DESIGN AND IMPLEMENTATION OF AN OPEN SECURITY ARCHITECTURE FOR A SOFTWARE-BASED SECURITY MODULE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

KAAN KAYNAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

MAY 2009

DESIGN AND IMPLEMENTATION OF AN
OPEN SECURITY ARCHITECTURE
FOR A SOFTWARE-BASED SECURITY MODULE

submitted by **KAAN KAYNAR** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen ____________________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit ____________________
Head of Department, **Computer Engineering**

Dr. Atilla Özgit ____________________
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Payidar Genç ____________________
Computer Engineering Dept., METU

Dr. Atilla Özgit ____________________
Computer Engineering Dept., METU

Assoc. Prof. Dr. Cem Bozşahin ____________________
Computer Engineering Dept., METU

Dr. Onur Tolga Şehitoğlu ____________________
Computer Engineering Dept., METU

MSc. Mert Özarar ____________________
Computer Engineering Dept., METU

Date: _________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Kaan Kaynar

Signature :

ABSTRACT

DESIGN AND IMPLEMENTATION OF AN
OPEN SECURITY ARCHITECTURE
FOR A SOFTWARE-BASED SECURITY MODULE

Kaynar, Kaan
M.S., Department of Computer Engineering
Supervisor : Dr. Attila Özgit

May 2009, 165 pages

Main purpose of this thesis work is to design a comprehensive and open security architecture whose desired parts could be realized on a general-purpose embedded computer without any special cryptography hardware. The architecture provides security mechanisms that implement known cryptography techniques, operations of some famous network security protocols and appropriate system security methods. Consequently, a server machine may offload a substantial part of its security processing tasks to an embedded computer realizing the architecture. The mechanisms provided can be accessed by a server machine using a client-side API and via a secure protocol which provides message integrity and peer authentication. To demonstrate the practicability of the security architecture, a set of its security mechanisms was realized on an embedded PC/104-plus computer. A server machine was connected to and requested mechanisms from the embedded computer over the Ethernet network interface. Four types of performance parameters were measured. They are; number of executions of a symmetric encryption method by the embedded computer per second, number of executions of a public-key signing method by the embedded computer per second, footprint

of the implementation on the embedded computer memory, and the embedded computer CPU power utilized by the implementation. Apart from various security mechanisms and the secure protocol via which they can be accessed, the architecture defines a reliable software-based method for protection and storage of secret information belonging to clients.

Keywords: Security Architecture, Open Architecture, Security Module, Software-based

ÖZ

YAZILIM TEMELLİ GÜVENLİK MODÜLÜ İÇİN
AÇIK BİR GÜVENLİK MİMARİSİ
TASARIMI VE UYGULAMASI

Kaynar, Kaan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Attila Özgit

Mayıs 2009, 165 sayfa

Bu tez çalışmasının temel amacı, kapsamlı ve arzu edilen kısımları hiçbir özel kriptoğrafi donanımı bulunmayan genel amaçlı bir gömülü bilgisayar üzerinde gerçeklenebilecek açık bir güvenlik mimarisi tasarlamaktır. Mimari, bilinen kriptoğrafi tekniklerini, bazı tanınmış ağ güvenlik protokollerinin operasyonlarını ve uygun sistem güvenliği metotlarını yerine getiren güvenlik mekanizmalarını sağlar. Dolayısıyla, bir sunucu makina güvenlik işlem görevlerinin önemli bir kısmını, bu mimariyi gerçekleyen bir gömülü bilgisayara yükleyebilir. Sağlanan mekanizmalara, bir sunucu makina tarafından, alıcı tarafındaki Uygulama Programlama Arayüzünü kullanarak ve mesaj bütünlüğü ve kaynak doğrulaması sağlayan bir güvenli protokol vasıtasıyla erişilebilir. Güvenlik mimarisinin uygulanabilirliğini göstermek için, güvenlik mekanizmalarının bir bölümü bir gömülü PC/104-plus bilgisayarı üzerinde gerçeklenmiştir. Bir sunucu makinası, Ethernet ağ arayüzü üzerinden gömülü bilgisayara bağlanmış ve ondan mekanizmalar talep etmiştir. Dört çeşit performans parametresi ölçülmüştür. Bunlar; bir simetrik şifreleme metodunun gömülü bilgisayar tarafından bir saniye başına icra edilme sayısı, bir açık-anahtar imzalama metodunun gömülü bilgisayar

tarafından bir saniye başına icra edilme sayısı, uygulamanın gömülü bilgisayarın hafızasında kapladığı alan, ve uygulama tarafından kullanılan gömülü bilgisayarın Merkezi İşlem Ünitesi gücüdür. Çeşitli güvenlik mekanizmaları ve bunlara erişme vasıtası olan güvenli protokol haricinde, mimari, alıcılara ait gizli bilgilerin korunması ve depolanması için, güvenilir yazılım temelli bir yöntem tanımlamaktadır.

Anahtar Kelimeler: Güvenlik Mimarisi, Açık Mimari, Güvenlik Modülü, Yazılım Temelli

TABLE OF CONTENTS

ABSTRACT iv
ÖZ vi
TABLE OF CONTENTS viii
LIST OF TABLES xiv
LIST OF FIGURES xvi
LIST OF ABBREVIATIONS xix
CHAPTERS
1. INTRODUCTION 1
2. LITERATURE SURVEY 3
2.1. Commercial Security Modules and Security Architectures 3
2.1.1. IBM Common Cryptographic Architecture (CCA) 3
2.1.2. IBM PCI Cryptographic Coprocessor (IBM 4758) 5
2.1.3. IBM PCI-X Cryptographic Coprocessor (IBM 4764) 6
2.1.4. nCipher Security World Key Management Architecture 7
2.1.5. nCipher Secure Execution Engine 8
2.1.6. nCipher miniHSM 9
2.1.7. nCipher netHSM 10
2.1.8. netHSM Technical Architecture 11
2.1.9. SafeNet Security Modules - Luna PCI 13
2.1.10. SafeNet Security Modules - Luna SA 14
2.2. Advantech PCM-3370F-M0A1E PC/104-plus CPU Board 14
3. PROPOSED SECURITY ARCHITECTURE 17
3.1. Basic Offloader Mechanisms 24
3.1.1. Symmetric_Encryption_Decryption Basic Offloader Mechanism 24
3.1.2. Asymmetric_Encryption_Decryption Basic Offloader Mechanism 25

3.1.3. Session_Key_Calculation Basic Offloader Mechanism............26
3.1.4. Message_Authentication_Code_Operations Basic Offloader Mechanism..27
3.1.5. Message_Digest_Operations Basic Offloader Mechanism.......28
3.1.6. Digital_Signature_Calculation Basic Offloader Mechanism....30
3.2. Administrational Offloader Mechanisms..31
3.2.1. Cryptographic_Keys_Parameters_Generation Administrational Offloader Mechanism..31
3.2.2. Time_Time_Window_Adjustment Administrational Offloader Mechanism..32
3.2.3. Cryptographic_Keys_Parameters_Entry_Erasure_Backup Administrational Offloader Mechanism..................................33
3.2.4. User_Role_Management Administrational Offloader Mechanism..34
3.2.5. Module_Code_Status_Receiving_Loading Administrational Offloader Mechanism..34
3.3. Cryptographic Keys, Parameters, Public-Key Certificates and CRLs Storage..35
3.4. SSLv3 Protocol Offloader Mechanisms..39
3.4.1. SSL_Record_Preparation SSLv3 Offloader Mechanism..........39
3.4.2. SSL_Record_Verification SSLv3 Offloader Mechanism.........40
3.4.3. Random_Value_Generation SSLv3 Offloader Mechanism......40
3.4.4. Server_Certificate_Retrieval SSLv3 Offloader Mechanism.....41
3.4.5. SSL_Certificate_Verification SSLv3 Offloader Mechanism....42
3.4.6. Server_Key_Exchange_Preparation SSLv3 Offloader Mechanism..42
3.4.7. Server_Key_Exchange_Verification SSLv3 Offloader Mechanism..43
3.4.8. Client_Certificate_Retrieval SSLv3 Offloader Mechanism......44
3.4.9. Client_Key_Exchange_Preparation SSLv3 Offloader Mechanism..45

