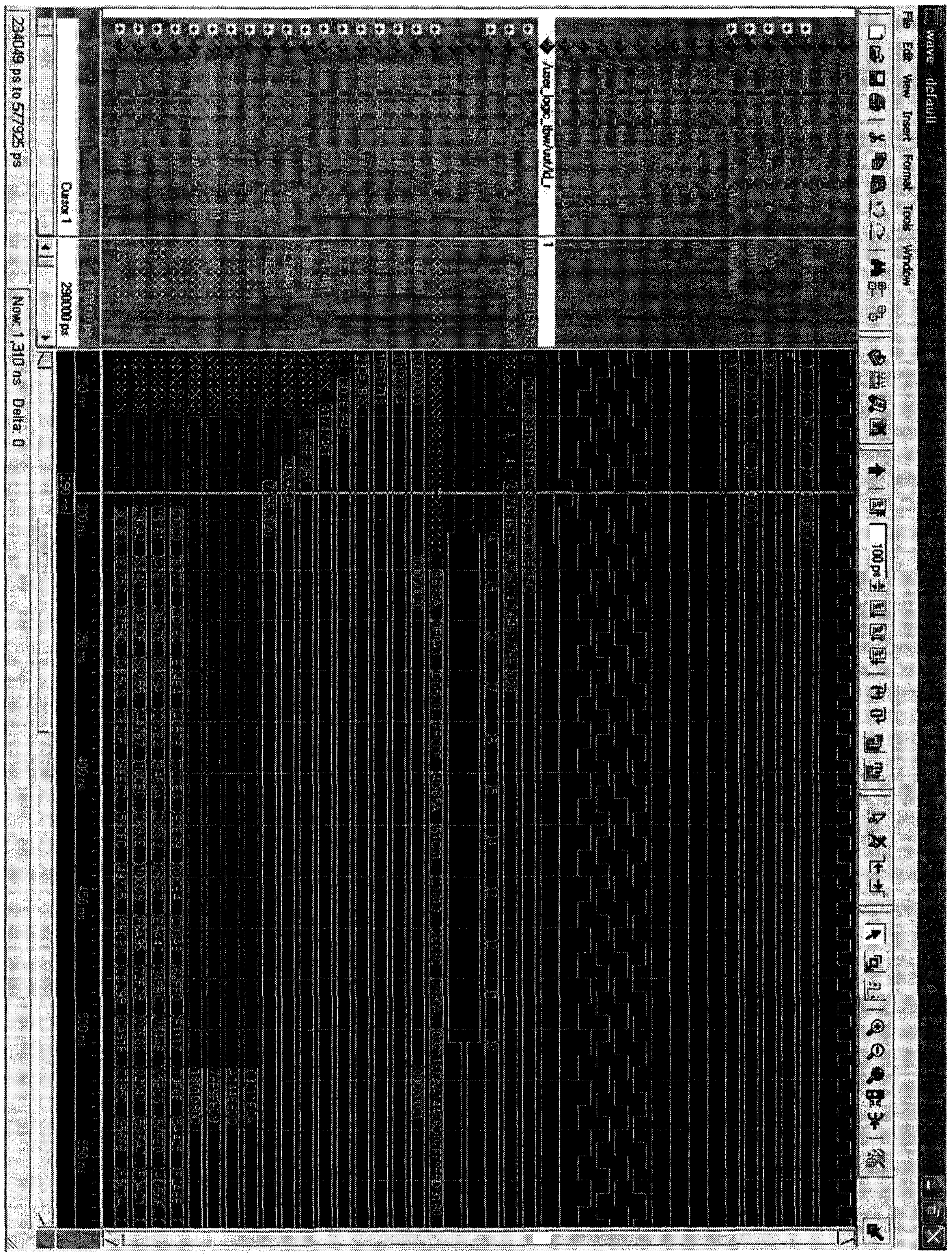


Figure 65. Encryption simulation "screenshot," showing autoload

B.1.1.3. Cipher-Block Chaining Mode

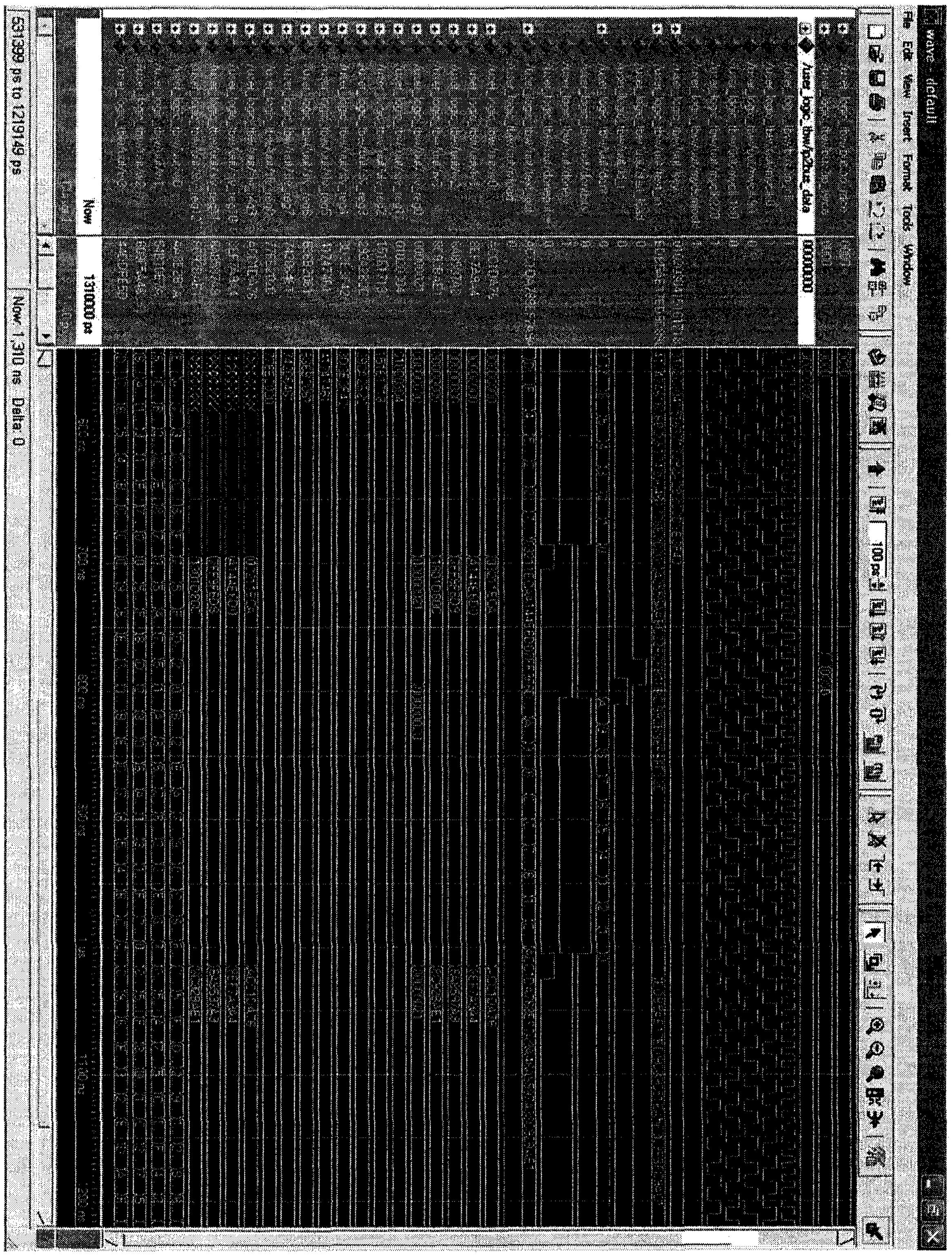


Figure 66. Encryption simulation "screenshot," showing CBC mode

B.1.2. Decryption

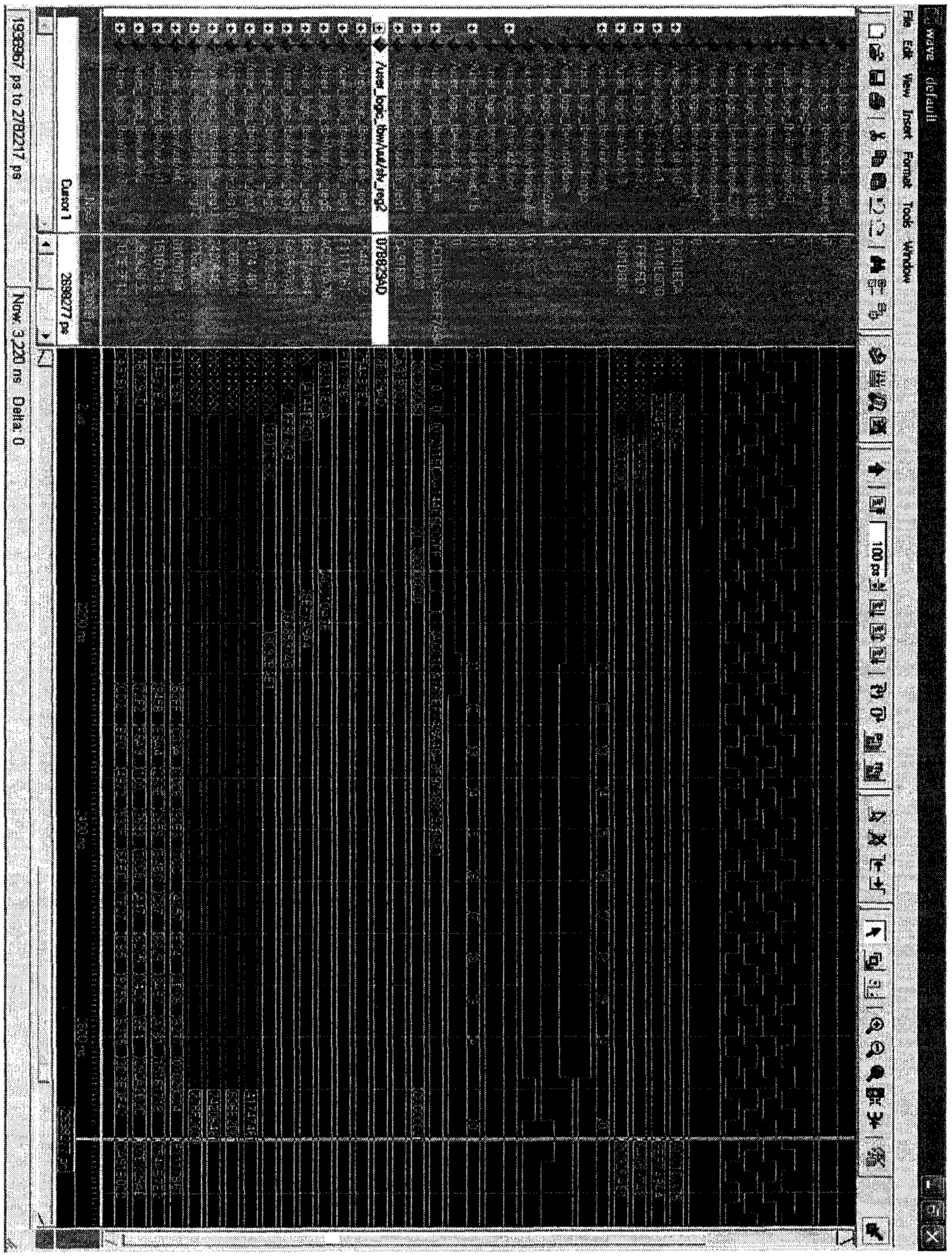


Figure 67. Decryption simulation "screenshot," showing IV load and save

B.2. Tabulated Data from “IPsecLoop” Oscilloscope Testing

B.2.1. Version 4, with “IPsecLoop” and “IPsecGUI” Comparison

B.2.1.1. AH Transport

AH Transport Mode packet processing times using IPsecLoop, - data of Mar 1-2, 2007
(entered here Mar 13-15) - James Wiebe
(Packet sizes in bytes - 8 bits in the ML403 board)
"GUI processing times" refer to use of the IPsecGUI; (1) laptop runs Mar 13-14, 2007
(2) on PC Mar 17, 2007

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms)	Incoming proc. time (ms)	GUI out. proc. time (ms) (1)	GUI inc. proc. time (ms) (1)	GUI out. proc. time (ms) (2)	GUI inc. proc. time (ms) (2)
28	40	0.72	0.97	2	2	1	1
128	296	3.9	3.6	2	2	1	1
228	552	5.7	6.3	2	1	1	1
328	808	7.6		2	1	1	1
428	1064	9.3	11.5	2	1	1	1
528	1320	11.3				1	2
628	1576	13	17	1	1	1	2
728	1832	15				2	1
828	2088	17	22	1	1	1	2
928	2344	18.8				2	2
a28	2600		28	1	1	1	3
c28	3112	24	33	1	2	1	3
e28	3624	28	38	1	2	2	2
1028	4136	31.5	44	2	3	3	3
1228	4648	35.5				3	3
1428	5160	39	54	2	4	3	4
1828	6184	46	64	3	5	3	5
1c28	7208	54	75	4	6	4	6
2028	8232	61	85	5	7	5	6
2428	9256	68	95	5	8	5	7
2828	10280	76	108	6	9	6	8
2c28	11304	83	118	7	10	5	8
3028	12328	90	128	7	11	6	8
3428	13352	98	140	7	12	7	9
3828	14376	108	150	9	14	7	10
3c28	15400	113	160	9	15	8	11
4028	16424	120	170	10	15	8	12

Table 10. AH Transport tabulated data – ver. 4, with “IPsecLoop” and “IPsecGUI” comparison

B.2.1.2. AH Tunnel

AH Tunnel Mode packet processing times using IPsecLoop, - data of Mar 3, 2007
(entered here Mar 15) - James Wiebe
(Packet sizes in bytes - 8 bits in the ML403 board)
"GUI processing times" refer to use of the IPsecGUI; (1) laptop runs Mar 13-14, 2007
(2) on PC Mar 17, 2007