3.4.10. Client_Key_Exchange_Verification SSLv3 Offloader Mechanism 46
3.4.11. Certificate_Verify_Preparation SSLv3 Offloader Mechanism 48
3.4.12. Certificate_Verify_Verification SSLv3 Offloader Mechanism 48
3.4.13. Finished_Preparation SSLv3 Offloader Mechanism 49
3.4.14. Finished_Verification SSLv3 Offloader Mechanism 49
3.4.15. Pre-master_Secret_Computation SSLv3 Offloader Mechanism 50
3.4.16. Master_Secret_And_Keys_Computation SSLv3 Offloader Mechanism 51
3.5. SET Protocol Offloader Mechanisms 51
3.5.1. Nonce_Generation SET Offloader Mechanism 52
3.5.2. Initiate_Response_Preparation SET Offloader Mechanism 52
3.5.3. Initiate_Response_Verification SET Offloader Mechanism 53
3.5.4. Purchase_Request_Preparation SET Offloader Mechanism 54
3.5.5. Purchase_Request_Verification SET Offloader Mechanism 55
3.5.6. Purchase_Response_Preparation SET Offloader Mechanism 55
3.5.7. Purchase_Response_Verification SET Offloader Mechanism 56
3.5.8. Authorization_Request_Preparation SET Offloader Mechanism 57
3.5.9. Authorization_Request_Verification SET Offloader Mechanism 58
3.5.10. Authorization_Response_Preparation SET Offloader Mechanism 59
3.5.11. Authorization_Response_Verification SET Offloader Mechanism 60
3.5.12. Capture_Request_Preparation SET Offloader Mechanism 60
3.5.13. Capture_Request_Verification SET Offloader Mechanism 61
3.5.14. Capture_Response_Preparation SET Offloader Mechanism 62

3.5.15. Capture_Response_Verification SET Offloader Mechanism..63
3.6. Kerberos Protocol Version 5 Offloader Mechanisms 64
3.6.1. Nonce_Generation Kerberos Offloader Mechanism.................64
3.6.2. TGT_Preparation Kerberos Offloader Mechanism...................64
3.6.3. TGT_Response_Verification Kerberos Offloader Mechanism .65
3.6.4. SGT_Authenticator_Preparation Kerberos Offloader Mechanism..66
3.6.5. TGT_Verification Kerberos Offloader Mechanism..................67
3.6.6. SGT_Preparation Kerberos Offloader Mechanism...................67
3.6.7. SGT_Response_Verification Kerberos Offloader Mechanism .68
3.6.8. Service_Authenticator_Preparation Kerberos Offloader Mechanism..69
3.6.9. SGT_Verification Kerberos Offloader Mechanism..................69
3.6.10. Service_Response_Preparation Kerberos Offloader Mechanism...70
3.6.11. Service_Response_Verification Kerberos Offloader Mechanism...71
3.7. X.509 Certification Operations Offloader Mechanisms 71
3.7.1. Certificate_Sign_Request_Preparation X.509 Offloader Mechanism..72
3.7.2. Certificate_Sign_Request_Verification X.509 Offloader Mechanism..72
3.7.3. Certificate_Preparation X.509 Offloader Mechanism 73
3.7.4. Certificate_Revocation_List_Preparation X.509 Offloader Mechanism..74
3.7.5. Certificate_CRL_Retrieval X.509 Offloader Mechanism.........74
3.7.6. Certificate_CRL_Verification X.509 Offloader Mechanism 75
3.8. System Security Offloader Mechanisms 76
3.8.1. User_Password_Records_Storing System Security Offloader Mechanism..78

3.8.2. Entered_Password_Verification System Security Offloader Mechanism 79
3.8.3. Access_Rights_Entries_Storing System Security Offloader Mechanism 79
3.8.4. Attempted_Access_Verification System Security Offloader Mechanism 81
3.8.5. Access_Rights_Entries_Retrieval System Security Offloader Mechanism 81
3.9. Offloader Mechanism Access Controller 82
3.10. Communication Infrastructure 84
4. REFERENCE IMPLEMENTATION 87
4.1. Reference Algorithms for Basic Offloader Mechanisms 88
4.1.1. Symmetric_Encryption_Decryption Basic Offloader Mechanism 88
4.1.2. Asymmetric_Encryption_Decryption Basic Offloader Mechanism 90
4.1.3. Session_Key_Calculation Basic Offloader Mechanism 92
4.1.4. Message_Authentication_Code_Operations Basic Offloader Mechanism 94
4.1.5. Message_Digest_Operations Basic Offloader Mechanism 97
4.1.6. Digital_Signature_Calculation Basic Offloader Mechanism 102
4.2. Offloader Mechanism Access Controller Server Program 104
4.3. Communication Infrastructure - SSL Library 107
4.4. Embedded Linux OS 108
4.5. Embedded Computer 111
4.6. Implementation Issues About Client-side API Functions 112
4.7. Performance Tests 114
5. SUMMARY CONCLUSION 129
REFERENCES 136
APPENDICES
A. COMMUNICATIONS PROTOCOL MESSAGE FORMATS 140

B. RESULTS OF PERFORMANCE MEASUREMENTS 156

LIST OF TABLES

TABLES

Table 1 Performance Measurements Results - Disk Space Usage 126
Table 2 Symmetric_Encryption_Decryption Offloader Mechanism Execution Request Message 141
Table 3 Symmetric_Encryption_Decryption Offloader Mechanism Execution Response Message 142
Table 4 Asymmetric_Encryption_Decryption Offloader Mechanism Execution Request Message 143
Table 5 Asymmetric_Encryption_Decryption Offloader Mechanism Execution Response Message 144
Table 6 Session_Key_Calculation Offloader Mechanism Execution Request Message 145
Table 7 Session_Key_Calculation Offloader Mechanism Execution Response Message 146
Table 8 Message_Authentication_Code_Operations Offloader Mechanism Execution Request Message 147
Table 9 Message_Authentication_Code_Operations Offloader Mechanism Execution Response Message 148
Table 10 Message_Digest_Operations Offloader Mechanism Execution Request Message 149
Table 11 Message_Digest_Operations Offloader Mechanism Execution Response Message 150
Table 12 Digital_Signature_Calculation Offloader Mechanism Execution Request Message 150
Table 13 Digital_Signature_Calculation Offloader Mechanism Execution Response Message 150

Table 14 Cryptographic_Keys_Parameters_Creation Offloader Mechanism Execution Request Message 151
Table 15 Cryptographic_Keys_Parameters_Creation Offloader Mechanism Execution Response Message 151
Table 16 Time_Time_Window_Adjustment Offloader Mechanism Execution Request Message 151
Table 17 Time_Time_Window_Adjustment Offloader Mechanism Execution Response Message 152
Table 18 Key_Entry_Erasure_Backup Offloader Mechanism Execution Request Message 153
Table 19 Key_Entry_Erasure_Backup Offloader Mechanism Execution Response Message 154
Table 20 User_Role_Administration Offloader Mechanism Execution Request Message 154
Table 21 User_Role_Administration Offloader Mechanism Execution Response Message 154
Table 22 Module_Code_Status_Receiving_Loading Offloader Mechanism Execution Request Message 155
Table 23 Module_Code_Status_Receiving_Loading Offloader Mechanism Execution Response Message 155
Table 24 Performance Measurements Results - Symmetric Encryption, 100-bytes of Input Data 163
Table 25 Performance Measurements Results - Signing, 100-bytes of Input Data 163
Table 26 Performance Measurements Results - Memory Space Usage 164
Table 27 Performance Measurements Results - CPU Power Consumption 165