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms)	Incoming proc. time (ms)	GUI out. proc. time (ms) (1)	GUI inc. proc. time (ms) (1)	GUI out. proc. time (ms) (2)	GUI inc. proc. time (ms) (2)
28	40	0.9	1.2	2	2	1	1
128	296	2.8	3.8	2	2	1	1
228	552	4.6	6.4	2	2	1	1
328	808			2	2	1	1
428	1064	8.3	12	2	1	1	1
528	1320					1	1
628	1576	12	17	1	1	1	2
728	1832					1	1
828	2088	17	22	1	1	2	2
928	2344					2	2
a28	2600	20	28	1	1	2	2
c28	3112	23	33	1	2	2	3
e28	3624	27	38	1	1	2	3
1028	4136	31	43	2	4	3	3
1228	4648					1	4
1428	5160	38	54	2	4	3	4
1828	6184	45	64	3	5	3	4
1c28	7208	53	75	4	8	3	5
2028	8232	60	85	4	6	4	6
2428	9256	68	95	5	8	5	7
2828	10280	75	105	6	9	5	7
2c28	11304	82	116	7	10	6	8
3028	12328	90	128	7	11	7	9
3428	13352	98	138	7	13	7	10
3828	14376	105	150	9	13	7	10
3c28	15400	111	160	9	14	8	10
4028	16424	120	170	10	15	7	11

Table 11. AH Tunnel tabulated data – ver. 4, with "IPsecLoop" and "IPsecGUI" comparison

B.2.1.3. ESP Transport

ESP Transport Mode packet processing times using IPsecLoop, - data of Mar 3, 2007
(entered here Mar 15) - James Wiebe

(Packet sizes in bytes - 8 bits in the ML403 board)
"GUI processing times" refer to use of the IPsecGUI; (1) laptop runs Mar 13-14, 2007
(2) on PC Mar 17, 2007

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms)	Incoming proc. time (ms)	GUI out. proc. time (ms) (1)	GUI inc. proc. time (ms) (1)	GUI out. proc. time (ms) (2)	GUI inc. proc. time (ms) (2)
28	40	0.6	0.41	2	2	1	1
128	296	2.7	1.8	2	2	1	1
228	552	4.7	3.2	2	2	1	1
328	808		4.6			1	1
428	1064	8.8	6	1	2	1	1
528	1320					1	1
628	1576	13	8.8	1	1	1	1
728	1832					2	1
828	2088	17	12	1	1	2	1
928	2344					2	2
a28	2600	21	14	1	1	2	1
c28	3112	25	17	1	1	2	2
e28	3624	30	20	1	1	3	2
1028	4136	34	23	2	1	2	2
1228	4648					3	2
1428	5160	42	28	2	1	3	2
1828	6184	50	34	3	2	4	2
1c28	7208	58	40	4	3	4	3
2028	8232	66	45	5	3	5	3
2428	9256	74	50			4	4
2828	10280	83	56	7	4	6	4
2c28	11304	90	62	7	4	7	4
3028	12328	100	68	7	5	6	5
3428	13352	108	73	8	6	7	6
3828	14376	115	78	10	1	7	6
3c28	15400	125	85	11	7	8	6
4028	16424	133	91	11	6	9	6

Table 12. ESP Transport tabulated data – ver. 4, with "IPsecLoop" and "IPsecGUI" comparison

B.2.1.4. ESP Tunnel

ESP Tunnel Mode packet processing times using IPsecLoop, - data of Mar 3, 2007 (entered here Mar 15) - James Wiebe
(Packet sizes in bytes - 8 bits in the ML403 board)
"GUI processing times" refer to use of the IPsecGUI; (1) laptop runs Mar 13-14, 2007
(2) on PC Mar 17, 2007

Packet size (hex)	Packet size (decimal)	Outgoing proc. time	Incoming proc. time	GUI out. proc. time (ms)	GUI inc. proc. time (ms) (1)	GUI out. proc.	GUI inc. proc. time

		(ms)	(ms)	(1)		time (ms) (2)	(ms) (2)
28	40	0.755	0.34	2	2	1	1
128	296	2.8	1.7	2	2	1	1
228	552	4.8	3.1	2	2	1	1
328	808	7.9	4.5			1	1
428	1064	8.9	5.9	2	2	1	1
528	1320					1	1
628	1576	13	8.7	1	2	2	1
728	1832					2	1
828	2088	17	12	1	1	1	1
928	2344					2	1
a28	2600	21	14	1	1	2	2
c28	3112	26	17	1	1	2	1
e28	3624	30	20	1	1	3	2
1028	4136	35	23	1	1	3	2
1228	4648					3	2
1428	5160	42	28	3	1	3	2
1828	6184	50	34	3	2	4	2
1c28	7208	58	40	3	2	4	3
2028	8232	66	45	5	2	5	4
2428	9256	75	51	5	3	5	3
2828	10280	83	56	6	3	6	4
2c28	11304	93	62	7	4	6	4
3028	12328	100	67	8	5	7	5
3428	13352	108	73	9	5	8	6
3828	14376	115	78	10	6	8	6
3c28	15400	125	85	10	7	7	6
4028	16424	133	90	11	7	9	7

Table 13. ESP Tunnel tabulated data – ver. 4, with "IPsecLoop" and "IPsecGUI" comparison

B.2.2. Version 6 Compared to Ver. 4

B.2.2.1. ESP Transport

ESP Transport Mode packet processing times using IPsecLoop, - data of Mar 20,
2007 - IPsecLoop Ver 006
(entered here Mar 26) - James Wiebe
(1) - Data of Mar 3, for comparison
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.6	0.41	0.39	0.37
128	296	2.7	1.8	1.3	1.4
228	552	4.7	3.2	2.3	2.3

328	808		4.6		
428	1064	8.8	6	4.2	4.2
528	1320				
628	1576	13	8.8	6	6.2
728	1832				
828	2088	17	12	7.9	8.1
928	2344				
a28	2600	21	14	9.8	10
c28	3112	25	17	12	12
e28	3624	30	20	14	14
1028	4136	34	23	16	16
1228	4648				
1428	5160	42	28	19	20
1828	6184	50	34	23	24
1c28	7208	58	40	28	28
2028	8232	66	45	30	31
2428	9256	74	50	35	35
2828	10280	83	56	38	39
2c28	11304	90	62	42	44
3028	12328	100	68	46	47
3428	13352	108	73	49	51
3828	14376	115	78	53	55
3c28	15400	125	85	57	59
4028	16424	133	91	60	63

Table 14. ESP Transport tabulated data – ver. 6 compared to ver. 4

B.2.2.2. ESP Tunnel

ESP Tunnel Mode packet processing times using IPsecLoop, - data of Mar 20,
2007 - IPsecLoop Ver 006
(entered here Mar 26) - James Wiebe
(1) - Data of Mar 3, for comparison
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.755	0.34	0.48	0.31
128	296	2.8	1.7	1.4	1.3
228	552	4.8	3.1	2.3	2.3
328	808	7.9	4.5		
428	1064	8.9	5.9	4.2	4.2
528	1320				
628	1576	13	8.7	6.1	6.1
728	1832				
828	2088	17	12	8	8.1
928	2344				
a28	2600	21	14	9.9	9.9

c28	3112	26	17	12	12
e28	3624	30	20	14	14
1028	4136	35	23	16	16
1228	4648				
1428	5160	42	28	19	20
1828	6184	50	34	23	23
1c28	7208	58	40	27	27
2028	8232	66	45	30	32
2428	9256	75	51	34	35
2828	10280	83	56	38	39
2c28	11304	93	62	42	43
3028	12328	100	67	46	47
3428	13352	108	73	49	51
3828	14376	115	78	54	55
3c28	15400	125	85	56	58
4028	16424	133	90	60	62