LIST OF FIGURES

FIGURES

Figure 1 Advantech PCM-3370F PC/104-plus CPU Module 15
Figure 2 Typical Application Context 18
Figure 3 Security Services Needed by Host Systems and Host Security Servers .. 20
Figure 4 Security Services Needed by a Security Module 22
Figure 5 Access to Offloader Mechanisms via the Communications Protocol 23
Figure 6 Secret Keys and Parameters Storage Format 36
Figure 7 Overall Security Architecture (Initial) 38
Figure 8 Peer Authentication, Message Exchange and Access Control 85
Figure 9 Overall Security Architecture 86
Figure 10 Performance Measurements Results - DES Symmetric Encryption, 100-bytes of Input Data 119
Figure 11 Performance Measurements Results - AES Symmetric Encryption, 100-bytes of Input Data 120
Figure 12 Performance Measurements Results - RSA 1024-bit Signing, 100-bytes of Input Data 120
Figure 13 Performance Measurements Results - RSA 2048-bit Signing, 100-bytes of Input Data 121
Figure 14 Performance Measurements Results - Physical Memory Space Usage by the OS 121
Figure 15 Performance Measurements Results - Physical Memory Space Usage by the Application Software 122
Figure 16 Performance Measurements Results - Physical Memory Space Usage Sum 122
Figure 17 Performance Measurements Results - Virtual Memory Space Usage by the OS 123

Figure 18 Performance Measurements Results - Virtual Memory Space Usage by the Application Software 123
Figure 19 Performance Measurements Results - Virtual Memory Space Usage Sum 124
Figure 20 Performance Measurements Results - CPU Power Consumption by the OS 124
Figure 21 Performance Measurements Results - CPU Power Consumption by the Application Software 125
Figure 22 Performance Measurements Results - CPU Power Consumption Sum 125
Figure 23 Performance Measurements Results - DES Symmetric Encryption, 20-bytes of Input Data 157
Figure 24 Performance Measurements Results - AES Symmetric Encryption, 20-bytes of Input Data 157
Figure 25 Performance Measurements Results - DES Symmetric Encryption, 200-bytes of Input Data 158
Figure 26 Performance Measurements Results - AES Symmetric Encryption, 200-bytes of Input Data 158
Figure 27 Performance Measurements Results - DES Symmetric Encryption, 500-bytes of Input Data 159
Figure 28 Performance Measurements Results - AES Symmetric Encryption, 500-bytes of Input Data 159
Figure 29 Performance Measurements Results - DES Symmetric Encryption, 1000-bytes of Input Data 160
Figure 30 Performance Measurements Results - AES Symmetric Encryption, 1000-bytes of Input Data 160
Figure 31 Performance Measurements Results - RSA 1024-bit Signing, 20-bytes of Input Data 161
Figure 32 Performance Measurements Results - RSA 2048-bit Signing, 20-bytes of Input Data 161

Figure 33 Performance Measurements Results - RSA 1024-bit Signing, 50-bytes of Input Data....... 162
Figure 34 Performance Measurements Results - RSA 2048-bit Signing, 200-bytes of Input Data 162

LIST OF ABBREVIATIONS

ACL	Access Control List
AES	Advanced Encryption Standard
AGP	Accelerated Graphics Port
AH	Authentication Header
AIX	Advanced Interactive Executive
ANSI	American National Standards Institute
API	Application Programming Interface
ARC4	Alleged RC4
AS	Authentication Server
AT	Advanced Technology
ATA	Advanced Technology Attachment
ATAPI	Advanced Technology Attachment Packet Interface
ATX	Advanced Technology eXtended
BIOS	Basic Input/Output System
BSP	Board Support Package
CA	Certification Authority
CAST	Carlisle Adams Stafford Tavares
CBC	Cipher Block Chaining
CCA	Common Cryptographic Architecture
CFB	Cipher Feedback
CFC	Compact Flash Card
CFI	Common Flash Interface
CMAC	Cipher-based Message Authentication Code
CML2	Configuration Menu Language 2
CPU	Central Processing Unit

CRAMFS	Compressed ROM File System
CREN	Corporation for Research & Educational Networking
CRL	Certificate Revocation List
CSR	Certificate Signing Request
DAA	Data Authentication Algorithm
DAC	Data Authentication Code
DBMS	Database Management System
DER	Distinguished Encoding Rules
DES	Data Encryption Standard
DH	Diffie-Hellman
DHCP	Dynamic Host Configuration Protocol
DIN	German Institute for Standardization
DMA	Direct Memory Access
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
e-Commerce	Electronic Commerce
e-mail	Electronic Mail
EBX	Embedded Board eXpandable
ECB	Electronic Codebook
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EIDE	Enhanced Integrated Drive Electronics
EMV	Europay MasterCard VISA
ESP	Encapsulating Security Payload
ETRI	Electronics and Telecommunications Research Institute
EXT3	Third Extended Filesystem

FAQ	Frequently Asked Questions
FIPS PUB	Federal Information Processing Standards Publications
GB	GigaByte
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GUI	Graphical User Interface
HMAC	keyed-Hash Message Authentication Code
HP UX	Hewlett Packard UniX
HSM	Hardware Security Module
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
ICV	Integrity Check Value
ID	Identifier
IDE	Integrated Drive Electronics
IDEA	International Data Encryption Algorithm
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPSec	IP Security
IRQ	Interrupt Request
ISA	Industry Standard Architecture
ISAKMP	Internet Security Association and Key Management Protocol
ISO	International Organization for Standardization
IV	Initialization Vector
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
JFFS	Journaling Flash Filesystem

JFS	Journaled File System
KB	KiloByte
KO	Not OK
LAN	Local Area Network
LCD	Liquid Crystal Display
LPT	Line Print Terminal
MAC	Message Authentication Code
MB	MegaByte
MD5	Message-Digest algorithm 5
MIME	Multipurpose Internet Mail Extensions
MIT	Massachusetts Institute of Technology
NFS	Network File System
NIST	National Institute of Standards and Technology
OEM	Original Equipment Manufacturer
OFB	Output Feedback
OI	Order Information
OIMD	Order Information Message Digest
OS	Operating System
PCBC	Propagating Cipher Block Chaining
PCI	Peripherial Component Interface
PCI-X	PCI eXtended
PED	PIN Entry Device
PEM	Privacy Enhanced Mail
PI	Payment Information
PIMD	Payment Information Message Digest

PIN	Personal Identification Number
PKCS	Public-key Cryptography Standards
PKDS	Public-key Distribution Scheme
PKI	Public-key Infrastructure
POSIX	Portable Operating System Interface
QPD	Qplus Package Descriptor
RAM	Random Access Memory
RC4	Rivest Cipher 4
RFC	Request for Comments
RIPEMD-160	RACE Integrity Primitives Evaluation Message Digest
RoHS	Restriction of Hazardous Substances Directive
ROM	Read Only Memory
ROMFS	ROM File System
RPM	RedHat Package Manager
RS232	Recommended Standard 232
RSA	Rivest Shamir Adleman
S/MIME	Secure/Multipurpose Internet Mail Extension
SA	Security Association
SAFE	Signatures and Authentication for Everyone
SBC	Single Board Computer
SDRAM	Synchronous Dynamic Random Access Memory
SEE	Secure Execution Engine
SET	Secure Electronic Transaction
SGT	Service-granting Ticket
SHA-1	Secure Hash Algorithm
SM	Security Module
SMK	Storage Master Key

SMTP	Simple Mail Transfer Protocol
SODIMM	Small Outline Dual In-line Memory Module
SPD	Security Policy Database
SPI	Security Parameters Index
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TDES	Triple DES
TFTP	Trivial File Transfer Protocol
TGS	Ticket-granting Server
TGT	Ticket-granting Ticket
TLS	Transport Layer Security
U	Rack Unit
UDX	User Defined Extensions
USB	Universal Serial Bus
USC-ISI	University of Southern California Information Sciences Institute
VGA	Video Graphics Array
WAN	Wide Area Network
WEP	Wired Equivalent Privacy
XOR	Exclusive OR
ZKA	German Customs Investigation Bureau

CHAPTER 1

INTRODUCTION

As the connectivity to the Internet has become a requirement for most organizations, securing their workstations and servers and data they transmit over the Internet becomes a necessity. Usually *dedicated security servers* are used in a local network of an organization to perform some widely adopted cryptography and security operations and protocols to counter against various types of attacks. These dedicated servers generally require high processing power and other resources to perform many CPU-demanding security-related operations as fast as possible and return the results to the requesting workstations and servers without much delay. Therefore, dedicated security servers usually need access to a separate security module and use cryptography and security mechanisms provided by that module. Besides securely performing cryptography and security operations for dedicated server machines to increase their performance, security modules store and protect keys and other secret data used in these operations on behalf of their owners.