Table 15. ESP Tunnel tabulated data – ver. 6 compared to ver. 4

B.2.3. Version 7 Compared to Ver. 6 and 4

B.2.3.1. AH Transport

AH Transport Mode packet processing times using IPsecLoop, - data of Mar 21,
2007 - IPsecLoop Ver 007
(entered here Mar 26) - James Wiebe
(1) - Data of Mar 1-2, for comparison
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.72	0.97	0.56	0.84
128	296	3.9	3.6	2	3.1
228	552	5.7	6.3	3.4	5.3
328	808	7.6			
428	1064	9.3	11.5	6.2	9.6
528	1320	11.3			
628	1576	13	17	9	14
728	1832	15			
828	2088	17	22	12	19
928	2344	18.8			
a28	2600		28	15	23
c28	3112	24	33	18	28
e28	3624	28	38	21	32
1028	4136	31.5	44	23	36
1228	4648	35.5		26	41
1428	5160	39	54	29	45
1828	6184	46	64	35	54

1c28	7208	54	75	40	63
2028	8232	61	85	46	72
2428	9256	68	95	52	80
2828	10280	76	108	58	88
2c28	11304	83	118	63	98
3028	12328	90	128	68	108
3428	13352	98	140	73	115
3828	14376	108	150	80	125
3c28	15400	113	160	85	133
4028	16424	120	170	90	143

Table 16. AH Transport tabulated data – ver. 7 compared to ver. 4

B.2.3.2. AH Tunnel

AH Tunnel Mode packet processing times using IPsecLoop, - data of Mar 21,
2007 - IPsecLoop Ver 007
(entered here Mar 26) - James Wiebe
(1) - Data of Mar 1-2, for comparison
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.9	1.2	0.69	0.975
128	296	2.8	3.8	2.1	3.2
228	552	4.6	6.4	3.6	5.4
328	808				
428	1064	8.3	12	6.4	9.8
528	1320				
628	1576	12	17	9.3	14
728	1832				
828	2088	17	22	12	19
928	2344				
a28	2600	20	28	15	23
c28	3112	23	33	18	28
e28	3624	27	38	21	32
1028	4136	31	43	23	36
1228	4648				
1428	5160	38	54	29	45
1828	6184	45	64	35	54
1c28	7208	53	75	41	62
2028	8232	60	85	46	72
2428	9256	68	95	52	80
2828	10280	75	105	58	90
2c28	11304	82	116	63	99
3028	12328	90	128	70	106
3428	13352	98	138	75	115
3828	14376	105	150	80	125

3c28	15400	111	160	85	131
4028	16424	120	170	93	143

Table 17. AH Tunnel tabulated data – ver. 7 compared to ver. 4

B.2.3.3. ESP Transport

ESP Transport Mode packet processing times using IPsecLoop, - data of Mar 21, 2007 - IPsecLoop Ver 007
(entered here Mar 26) - James Wiebe
(1) - Data of Mar 3 - ver 4
(2) - Data of Mar 20 - ver 6
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms) (2)	Incoming proc. time (ms) (2)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.6	0.41	0.39	0.37	0.38	0.315
128	296	2.7	1.8	1.3	1.4	1.3	1.3
228	552	4.7	3.2	2.3	2.3	2.2	2.2
328	808		4.6				
428	1064	8.8	6	4.2	4.2	4.1	4.1
528	1320						
628	1576	13	8.8	6	6.2	5.9	6
728	1832						
828	2088	17	12	7.9	8.1	7.8	8
928	2344						
a28	2600	21	14	9.8	10	9.6	9.9
c28	3112	25	17	12	12	12	12
e28	3624	30	20	14	14	13	14
1028	4136	34	23	16	16	15	16
1228	4648						
1428	5160	42	28	19	20	19	20
1828	6184	50	34	23	24	23	23
1c28	7208	58	40	28	28	26	27
2028	8232	66	45	30	31	30	31
2428	9256	74	50	35	35	34	35
2828	10280	83	56	38	39	38	39
2c28	11304	90	62	42	44	41	43
3028	12328	100	68	46	47	45	46
3428	13352	108	73	49	51	48	50
3828	14376	115	78	53	55	52	54
3c28	15400	125	85	57	59	56	58
4028	16424	133	91	60	63	60	62

Table 18. ESP Transport tabulated data – ver. 7 compared to ver. 6 and 4

B.2.3.4. ESP Tunnel

ESP Tunnel Mode packet processing times using IPsecLoop, - data of Mar 21, 2007 - IPsecLoop Ver 007
(entered here Mar 26) - James Wiebe
(1) - Data of Mar 3 - ver 4
(2) - Data of Mar 20 - ver 6
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms) (2)	Incoming proc. time (ms) (2)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.755	0.34	0.48	0.31	0.44	0.255
128	296	2.8	1.7	1.4	1.3	1.4	1.2
228	552	4.8	3.1	2.3	2.3	2.3	2.2
328	808	7.9	4.5				
428	1064	8.9	5.9	4.2	4.2	4.2	4.1
528	1320						
628	1576	13	8.7	6.1	6.1	6	6
728	1832						
828	2088	17	12	8	8.1	7.8	7.9
928	2344						
a28	2600	21	14	9.9	9.9	9.7	9.8
c28	3112	26	17	12	12	12	12
e28	3624	30	20	14	14	14	14
1028	4136	35	23	16	16	15	16
1228	4648						
1428	5160	42	28	19	20	19	19
1828	6184	50	34	23	23	23	23
1c28	7208	58	40	27	27	27	27
2028	8232	66	45	30	32	30	31
2428	9256	75	51	34	35	34	35
2828	10280	83	56	38	39	38	39
2c28	11304	93	62	42	43	41	42
3028	12328	100	67	46	47	45	46
3428	13352	108	73	49	51	49	50
3828	14376	115	78	54	55	53	54
3c28	15400	125	85	56	58	56	58
4028	16424	133	90	60	62	60	62

Table 19. ESP Tunnel tabulated data – ver. 7 compared to ver. 6 and 4

B.2.4. Version 8 Compared to Ver. 7

B.2.4.1. AH Transport

AH Transport Mode packet processing times using IPsecLoop, - data of Apr 9,

2007 - IPsecLoop Ver 008
(entered here Apr 10) - James Wiebe
(1) - Data of IPsecLoop ver 007, Mar 21, for comparison
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.56	0.84	0.435	0.83
128	296	2	3.1	1.19	3.03
228	552	3.4	5.3	1.94	5.2
328	808				
428	1064	6.2	9.6	3.43	9.6
528	1320				
628	1576	9	14	4.9	14
728	1832				
828	2088	12	19	6.4	18.4
928	2344				
a28	2600	15	23	7.9	22.8
c28	3112	18	28	9.25	27.3
e28	3624	21	32	10.9	31.8
1028	4136	23	36	12.4	36
1228	4648	26	41	13.8	40.3
1428	5160	29	45	15.3	45
1828	6184	35	54	18.3	53
1c28	7208	40	63	21.3	62
2028	8232	46	72	24.3	71
2428	9256	52	80	27.3	79.5
2828	10280	58	88	30.3	88
2c28	11304	63	98	33.3	98
3028	12328	68	108	36.3	106
3428	13352	73	115	39	115
3828	14376	80	125	42	123
3c28	15400	85	133	45	133
4028	16424	90	143	48	141