In this thesis work a comprehensive security architecture, which includes various cryptography and security mechanisms responding to the needs of different application areas, and at the same time which provides secure storage for various kinds of secret keys, parameters and data belonging to users of the local network, is designed. Desired parts of the designed security architecture could be realized to build a security module according to the needs of a specific application area.

Some existing commercial security modules and security architectures, based on which they were built, are examined in Chapter 2 "Literature Survey" of this document. As can be seen in that chapter, a few open commercial security

architectures exist. The security architecture in this work is designed to be as extensive as possible and yet is open.

The security architecture designed in this work is described in detail in Chapter 3 “Proposed Security Architecture” of this document. It is completely a software architecture without involving any hardware requirements.

To prove the practicability of the designed security architecture, some of the security mechanisms the architecture provides were realized on a PC/104-plus embedded computer to build a security module and critical performance parameters of the built security module were measured. Reference algorithms for the implemented security mechanisms, the methods used for evaluating the performance of the built security module and the results of performance evaluations are explained in detail in Chapter 4 “Reference Implementation” of this document.

Finally, in Chapter 5 “Summary Conclusion”, the writer’s comments on some strong and weak points of the designed security architecture, further improvements that could and should be done, and on whether the aim of this thesis work is reached are given.

CHAPTER 2

LITERATURE SURVEY

2.1. Commercial Security Modules and Security Architectures

This section gives detailed information about some famous commercial security modules and security architectures on which they are based. This information is useful in comparing the features of the security architecture designed with those of commercial ones, and also in comparing the results of performance tests, done on the built software-based security module, with the values of certain performance parameters given for some commercial security modules.

2.1.1. IBM Common Cryptographic Architecture (CCA)

IBM CCA provides a comprehensive, integrated family of services that use major capabilities of IBM Coprocessors. Release 2.x supports IBM Coprocessor models 002 and 023 on Windows NT, Windows 2000, and AIX. CCA provides the usual DES and RSA functions for data confidentiality and data integrity support. In addition, CCA features extensive support for distributed key management and many functions of special interest to the finance industry. CCA software has been independently reviewed and certified by the German ZKA industry organization for use in specific finance systems. **[5]**

CCA is capable of cryptographic-quality random number generation utilizing the Coprocessor hardware to seed a FIPS PUB 140-1 compliant random number generator. Keys are securely held in one of two ways:

- A modest number of (75 to 150 Coprocessor-generated) RSA private keys can be retained within the secure Coprocessor,

- An unlimited number of private keys and DES keys can be held external to the Coprocessor encrypted (wrapped) by the triple DES master key. **[5]**

The master key can be randomly generated within the Coprocessor, or can be inserted by two or more trusted individuals. With CCA control vector technology, extensive control of key usage in distributed cryptographic systems can be enabled. Cloning of a master key enables back-up or redundant Coprocessors to process encrypted local keys with the same master key. **[5]**

CCA provides SET services that support e-Commerce applications in merchant and acquirer credit card transaction processing. With Release 2.x, encrypted PIN block support is added consistent with the latest addition to the SET standards. PIN generation and verification services support several popular algorithms including customer-selected PIN options. A variety of PIN block formats are processed with support for secure re-encryption and re-formatting of PIN blocks. **[5]**

Digital signature generation and validation using RSA supports several different hash formatting methods including ISO-9796 and PKDS #1 standards. SHA-1 and MD5 hash algorithms are supported. Large blocks are hashed using the hardware SHA-1 hashing engine within the Coprocessors. The modular exponentiation hardware engine supports RSA keys up to 2048 bits in length. **[5]**

DES data encryption supports CBC and ANSI X9.23 last block padding rules. Triple-DES support comes in Release 2.x. Derived key support is available for dynamically creating DES keys from a key generating key in support of protocols such as used with EMV smart cards. MAC generation is supported using the DES algorithm and rules defined in the ANSI X9.9-1 and the ANSI X9.19 algorithms for single and double length keys. **[5]**

Through use of the UDX (User Defined Extensions) toolkit available under custom contract, customers or software vendors can implement their own applications for the Coprocessors or can extend CCA to support many other operations needed by their applications. IBM will issue each of them a unique identifier and certify their code-signing keys so that they can sign their own custom Coprocessor software. Developers can develop their software using conventional IBM or Microsoft C-language compilers and use the toolkit-provided debugging programs. Then, the software can be loaded to the Coprocessor. **[5] [6]**

The API for the CCA Release 2.x differs in certain details from the API for the Release 1.31. Application programs designed to work with CCA Release 1.31 may require modifications to run with Release 2.x. **[5]** Detailed information about CCA is given in the CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors. **[9]**

2.1.2. IBM PCI Cryptographic Coprocessor (IBM 4758)

The FIPS PUB 140-1 "Security Requirements for Cryptographic Modules" is the benchmark standard by which cryptographic implementations are evaluated. The IBM 4758 Model 002 is certified at level 4, the highest certification. The Model 023, which uses a different method of detecting physical penetration attacks, is certified at level 3. The evaluations cover the processing subsystem and its specialized cryptographic hardware, code loading, tamper detection and response mechanisms, and cryptographic algorithms: DES, triple-DES, RSA, DSS, and SHA-1. IBM 4758 Models 002 and 023 operate both on a 3.3-volt and 5-volt PCI bus and have batteries to power its tamper-sensing electronics when no system power is supplied. **[6]**

IBM supplies support program codes for two cryptographic APIs, PKCS #11 and IBM CCA API. PKCS #11, Cryptographic Token Interface Standard or Cryptoki, support program code provides access to the Coprocessor from AIX, Windows NT

and Windows 2000 platforms to perform MD2, MD5, SHA-1, RSA, DSS, DES, and triple-DES capabilities according to industry-standard API. Programs such as the Netscape security server can exploit security-afforded RSA private keys and offloading of host system processing through the use of one or more Coprocessors. IBM CCA API standard capabilities include PIN processing, Secure Electronic Transaction services, data encryption and hashing techniques, and RSA-based public-key cryptography. CCA and its API implementation can be extended through custom programming. **[6]**

Using PKI-based outbound authentication capabilities of the Coprocessor's control program, it can be securely administered even from remote locations. Auditors can inspect the Coprocessor's digitally signed status response to confirm that the Coprocessor remains untampered and running uniquely identified software. **[6]**

Models 002 and 023 support upto 175 1024-bit RSA private key operations per second. The Coprocessor also supports high-throughput bulk DES processing. DES encryption throughput of 15.3 MBytes/second has been measured on fast host systems. **[6]**

2.1.3. IBM PCI-X Cryptographic Coprocessor (IBM 4764)

IBM 4764 PCI-X Cryptographic Coprocessor provides a high-security, high-throughput cryptographic subsystem. The tamper-responding hardware is designed to qualify at the highest level under the stringent FIPS 140-2 standard. Specialized hardware performs DES, TDES, modular-exponentiation (for RSA, DSA) and SHA-1 cryptographic functions, relieving main processor from these tasks. The coprocessor design protects cryptographic keys and sensitive functions. **[7]**

IBM 4764 Coprocessor board has a PCI-X 1.0 and PCI 2.2 local bus compatible interface. The board holds a secured subsystem module, batteries for backup power, serial interface and 10/100 Ethernet. The securely encapsulated subsystem

contains an IBM PowerPC 405GPR, RAM, flash, and battery-powered memory, cryptographic-quality random number generator, specialized cryptographic hardware and full-duplex DMA communications. **[7]**

Secure code loading enables control program and application program loading and refreshes. IBM offers a Linux-based subsystem control program and an application program, which implements IBM Common Cryptographic Architecture (CCA). **[7]**

The Coprocessor is supported in IBM System z mainframes under z/OS, OS/390, Linux, in IBM System i servers, and in IBM System x servers running either 32-bit SUSE Linux Enterprise Server 9 or 32-bit Windows Server 2003 Standard Edition, to provide cryptographic services using the CCA cryptographic API. It is also supported in IBM System p servers running under the AIX operating system to provide cryptographic services using both the CCA cryptographic API and the PKCS #11 cryptographic API. **[7]**

2.1.4. nCipher Security World Key Management Architecture

Hardware security modules provide physical security of keys but do not completely address the issue of how keys are created, stored, managed and destroyed. This process of controlling key lifecycles is known as key management and requires software that interfaces between a security module and the external world. nCipher's Security World technology fills this need. **[8]**

Security World offers clear benefits to security architects and system managers in many respects. First of all, the approach places the emphasis on secure management of keys including secure creation, backup and recovery of keys. Keys are stored as encrypted and protected files outside the physical confines of a security module (in a smart card or hard disk). Hence, there is virtually unlimited key storage space available, and loss of a SM does not necessarily mean loss of the keys. When a key is transported outside a SM, the key data itself is encrypted

using strong (triple-DES) encryption, its Access Control List (ACL) is appended to the encrypted key data, and then confidentiality and integrity of the key data and its ACL are protected by one more encryption, and authentication (MAC). A key object prepared and stored in this way is known as a "key blob". This structure ensures that when keys are not physically protected within a module (when being created and managed), they are instead logically protected. **[8]**

Each key has its own ACL, which lists operations that can be performed on the key, and subjects that are entitled to request those operations. Access can be further controlled by requiring secret sharing which enables a key's fragments to be stored separately on tokens (smart cards or hard disks) so that 'k of n' key fragments are required in order to reconstitute the key. This ensures keys have a higher level of protection. **[8]**

nCipher's security modules provide a true hardware random number generator. This is used to create a truly random and externally unknowable module (root) key. Several nCipher security modules can be added to a network to be used together to provide consistent security management by configuring each module to use the same module key. Security World can be used in conjunction with the nCipher Secure Execution Engine technology to develop advanced custom security infrastructures. **[8]**

2.1.5. nCipher Secure Execution Engine

nCipher Secure Execution Engine (SEE) expands security beyond key management to include a secure environment in which only trusted, authenticated application code can run. By enabling secure execution of application code, which can act as a "Trusted Agent", data can be secured and access controlled. Trusted agents allow security architects to delegate authority to application codes, that they trust to act on their behalf, in server environments that are outside of their direct

control. Trusted agents (signed and authenticated) operate on nCipher SMs that have FIPS 140-1 validated hardware protection. **[10]**

In a large-scale system such as an automated trading network, trusted human operators cannot oversee every critical security instruction issued on the system. This creates a situation where the right to use cryptographic keys must be delegated to trusted applications, without requiring any human intervention. Fundamental problem common to many existing SMs is that although a security module provides secure storage for keys and the root key of the module, in most security architectures there is no way defined for the module to know the security-critical application with which it is communicating. SEE addresses this problem by transferring security-sensitive applications, previously running on a host server, into the secure confines of a SM. **[10]**

2.1.6. nCipher miniHSM

miniHSM is a FIPS 140-2 (a widely recognized security benchmark for cryptographic modules) level 3 validated platform for OEMs that require a hardware security module for performing cryptographic operations or protecting critical information. It is a secure, simple solution to provide data encryption, digital signing and strong authentication, and creates an opportunity for developers to leverage nCipher's unique application security and key management technology. **[11]**

miniHSM supports a wide range of cryptographic algorithms, including all common symmetric-key, asymmetric-key, hash and MAC algorithms (triple-DES, AES, RSA, DSA, DH, MD5, SHA-1, HMAC etc..) as well as more special algorithms such as Elliptic Curve Cryptography. SSL/TLS master key derivation and PKCS #8 key wrapping is also supported. Incorporating a true random number generator, miniHSM offers secure cryptographic key creation, and a built-in real-time clock enables a trustworthy time source for time-stamping and Digital Rights

Management applications. The clock is protected by a strong physical and logical security boundary. **[11]**

miniHSM optionally provides support for nCipher's suite of developer toolkits including CodeSafe and Secure Execution Engine for executing high assurance, trusted, custom applications within the device's security boundary and hence ensuring that confidential key material can only be used by those applications. miniHSM supports a standard smartcard interface that utilizes nCipher's Security World key management system. This provides a flexible approach to key management, backup and recovery. **[11]**

The module has two serial ports and a single 8 bit parallel port for communication and supports Windows, Linux and Solaris host operating systems. **[11]**

2.1.7. nCipher netHSM

A highly secure, network-attached hardware security module that provides a shareable cryptographic resource for multiple servers. Applications that require access to hardware-protected cryptographic keys, from PKI and authentication systems to Web services and SSL, can share access to a netHSM over secured connections. netHSM provides a cost-effective deployment alternative to dedicated security modules. All cryptographic functions are performed within a FIPS 140-2 level 3 validated security boundary. **[12]**

In line with all of nCipher's SMs, netHSM is fully compliant with nCipher's Security World key management framework. This enables keys to be managed and shared across different types of installed nCipher's SMs. Access control lists and smartcard-based operator authentication allow individual keys or groups of keys to be logically separated and specific usage rules enforced. netHSM can be configured for dual control and split responsibility ensuring that there is no single

point of compromise. Also through use of strong authentication of remote servers, use of an individual key can be restricted to a specific remote server or servers. **[12]**

netHSM is a 1U, standard rack-mounted unit, offering high performance (can perform upto 2000 signing operations per second) with a low impact on valuable rack space. It has two 10/100 Ethernet ports and a RS232 mini-DIN serial connection. It supports AIX, HP UX, Linux, Solaris and Windows operating systems. The product is certified by SAFE (Signatures and Authentication for Everyone) as having met their digital identity standard. **[12]**

AES, RC4, CAST, DES and triple-DES symmetric ciphers, RSA, DSA, El Gamal public-key ciphers, DH key exchange mechanism, MD2, MD5, SHA-1, SHA-2, RIPEMD-160 hash functions, and HMAC authentication function are supported by the module. Elliptic Curve Cryptography support is optional. **[12]**

2.1.8. netHSM Technical Architecture

netHSM architecture provides a layered approach to security. It is possible to identify four discrete security boundaries in a typical netHSM deployment:

- FIPS boundary: core security boundary for all cryptographic operations,
- Platform boundary: remaining system components and physical chassis of netHSM appliance,
- Transport boundary: network connections linking remote servers to netHSM,
- Client boundary: security features relating to netHSM but hosted on remote servers.

[13]

The innermost security zone is the FIPS security boundary. All requests for cryptographic processing by remote servers along with sensitive internal netHSM

control activities are handled within the FIPS security boundary. All plaintext key data and their associated access control lists are only exposed within this boundary. netHSM can optionally contain a Secure Execution Engine that can execute signed custom software. SEE's associated data and control logic execute within the FIPS boundary. FIPS boundary is physically protected from tampering by embedding the circuitry in hardened epoxy resin, a process known as potting. Techniques used to control access to the keys is at least equally important as physical protection of keys. In netHSM, every key is bound to an individual access control list to deliver fine-grained control over the keys. **[13]**

netHSM chassis marks the Platform security boundary. This boundary protects the embedded Intel microprocessor, its operating system and the user interface. netHSM keeps Security World data within the platform security boundary allowing control and audit via netHSM's secure user interface. The boundary is physically protected with tamper-evident seals, and cryptographically protected during upgrades using digital signature techniques. **[13]**

To prevent eavesdropping, it is vital to secure communication between netHSM and 'client' machines (remote servers). It is also important that netHSM and client machines are mutually authenticated to prevent unauthorized use of keys. A security protocol known as Impath is used to provide secure, encrypted communication of all data over network. **[13]**

netHSM security architecture supports different strengths of authentication, each appropriate for a different use. Options include: ability to connect a server as a 'soft' client with no hardware present, use of a discrete hardware token (nToken) to provide strong protection of a secret associated with the Impath connection, and use of smart cards to authorize key usage. Only clients that have been previously enrolled can connect to netHSM. The nature of the Impath connection setup protocol means that all traffic using that connection can only be read or generated by the party, which undertook the key exchange process. Identification of the party

is done by that party signing the messages with a particular key. netHSM also maintains a unique key pair for signing its Impath session setup messages, hence a remote server can be confident that it is communicating with a particular netHSM when setting up a connection. **[13]**