Table 20. AH Transport tabulated data – ver. 8 compared to ver. 7

B.2.4.2. AH Tunnel

AH Tunnel Mode packet processing times using IPsecLoop, - data of Apr 9,
2007 - IPsecLoop Ver 008
(entered here Apr 10) - James Wiebe
(1) - Data of Mar 21, IPsecLoop ver 007, for comparison
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)

28	40	0.69	0.975	0.51	0.975
128	296	2.1	3.2	1.25	3.2
228	552	3.6	5.4	2	5.4
328	808				
428	1064	6.4	9.8	3.5	9.88
528	1320				
628	1576	9.3	14	4.95	14.1
728	1832				
828	2088	12	19	6.45	18.6
928	2344				
a28	2600	15	23	7.95	23
c28	3112	18	28	9.38	27.3
e28	3624	21	32	10.9	32
1028	4136	23	36	12.8	36.5
1228	4648			13.9	41
1428	5160	29	45	15.4	45
1828	6184	35	54	18.3	54
1c28	7208	41	62	21.3	63
2028	8232	46	72	24.3	71.5
2428	9256	52	80	27.3	81.5
2828	10280	58	90	30.3	89
2c28	11304	63	99	33.3	98
3028	12328	70	106	36.3	108
3428	13352	75	115	39.3	116
3828	14376	80	125	42.3	125
3c28	15400	85	131	45	133
4028	16424	93	143	48	142

Table 21. AH Tunnel tabulated data – ver. 8 compared to ver. 7

B.2.4.3. ESP Transport

ESP Transport Mode packet processing times using IPsecLoop, - data of Apr 9,
2007 - IPsecLoop Ver 008
(entered here Apr 10) - James Wiebe
(1) - Data of Mar 21 - ver 7
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.38	0.315	0.27	0.24
128	296	1.3	1.3	1.23	1.33
228	552	2.2	2.2	2.16	2.4
328	808				
428	1064	4.1	4.1	4.1	4.55
528	1320				
628	1576	5.9	6	5.9	6.7
728	1832				

828	2088	7.8	8	7.8	8.9
928	2344				
a28	2600	9.6	9.9	9.63	11
c28	3112	12	12	11.5	13.1
e28	3624	13	14	13.4	15.4
1028	4136	15	16	15.3	17.5
1228	4648			17.3	19.8
1428	5160	19	20	19.1	21.9
1828	6184	23	23	22.8	26.3
1c28	7208	26	27	26.5	30.3
2028	8232	30	31	30.3	34.8
2428	9256	34	35	34	39.3
2828	10280	38	39	37.8	43.3
2c28	11304	41	43	41.5	47.5
3028	12328	45	46	45	52
3428	13352	48	50	49	56.5
3828	14376	52	54	53	61
3c28	15400	56	58	56.5	65
4028	16424	60	62	60	69

Table 22. ESP Transport tabulated data – ver. 8 compared to ver. 7

B.2.4.4. ESP Tunnel

ESP Tunnel Mode packet processing times using IPsecLoop, - data of Apr 9,
2007 - IPsecLoop Ver 008
(entered here Apr 10) - James Wiebe
(1) - Data of Mar 21 - ver 7
(Packet sizes in bytes - 8 bits in the ML403 board)

Packet size (hex)	Packet size (decimal)	Outgoing proc. time (ms) (1)	Incoming proc. time (ms) (1)	Outgoing proc. time (ms)	Incoming proc. time (ms)
28	40	0.44	0.255	0.34	0.26
128	296	1.4	1.2	1.3	1.35
228	552	2.3	2.2	2.23	2.43
328	808				
428	1064	4.2	4.1	4.1	4.6
528	1320				
628	1576	6	6	6	6.75
728	1832				
828	2088	7.8	7.9	7.9	8.9
928	2344				
a28	2600	9.7	9.8	9.8	11.1
c28	3112	12	12	11.6	13
e28	3624	14	14	13.5	15.4
1028	4136	15	16	15.4	17.5
1228	4648			17.3	19.8
1428	5160	19	19	19.1	21.9

1828	6184	23	23	22.8	26.3
1c28	7208	27	27	26.5	30.5
2028	8232	30	31	30.5	34.8
2428	9256	34	35	34.3	39.3
2828	10280	38	39	37.8	43.5
2c28	11304	41	42	41.5	47.5
3028	12328	45	46	45	52
3428	13352	49	50	49.5	56
3828	14376	53	54	54	60.5
3c28	15400	56	58	57	65.5
4028	16424	60	62	60.5	69.5

Table 23. ESP Tunnel tabulated data – ver. 8 compared to ver. 7

REFERENCES

R.1. Journal and Conference Papers

[AIE2002] Aiello, W., Bellovin, S., Blaze, M., Canetti, R., Ioannidis, J., Keromytis, A., and Reingold, O., "Efficient, DoS-Resistant, Secure Key Exchange for Internet Protocols," *Proc. of the 9th ACM Conf. on Computer and Communications Security*, Nov. 2002, pp. 48-58.

[ARK2005] Arkko, J., and Nikander, P., "Limitations of IPsec Policy Mechanisms," *Lecture Notes in Computer Science*, Vol. 3364, 2005, Springer-Verlag GmbH, pp. 241-251.

[BEL2002] Bellows, P., Flidr, J., Lehman, T., Schott, B., and Underwood, K., "GRIP: a Reconfigurable Architecture for Host-based Gigabit-rate Packet Processing," *Proc.10th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, 22-24 Apr., 2002, pp. 121-130.

[CHE2002] Cheung, O., and Leong, P., "Implementation of an FPGA Based Accelerator for Virtual Private Networks," *Proc., 2002 IEEE Int'l Conf. on Field-Programmable Tech.*, 16-18 Dec., 2002, pp. 34-41.

[CHO2003] Choi, M., Kwak, D, and Moon, S., "A Proposal for DoS-Defensive Internet Key Exchange," *Lecture Notes in Computer Science*, Vol. 2668, 2003, Springer-Verlag GmbH, pp. 328-337.

[COM2002] Compton, K., and Hauck, S., "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, Vol. 34, No. 2, June 2002, pp. 171–210.

[DAN2000] Dandalis, A., Prasanna, V., and Rolim, J., "An Adaptive Cryptographic Engine for IPSec Architectures," *2000 IEEE Symp. on Field-Programmable Custom Computing Machines*, 17-19 April, 2000 pp.132-141.

[DEE2001] Deepakumara, J., Heys, H. and Venkatesan, R., "FPGA Implementation of MD5 Hash Algorithm," *Canadian Conf. on Electrical and Computer Engineering (CCECE 2001)*, Vol. 2, 13-16 May 2001, pp. 919-924.

[DON2004] Dong, Y., Choi, C., and Zhang, Z., "A Security Framework for Protecting Traffic Between Collaborative Domains," *Microprocessors and Microsystems* 28, Dec. 2004, Elsevier Science Ltd., pp. 547-559.

[DUF2002] Duflos, S., Kervella, B. and Horlait, E., "An Architecture for Policy-based Security Management for Distributed Multimedia Services," *Proc. of the Tenth ACM Int'l Conf. on Multimedia*, Dec. 2002, ACM, pp. 653-655.