2.1.9. SafeNet Security Modules - Luna PCI

Luna PCI is a family of PCI card hardware security modules designed to protect cryptographic keys and accelerate sensitive cryptographic operations across a wide range of security applications. Luna PCI offers dedicated hardware-secured key management (generation, storage and backup) to ensure integrity and protection of sensitive keys throughout their lifecycle. All digital signing and verification operations are performed within the SM to increase performance and maintain security. Luna PCI SMs provide accelerated encryption in a range of models and configurations, thus offering a wide selection of security, performance and operational capabilities. **[14]**

Low-end Luna PCI models provide over 1200 asymmetric 1024-bit RSA operations per second, while high-end Luna PCI models offer a blazing 7000 asymmetric 1024-bit RSA operations per second. Luna PCI is validated at both FIPS 140-2 level 2 and level 3 operations depending on configuration and model selected. It offers strong two-factor authentication for FIPS level 3 using well-known Luna PED (PIN Entry Device) which is an integrated handheld authentication console, that does not rely on commercial keyboards or displays for administrator PIN code entry, thus creating a true trusted path authentication. The modules are packaged inside a specially designed enclosure to meet stringent requirements for tamper and intrusion resistance. **[14]**

Luna PCI supports PKCS #11, Microsoft CryptoAPI and Java JCE/JCA cryptographic APIs to simplify development and speed application deployment. It supports a broad range of cryptographic algorithms including RSA (up to 4096-bit

key), SHA-256/512, AES, TDES and many more. The modules offer Plug and Play support for Windows platforms. **[14]**

2.1.10. SafeNet Security Modules - Luna SA

Luna SA is a flexible, Ethernet-attached hardware SM offering powerful cryptographic acceleration, hardware-based key management, and multiple configuration profiles for applications where security and performance are a priority. It features a tamper-resistant rackmount chassis, secure remote management, and scalable and upgradeable configuration options. It is available in either a FIPS validated, or RoHS compliant version. Designed to ensure integrity and security of cryptographic keys for PKI root key protection applications, genuine hardware key generation for smartcard issuance applications, blazing cryptographic processing for digital signing applications, or SSL acceleration for web servers, Luna SA has the features to deliver security and performance. **[15]**

Luna SA is a high assurance SM, i.e. keys are always in hardware. It features two built-in Ethernet ports for drop-in deployment onto networks. The SM offers over 1200 1024-bit RSA decrypt operations per second for the most demanding applications. Luna SA features secure remote administration to simplify management, and FIPS 140-2 level 3 validated models offer true two-factor, trusted path, multi-person authentication of SM administrative users. Luna SA offloads computationally intensive SSL connection setups from web servers when configured for SSL acceleration. The module has already been fully integrated with popular certificate authorities, including Microsoft Certificate Services, Entrust Authority, VeriSign, and others. **[15]**

2.2. Advantech PCM-3370F-M0A1E PC/104-plus CPU Board

This section gives specifications of the Advantech PC/104-plus CPU board used in building the software-based security module in the implementation phase. This

embedded single board computer is manufactured by Advantech Corporation, and distributed by Lima Endüstriyel Bilgisayarlar A.Ş. in Turkey. A picture of the board is given in figure 1 with its connectors labeled **[33]**.

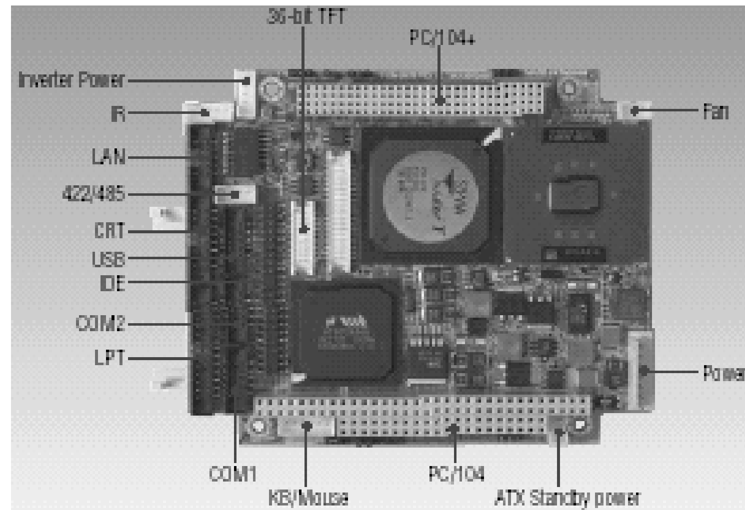

Figure 1 Advantech PCM-3370F PC/104-plus CPU Module

Regarding general specifications, the SBC has:

- Onboard Ultra Low Voltage Intel Celeron 650 MHz fanless CPU,
- VIA® VT8606/TwiserT and VT82C686B system chipset,
- AWARD® 256 KB Flash BIOS,
- Support for Advanced Power Management,
- A SODIMM socket supporting up to 512 MB SDRAM system memory,
- CompactFlash® Card (CFC) Type I socket,
- 1.6 sec interval watchdog timer which can be set up by software and generate system reset or IRQ11,
- 104-pin PC/104 and 120-pin PC/104-plus expansion connectors. **[33]**

Regarding I/O specifications, the SBC has:

- An enhanced IDE connector for hard disk or CD-ROM connections,

- An LPT parallel port,
- An RS-232/422/485 serial port and an RS-232 serial port,
- A keyboard connector and a mouse connector,
- Two USB 1.1 compliant ports,
- Realtek® RTL8139D 10/100 Mbps Ethernet with RJ-45 interface. **[33]**

Regarding display capability, the SBC has:

- Four AGP VGA/LCD controller providing up to 1024 x 768 resolution. **[33]**

The board's dimensions are PC/104 standard dimensions, 96 x 115 mm. The board operates between temperatures 0 and 60°C. +5V or +12V AT/ATX power supply is required to drive the board, maximum power required is 15 Watts. **[33]**

A VGA cable, an Ethernet RJ-45 conversion cable, an IDE cable, ATX and AT power cables and various other cables, excluding the USB cable, come free of charge in the board's package. **[33]** A 256 MB SDRAM memory, a 1 GB Type I CompactFlash® Card and a compatible power supply were procured separately.

CHAPTER 3

PROPOSED SECURITY ARCHITECTURE

The aim of this thesis work, which mainly involves design work rather than implementation, can be defined as: to design an extensive, modular and extendable (i.e. open) security architecture, whose desired parts could be realized *even* on a software-based security module and *should* be realized on a network-attachable security module. The parts of this aim and why each is included are elaborated in the following paragraphs.

The description of the security architecture designed in this work begins with the depiction of its application context. Five major elements define the typical application context of the architecture: host systems, remote systems, host security servers, the security module and administrator users, as shown in figure 2 below. Host systems are workstations or conventional servers of the local network and run host system software and server programs. Remote systems are elements of other networks and run client programs that communicate with host server programs. Host security servers are elements of the local network and run server programs that perform cryptography and security-related operations and protocols needed by host systems. These dedicated security servers may offload a part of their security-related operations to the network-attached security module (SM), which provides various cryptography and security mechanisms and securely stores secret data/information required by these mechanisms and belonging to host and remote users. The SM applies access control restrictions so that only a few administrator host users have the right to load or remove the secret data to/from the SM, and also modify the SM internal data. Various kinds of security mechanisms provided by and secret data stored in the SM together corresponds to just *a* realization of the designed security architecture.