[DUN2001] Dunbar, N., "IPsec Networking Standards - An Overview," *Information Security Technical Report*, Vol. 6, No. 1, March 2001, Elsevier Science Ltd., pp. 35-48.

[ELB2000] Elbirt, A. and Paar, C., "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher," *Proc. of the 2000 ACM/SIGDA Eighth Int'l Symp. on Field Programmable Gate Arrays (FPGA '2000)*, Feb. 2000, ACM Press, pp. 33-40.

[FER1999] Ferguson, N. and Schneier, B., "A Cryptographic Evaluation of IPSec," Counterpane Internet Security Inc., San Jose, CA, 1999.

[FER2005] Ferrante A., Piuri V., Castanier F., "A QoS-enabled Packet Scheduling Algorithm for IPSec Multi-Accelerator Based Systems," *Proc. of the 2nd Conf. on Computing Frontiers*, May 2005, ACM, pp. 221-229.

[FUM1998] Fumy, W., "Internet Security Protocols," *Lecture Notes in Computer Science*, Vol. 1528, 1998, Springer-Verlag GmbH, pp. 186-208.

[GAB2004] Gabrijelčič, D., Blazič, B., and Tasič, J., "Future Active IP Networks Security Architecture," *Computer Communications* 28, Aug. 2004, Elsevier Ltd., pp.688-701.

[GEN2006] Genz, C., and Drechsler, R., "System Exploration of SystemC Designs," *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006*, Volume 00, 2-3 March 2006.

[GOD2002] Godber, A., and Dasgupta, P., "Secure Wireless Gateway," *Proc. of the 3rd ACM workshop on Wireless security*, Sept. 2002, ACM, pp. 41-46.

[GUT2004] Guttman, J., and Herzog, A., "Rigorous Automated Network Security Management," *International Journal of Information Security*, Vol. 4, Nos. 1-2, Dec. 2004, Springer-Verlag GmbH, pp. 29-48.

[HEL2003] Hellekalek, P., "Empirical Evidence Concerning AES," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Vol. 13, Issue 4, Oct. 2003, pp. 322-333.

[HUN1998] Hunt, R., "Internet/Intranet Firewall Security-policy, Architecture and Transaction Services," *Computer Communications* 21, Sept. 1998, Elsevier Ltd., pp.1107-1123.

[JAR2003] Järvinen, K., Tommiska, M. and Skyttä, J., "A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor," *Proc. of the 2003 ACM/SIGDA 11th Int'l Symposium on FPGAs*, Feb. 2003, pp. 207-215.

[KAN2002] Kang, Y., Kim, D., Kwon, T., Choi, J., "An Efficient Implementation of Hash Function Processor for IPSEC," *Proc. of the 2002 IEEE Asia-Pacific Conference on ASIC*, 6-8 Aug. 2002, ACM Press, pp. 93-96.

[KAN2004] Kanda, M., Miyazawa, K. and Esaki, H., "USAGI IPv6 IPsec development for Linux," *Proc. of the 2004 Int'l Symp. on Applications and the Internet Workshops (SAINTW'04)*, 26-30 Jan. 2004, IEEE, pp.159-163.

[KAR2000] Karras, D., and Zorkadis, V., "Overfitting in Multilayer Perceptrons as a Mechanism for (Pseudo) Random Number Generation in the Design of Secure Electronic Commerce Systems," *EUROCOMM 2000, Information Systems for Enhanced Public Safety and Security*, 17 May 2000, IEEE, pp. 345-349..

[KER1997] Keromytis, A., Ioannidis, J. and Smith, J., "Implementing IPsec," *Global Telecom. Conf.*, Nov. 1997, Vol. 3, IEEE, pp.1948-1952.

[KIM2002] Kim, G., Park, W., Nah, J. and Sohn, S., "Security Policy Deployment in IPSec," *Lecture Notes in Computer Science*, 2002, Vol. 2344, Springer-Verlag, pp. 453-464.

[KIM2004] Kim, H. and Lee, S., "Design and Implementation of a Private and Public Key Crypto Processor and its Application to a Security System," *IEEE Transactions on Consumer Electronics*, Feb 2004, Vol. 50, Issue 1, pp. 214-224.

[LEC1998] L'Ecuyer, P., "Uniform Random Number Generators," *Proc. of the 1998 Winter Simulation Conference*, 13-16 Dec. 1998, Vol. 1, IEEE, pp. 97-104.

[LEI2000] Leiwo, J., Aura, T., and Nikander, P., "Towards Network Denial of Service Resistant Protocols," *Proc. of the 15th Int'l Information Security Conf. (IFIP/SEC 2000)*, Beijing, China, August 2000, pp. 1-10.

[LIH2005] Li, H., Katkoori, S., and Liu, Z., "Feedback Driven High Level Synthesis for Performance Optimization," *ASICON 2005. 6th Int'l Conf. On ASIC*, Vol. 2, 24-27 Oct. 2005, IEEE, pp. 882-885.

[LIM2003] Li, M., "Policy-Based IPsec Management," *IEEE Network*, Nov/Dec 2003, IEEE, pp. 36-43.

[LUJ2005] Lu, J. and Lockwood, J., "IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application," *Proc.of the 19th IEEE Int'l Parallel and Distributed Processing Symp.*, 4-8 Apr. 2005, pp. 158b-164b.

[MCF1988] McFarland, M., Parker, A., and Carnposano, R., "Tutorial on High-Level Synthesis," *25th ACM/IEEE Design Automation Conf.*, 1988, pp. 330-336.

[MCL2002] McLoone, M., and McCanny, J., "A Single-chip IPSEC Cryptographic Processor," *Proc. IEEE Workshop on Signal Processing Systems*, Oct. 2002, pp. 133-138.

[NAY2005] "Modeling and Evaluation of Security Architecture for Wireless Local Area Networks by Indexing Method: A Novel Approach," *Lecture Notes in Computer Science*, Vol. 3439, 2005, Springer-Verlag GmbH, pp. 25-35.

[PAR2002] Park, W., Nah, J. and Sohn, S., "A Study of Security Association Management Oriented to IP Security," *Lecture Notes in Computer Science*, 2002, Vol. 2344, Springer-Verlag GmbH, pp. 381-388.

[REJ2003] Rejeb, J. and Ramaswamy, V., "Efficient Rijndael implementation for high-speed optical networks," *10th Int'l Conf. on Telecommunications, 2003 (ICT 2003)*, Vol. 1, 23 Feb.-1 Mar. 2003, IEEE, pp. 641-645.

[ROE2001] Roe, M. "Authentication and Naming (Transcript of Discussion)," *Lecture Notes in Computer Science*, 2001, Vol. 2133, Springer-Verlag GmbH, pp. 20-23.

[SCH2002] Schaumont, P., Kuo, H., and Verbauwhede, I., "Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor," *Proc. of the 39th Conf. on Design Auto.*, Jun 10-14, 2002.

[SOT1999] Soto, J., "Statistical Testing of Random Number Generators," *Proc. of the 22nd National Information Systems Security Conf.*, Oct. 1999, National Institute of Standards & Technology. Available: http://csrc.nist.gov/rng/nissc-paper.pdf.

[STN2003] Standaert, F., Rouvroy, G., Quisquater, J. and Legat, J., "A Methodology to Implement Block Ciphers in Reconfigurable Hardware and its Application to Fast and Compact AES Rijndael," *Proc. of the 2003 ACM/SIGDA 11th Int'l Symposium on Field Programmable Gate Arrays (FPGA '03)*, 23-25 Feb. 2003, ACM Press, pp. 216-224.

[TRC2003] Trček, D., "An Integral Framework for Information Systems Security Management," *Computers & Security*, May 2003, Vol. 22, No. 4, Elsevier Ltd., pp.337-360.

[VER2003] Verbauwhede, I., Schaumont, P., and Kuo, H., "Design and Performance Testing of a 2.29 GB/s Rijndael Processor," *IEEE Journal Of Solid-State Circuits*, Vol. 38, No. 3, March 2003, pp. 569-572.

[WAN2004] Wang, M., Su, C., Huang, C. and Wu, C., "An HMAC processor with integrated SHA-1 and MD5 algorithms," *Proc. of the 2004 Conf. on Asia South Pacific Design Automation: Electronic Design and Solution Fair (ASP-DAC '04)*, January 2004, IEEE, pp. 456-458.

[WEI2004] Wei, C, Chengshu, L, and Xin, S., "FPGA Implementation Of Universal Random Number Generator," *ICSP '04, 7th Int'l Conf. On Signal Processing*, Vol. 1, 31 Aug.-4 Sept. 2004, IEEE, pp. 495-498.

[WIE2006] Wiebe, J., "IPSec Implementation and Management Methods," *Int'l Conference for Upcoming Engineers (ICUE)*, May 13-14, 2006.

[WOL2004] Wollinger, T., Guajardo, J., and Paar, C., "Security on FPGAs: State-of-the-Art Implementations and Attacks," *ACM Trans. on Embedded Computing Systems*, Aug 2004, Vol. 3, No. 3, pp. 534-574.

[WUL2001] Wu L., Weaver, C. and Austin, T., "CryptoManiac: a Fast Flexible Architecture for Secure Communication," Proc. of the 28th Annual Int'l Symp. on Computer Architecture (ISCA '01), 30 Jun-4 Jul 2001, IEEE, pp.110-119.

[ZHO2000] Zhou, J., "Further Analysis of the Internet Key Exchange Protocol," *Computer Communications* 23, Nov. 2000, Elsevier Ltd., pp.1606-1612.

[ZIB2003] Zibin, D. and Ning, Z., "FPGA Implementation of SHA-1 Algorithm," *Proc. 5th Int'l Conference on ASIC,* 21-24 Oct. 2003, IEEE, Vol. 2, pp. 1321-1324.

R.2. Books, General Papers and other Resources

[BAC1997] Baccala, B., ed., "Connected: An Internet Encyclopedia," Freesoft.org, April 1997. Available: http://www.freesoft.org/CIE/index.htm.

[DEV2003] "Ultra High Speed AES (Rijndael) Crypto Processor," DeverSYS.com, 2003. Available: http://deversys.com/?action=project&id=43.

[DRA2004] Drange, G., "Random Number Generator Library," Norway, Sept. 30, 2004. Available: http://www.opencores.org/projects.cgi/web/rng_lib/overview.

[ERF2003] Erfani, S., "Security Management System and Method," US Patent 6,542,993 B1, Apr. 1, 2003.

[FAH2005] Fahandezh, M., "A Framework for IPSec Functional Architecture," MASc Thesis, ECE, Faculty of Grad. Studies and Research, U. Windsor, 2005.

[FIPS197] Federal Information Processing Standards Publication 197, "Announcing the Advanced Encryption Standard (AES)," Nov. 26, 2001. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[HAA1999] Haahr, M., "Introduction to Randomness and Random Numbers," Random.org, June 1999. Available: http://dirk.eddelbuettel.com/code/random/random-essay.pdf.

[HP12006] Hellakalek, P., "Random Number Generators – the pLab Project – Generators," University of Salzburg, Austria, 2006. Available: http://random.mat.sbg.ac.at/generators/.

[HUR2002] Huriadi, A., "AES Core design with Alliance," Institute of Technology, Bandung, Indonesia, 2002. Available: http://ic.ee.itb.ac.id/%7Ehuriadi/AES/.

[JAC1998] Jacobson, I, Booch, G, and Rumbaugh, J, *The Unified Software Development Process,* Addison Wesley Longman, 1998, ISBN 0-201-57169-2.

[KEN1994] Kent, S, "IPSEC SMIB", e-mail to ipsec@ans.net, Aug.10, 1994. Available: http://www.sandelman.ottawa.on.ca/ipsec/1994/08/msg00139.html.

[KER1988] Kernighan, B., and Ritchie, D., *The C Programming Language*, 2nd ed., AT&T Bell Laboratories, Murray Hill, NJ, Prentice Hall PTR, Englewood Cliffs, NJ, 1988, ISBN 0-13-110362-8 (pbk.), ISBN 0-13-110370-9.

[KHA2006] Khalid, M., "Reconfigurable Computing Systems: Challenges and Opportunites," *Reconfigurable Computing Course*, ECE Dept., University of Windsor, Jan. 2006, First lecture, p. 8.

[KLE2003] de Klein, R., "Serial library for C++," The Code Project, Nov. 13, 2003. Available: http://www.codeproject.com/system/serial.asp.

[KUR2000] Kurose, J., Ross, K., *Computer Networking – A Top-Down Approach Featuring the Internet*, 1st ed., Jul. 10, 2000, 3rd ed., 2005, Addison-Wesley Publishing Company, ISBN 0-20-147711-4 (1st ed.), 0-321-22735-2 (3rd ed.).

[MAL2002] Malik, D., *C++ Programming: Program Design Including Data Structures*, Course Technology, div. of Thomson Learning, Inc., Boston, MA, 2002, ISBN 0-619-03569-2.

[MAR1995] Marsaglia, G., "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness," Florida State University, 1995. Available: http://www.stat.fsu.edu/pub/diehard/.

[MAR2003] Martin, T., "Lectures, ECE 4514 Digital Design II," Virginia Tech, Fall 2003. Available: http://www.ece.vt.edu/tlmartin/ece4514/lectures/index.html.

[MAT2006] "SELEX Sensors and Airborne Systems Streamlines FPGA Development with MathWorks and Xilinx Tools," *User Stories*, The MathWorks, Inc., 2006. Available: http://www.mathworks.com/company/user_stories/userstory10995.html?by=company.

[MRO2000] Mroczkowski, P., "Implementation of the Block Cipher Rijndael using Altera FPGA," 2000. Available: http://csrc.nist.gov/encryption/aes/round2/comments/20000510-pmroczkowski.pdf

[OGA2004] Ogawa, J., "Living in the Product Development 'Valley of Death'," *FPGA and Structured ASIC Journal*, ABG Solutions Marketing, Altera Corp., Nov. 23, 2004. Available: http://www.fpgajournal.com/articles/20041123_altera.htm.

[PER2000] Perlmutter, B., with Zarkower, J., *Virtual Private Networking – A View from the Trenches*, Prentice Hall, Upper Saddle River, NJ, 2000, ISBN 0-13-020335-1.

[SAT2004] Satyanarayana, H, "AES128 [Implementation]." Available: http://www.opencores.org/projects.cgi/web/aes_crypto_core/overview.