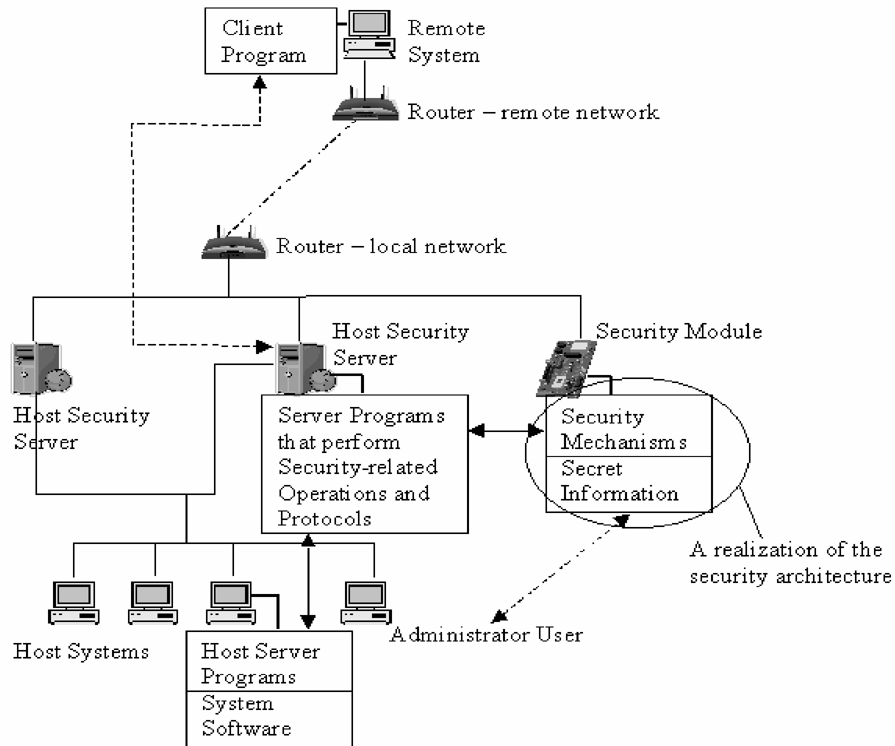

Figure 2 Typical Application Context

Mostly, security modules are coprocessor add-in cards that are plugged to one of the electrical bus sockets of a server machine and accessed only by this machine. Alternatively, they can be embedded single board computers (SBCs) that can be attached to the local network as a network device, like the one in figure 2. While an add-in card can exchange large amounts of data with the server machine directly at a time, an SBC can be accessed by any server machine in the local network. Hence, a network-attachable SBC can store secret keys and other data for all servers and users of the local network. **[1] [4]** Also, they are low cost and increasingly becoming more common.

The security architecture designed in this work should be realizable for a *network-attachable SBC* as the hardware platform.

Security modules are almost always hardware-based, but a software-based implementation is also possible. Hardware-based security modules have special cryptography hardware that accelerates certain cryptography operations, and hence they offer performance benefits over software-based implementations. In addition, a higher level of security can be achieved using hardware-based SMs, because keys and other secret data are protected against unauthorized modification and removal from the module by both hardware and software means. Software-based security modules are general-purpose computers with limited processing capabilities. In addition, they are more vulnerable to viruses and system failures. **[2]** Hardware-based security modules are costly, especially in case a security module is needed for every local network of an organization.

The security architecture designed in this work could be realizable for a hardware-based security module and *even* for a *software-based* security module.

Dedicated security servers on a local network of an organization may need a different combination of security mechanisms than the dedicated servers on another local network. First of all, this requires the software of the security module to be configurable according to the needs of the dedicated servers on any local network of the organization. Hence, the security architecture should be designed as *modular* enabling any desired combination of its security mechanisms to be realized on a security module according to the needs of the dedicated servers. Secondly, it is required that the security module software is capable of providing most of the existing and new cryptography and security mechanisms of proven reliability. This implies both an *extensive* and *extendable* security architecture to be designed with the ability to integrate new security mechanisms and remove obsolete ones to/from the architecture easily. Combining modularity and extendibility, the security architecture designed in this thesis work will be an *open* architecture enabling other developers to improve and realize parts of the architecture they wish according to their security considerations.

Then, the point is to decide which security mechanisms could be included in and which kinds of secret data could be handled by the security architecture designed. This decision is made by determining which security services could be needed by host systems and host security servers to enhance the security of data transfers with remote systems (i.e. network security services **[3]**) and to enhance the security of data processing of themselves (i.e. system security services **[3]**). The security services considered are shown in ovals in figure 3 below.

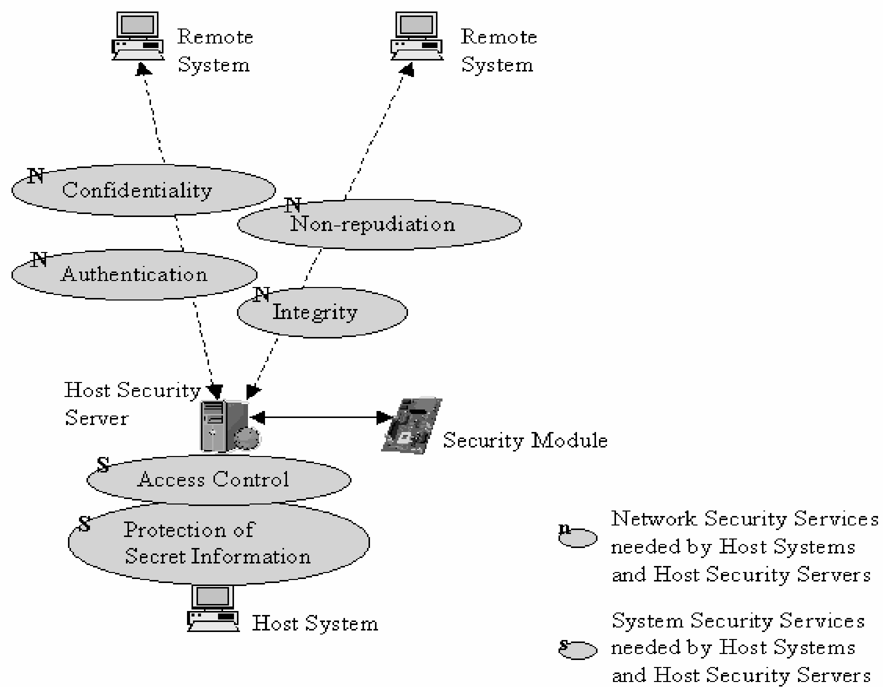

Figure 3 Security Services Needed by Host Systems and Host Security Servers

Four important network security services needed by any host system or security server to protect data during transfer to a remote system are confidentiality, authentication, integrity and non-repudiation. The most common network security mechanism to provide confidentiality is masking the data using *encryption,* either

symmetric or asymmetric **[3]**. The most commonly used network security mechanism to provide data/message authentication and integrity is the *Message authentication code* (MAC) **[3]**. The most common network security mechanism to provide non-repudiation service is the *digital signature*.

Two of the important services, related to system security, needed by any host system or security server to protect its own data processing are access control (including identification of a remote system trying to access to a host system) and protection of secret information belonging to a host system or host users. System password schemes and files, access control and capability lists are major system security mechanisms to provide access control service **[3]**. For protection of secret information, security mechanisms that protect the integrity and privacy of stored secret data are required.

The network security mechanisms (i.e. common cryptography techniques) and system security mechanisms mentioned above are included in the designed security architecture as *basic offloader mechanisms* and *system security offloader mechanisms*, respectively. The security architecture designed does not only include network and system security mechanisms, but also other security mechanisms that implement certain operations of three renowned network security protocols and certain X.509 certification operations. Three network security protocols considered within the scope of this work are: SSL transport layer security protocol, SET security protocol which protects credit card transactions over the Internet **[16]**, and Kerberos network authentication protocol. Finally, *administrational offloader mechanisms* are included in the security architecture that aim to perform security module administration tasks and can only be requested by a few administrator users (whose number should be *more than* 1). All of the security mechanisms included as part of the security architecture are referred to as *offloader mechanisms*, because host security servers may *offload* a substantial part of their security processing tasks to a security module realizing the designed security architecture.

Like host systems and host security servers, a security module, on which parts of the security architecture will be realized, needs similar network and system security services to ensure secure data exchange with connected host security servers and to ensure secure data processing of itself, respectively. Figure 4 shows these security services in ovals.