[SIM1999] Simpson, W., "IKE/ISAKMP considered harmful," USENIX, http://www.usenix.org/publications/login/1999-12/features/harmful.html, 1999

[STA2003] Stallings, W., *Network Security Essentials: Applications and Standards*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2003 , ISBN 0-13-035128-8.

[STY2006] Styer, E., "JavaScript AES Example," Eastern Kentucky University, 2006. Available: http://www.cs.eku.edu/faculty/styer/460/Encrypt/JS-AES.html.

[TEKTDS] User Manual, TDS1000- and TDS2000-Series Digital Storage Oscilloscope Tektronix, Beaverton, OR, Tektronix Part No. 071-1064-00.

[TKA2002] Tkacik, T., "A Hardware Random Number Generator," CHES2002, Rev 0.1, Motorola, 2002. Available: http://ece.gmu.edu/crypto/ches02/talks_files/Tkacik.pdf.

[USS2002] Usselmann, Rudolf, "Advanced Encryption Standard / Rijndael IP Core," Rev. 1.1, Nov. 12, 2002. Available: http://www.opencores.org/projects.cgi/web/aes_core/overview.

[UTXILT] Xilinx, "Xilinx Training Labs at University of Toronto," Nov 2003. Available: http://www.eecg.toronto.edu/~pc/courses/edk/.

[VILL2005] Villar, J, "A SystemC/Verilog Random Number Generator," Universidad Rey Juan Carlos, Spain, 2005. Available: http://www.opencores.org/projects.cgi/web/systemc_rng/overview.

[VILL22005] Villar, J, "128/192 AES [Using System C]," Universidad Rey Juan Carlos, Spain, 2005. Available: http://www.opencores.org/projects.cgi/web/systemcaes/overview.

[WAL2007] Walton, J., "A Survey of Pseudo Random Number Generators," The Code Project, Jan. 2007. Available: http://www.codeproject.com/useritems/PRNG.asp.

[XILEST] Xilinx, "Embedded System Tools Reference Manual, Embedded Development Kit, EDK 8.2i," UG111 (v6.0), June 23, 2006.

[XILIDT] Xilinx, "ISE 8.2 In-Depth Tutorial," Available: http://direct.xilinx.com/direct/ise8_tutorials/ise8tut.pdf.

[XILIPTS3] Xilinx, "Import Peripheral Tutorial which targets the Spartan-3 devices 7.1." Available: http://www.xilinx.com/support/techsup/tutorials/edk_tutorials.htm.

[XILML403T] Xilinx, "EDK 8.2 PowerPC Tutorial in Virtex-4," WT001 (v4.0) January 30, 2006. Available: http://www.xilinx.com/support/techsup/tutorials/EDK_82_PPC_Tutorial.pdf.

[XILOPBIP2H] Xilinx, "OPB IPIF (v2.00h)," DS414, Apr. 6, 2005.

[XILPKG] Xilinx, "Device Package User Guide," UG112 (v2.0) May 31, 2006. Available: http://www.xilinx.com/bvdocs/userguides/ug112.pdf.

[XILQST] Xilinx, "ISE 8.2i Quick Start Tutorial" Available: http://www.xilinx.com/support/sw_manuals/xilinx82/download/.

[XILRIV] Rivoallon, F., "Achieving Breakthrough Performance in Virtex-4 FPGAs," WP218 (v1.4), May 19, 2006. Available: http://direct.xilinx.com/bvdocs/whitepapers/wp218.pdf.

[XILTMAT] Xilinx, "Teaching Materials," 1994-2007. Available: http://www.xilinx.com/univ/teaching_material.htm.

[XILUG80] Xilinx, "ML401/ML402/ML403 Evaluation Platform User Guide," UG080 (v2.5) May 24, 2006. Available: http://direct.xilinx.com/bvdocs/userguides/ug080.pdf.

[XILUT2003] Xilinx Processor IP Team, "EDK Training at University of Toronto," Nov 2003. Available: http://www.eecg.toronto.edu/~pc/courses/432/2004/handouts/TrainingLecture.pdf , via http://www.xilinx.com/univ/downld_partnerteaching.htm, via http://www.xilinx.com/univ/teaching_material.htm.

[XILV2DS] Xilinx, "Virtex-II Platform FPGAs: Complete Data Sheet," DS031 (v3.4) Mar. 1, 2005. Available: http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[XILV4DS] "Virtex-4 Family Overview," DS112 (v1.5) Feb. 10, 2006. Available: http://direct.xilinx.com/bvdocs/publications/ds112.pdf.

[XILXST] Xilinx, "XST User Guide," 8.2i. Available: http://www.xilinx.com/support/sw_manuals/xilinx82/download/.

R.3. RFCs

Note: all RFCs are available using the following format: http://www.ietf.org/rfc/rfcNNNN.txt , where "NNNN" is the four-digit RFC number – insert leading zeroes to make four digits if the number is shorter.

[IPS2005] "IPsec Charter [and list of IPsec RFCs]," Apr. 2005, Available: ftp://ftp.ietf.org/ietf/ipsec/ipsec-charter.txt.

[RFC0791] "Internet Protocol," Information Sciences Institute, University of Southern California, Sept. 1981

[RFC1700] Reynolds, J. and Postel, J., "Assigned Numbers," Oct. 1994.

[RFC1750] Eastlake, D., Crocker, S., and Schiller, J., "Randomness Recommendations for Security," Dec. 1994.

[RFC2401] Kent, S., and Atkinson, R., "Security Architecture for the Internet Protocol," Nov. 1998.

[RFC2402] Kent, S., and Atkinson, R., "IP Authentication Header," Nov. 1998.

[RFC2406] Kent, S., and Atkinson, R., "IP Encapsulating Security Payload (ESP)," Nov. 1998.

[RFC2408] Maughan, D., Schertler, M., Schneider, M., and Turner, J., "Internet Security Association and Key Management Protocol (ISAKMP)," Nov. 1998.

[RFC2409] Harkins, D., and Carrel, D., "The Internet Key Exchange (IKE)," Nov 1998.

[RFC2522] Karn, P., and Simpson, W., "Photuris: Session-Key Management Protocol," March 1999.

[RFC3159] McCloghrie, K. *et al.*, "Structure of Policy Provisioning Information (SPPI)," IETF RFC 3159, Aug. 2001.

R.4. Websites

[FSWAN] The Free Secure Wide-Area Network (Free S/WAN) project, an open-source software implementation of IPSec, http://www.freeswan.org/.

[IETF] The Internet Engineering Task Force, http://www.ietf.org.

[ITEF-IPSEC] IP Security Protocol Working Group (ipsec) FTP site, ftp://ftp.ietf.org/ietf/ipsec/.

[OPENCORES] "Open Cores," http://www.opencores.org.

[PGPI] "Pretty Good Privacy," http://www.pgpi.org.

[USPTO] The US Patent and Trademark Office, http://www.uspto.gov.

[WIKIP] Wikipedia, http://www.wikipedia.org; English home page, http://en.wikipedia.org.

VITA AUCTORIS

NAME:	James H. L. Wiebe
PLACE OF BIRTH:	London, Ontario
YEAR OF BIRTH:	1963
EDUCATION:	Cameron Heights Collegiate Institute Kitchener, Ontario 1978-1982
	University of Waterloo Waterloo, Ontario 1982-1987 B.A.Sc.
	University of Windsor Windsor, Ontario 2004-2007 M.A.Sc.