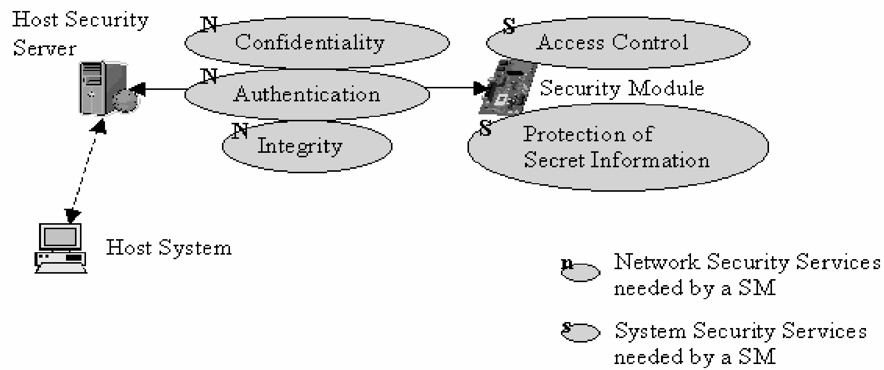

Figure 4 Security Services Needed by a Security Module

First of all, a security module needs access control system security service. A host system/user initiating a connection to the SM via a host security server should be authenticated (identified) by the SM, and vice versa the security module should be authenticated by the host user. Then, certain access rights on SM resources (i.e. offloader mechanisms and stored secret data) should be delivered to the host user by the SM. Secondly, the SM should store and protect data, either secret or not, belonging to *itself*. Therefore, the security architecture designed should reserve a place for the storage of SM internal data apart from the one for the storage of security-related data belonging to host and remote users.

As network security services, integrity and authenticity of the messages exchanged between a host security server and the SM should *always* be guaranteed by both

the SM and host security server. Also, confidentiality of the messages exchanged should be ensured by both the SM and host security server *in cases* where the messages concern exchange of secret data, like keys. Non-repudiation network security service is considered not to be definitely required by a SM, since it may seriously slow down the speed of communication between the SM and a host security server. Therefore, the security architecture *should* provide necessary security mechanisms and/or protocols that meet the SM's own security needs.

To provide access by host security servers to the offloader mechanisms provided by the security architecture, an easy-to-process *communications protocol* and associated *client-side API* functions are designed. To request an offloader mechanism, a host security server can prepare and transmit the related offloader mechanism execution request message to the SM, that is in fact a communications protocol message, by calling the related function provided by the client-side API. Final output of the executed offloader mechanism is returned by the SM to the requesting host security server in the related offloader mechanism execution response message that is also a communications protocol message. The host security server can receive this message and interpret the result of the executed offloader mechanism using the related function provided by the client-side API. Figure 5 illustrates access to offloader mechanisms.

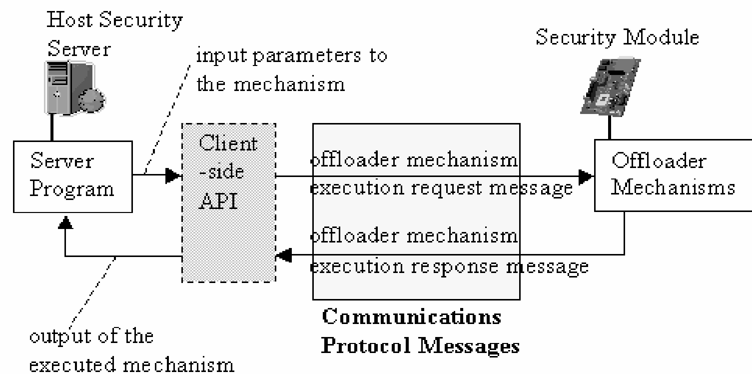

Figure 5 Access to Offloader Mechanisms via the Communications Protocol

The sections below describe various offloader mechanisms designed as part of the security architecture. Apart from them, the offloader mechanism access controller server program, which manages and controls accesses to the offloader mechanisms and connections to the security module by host security servers, is explained. Next, the communication infrastructure, which ensures authenticity and integrity (and in some cases confidentiality) of the messages exchanged between the SM and a host security server and which provides verification of each other's identities during connection to the SM, is described.

3.1. Basic Offloader Mechanisms

3.1.1. Symmetric_Encryption_Decryption Basic Offloader Mechanism

A host security server may have plaintext data encrypted on behalf of a host system/user by requesting the execution of this offloader mechanism from the security module that realizes the designed security architecture. After that, the host security server will send the resulting ciphertext data returned by the security module to the remote system/user with which the host user is communicating. In addition, a host security server may have ciphertext data (received before from a remote user) decrypted on behalf of a host user by requesting the execution of this mechanism from the SM. Then, the host security server will get the plaintext data returned by the SM and send it to the host user.

In the execution of this mechanism, the input data (plaintext or ciphertext data) may be encrypted or decrypted using the host user's symmetric encryption key and initialization vector (IV) already stored in the security module storage, according to various encryption/decryption options provided by the input communications protocol offloader mechanism execution request message. The fields of all offloader mechanism execution request messages for basic and administrational offloader mechanisms are given in detail in the tables in Appendix A. Final status of the executed mechanism along with the resulting encrypted or decrypted data (if

an error has not occurred during execution) are returned via a communications protocol offloader mechanism execution response message. The fields of all offloader mechanism execution response messages for basic and administrational offloader mechanisms are also given in detail in the tables in Appendix A.

As a requisite to include this basic offloader mechanism, the security architecture designed should provide the well-known DES, TDES, Blowfish, RC5, CAST-128, RC2, RC4, AES and IDEA symmetric encryption methods for each mode of operation (only for block encryption methods), namely ECB, CBC, CFB, OFB and PCBC.

3.1.2. Asymmetric_Encryption_Decryption Basic Offloader Mechanism

A host security server may have plaintext data (usually a secret key or a hash value) asymmetrically encrypted or signed or both signed and encrypted on behalf of a host user by requesting the execution of this offloader mechanism from the security module. Then, the host security server will send the resulting encrypted, signed, or signed and encrypted data returned by the SM to the remote user with who the host user is involved in a secure data exchange. Additionally, a host security server may have encrypted data (received before from a remote user) asymmetrically decrypted, signed data verified, or signed and encrypted data asymmetrically decrypted and verified by requesting the execution of this mechanism from the SM. As a result, the host security server will obtain the original plaintext data.

In the execution of this mechanism, the input data may be:

1. asymmetrically encrypted using the remote user's public encryption key (certificate) stored in the SM, or
2. signed using the host user's private signing key stored in the SM, or
3. signed and then asymmetrically encrypted using both the private signing key of the host user and public encryption key (certificate) of the remote

user stored in the module, or

4. asymmetrically decrypted using the host user's private encryption key stored in the module, or
5. verified using the remote user's public signing key (certificate) stored in the SM, or
6. asymmetrically decrypted and then verified using both the private encryption key of the host user and public signing key (certificate) of the remote user stored in the module,

according to various encryption and signing parameters provided by the input offloader mechanism execution request message. Final status of the executed mechanism along with the resulting encrypted, signed, signed and encrypted, decrypted, verified, or decrypted and verified output data (if an error has not occurred during the execution) are returned via a mechanism execution response message.

As a requisite to include this offloader mechanism, the security architecture designed should provide the RSA, ECC and ElGamal public-key methods.

3.1.3. Session_Key_Calculation Basic Offloader Mechanism

A host security server may have a session (symmetric) key computed on behalf of a host user, who wants to set up a secure data exchange session with a remote user, by requesting the execution of this offloader mechanism from the security module. Afterwards, the SM will securely store the computed session key on behalf of the host user.

In the execution of the mechanism, a symmetric session key is computed using:

- the host user's private key exchange key stored in the SM,
- the remote user's public key exchange key (certificate) stored in the SM, and
- the global key exchange parameters of the host user stored in the SM,

according to the options provided via the input mechanism execution request message. Then, the computed session key is securely stored in the SM storage as its integrity and privacy protected, as: its SHA-1 hash code appended and then triple-DES encrypted with the storage master key and IV of the SM. Final status of the session key calculation operation is returned via a mechanism execution response message.

As a requisite to include this offloader mechanism, the security architecture designed should provide the Diffie-Hellman and ECC public-key key exchange methods.

3.1.4. Message_Authentication_Code_Operations Basic Offloader Mechanism

A host security server may have plaintext data authenticated (using a MAC) and optionally encrypted on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the authenticated and optionally encrypted data returned by the SM to the remote user with whom the host user is communicating. In addition, a host security server may have authenticated and optionally encrypted data (received before from a remote user) optionally decrypted and then verified on behalf of a host user by requesting the execution of this mechanism from the SM. Consequently, the host security server will obtain the original data verified by the SM as coming from the claimed remote user.

In the execution of the mechanism, the input data may be:

1. authenticated by appending it a MAC computed using the host user's MAC secret key already stored in the module, or
2. authenticated as in the previous step and then encrypted using the host user's symmetric encryption key and IV already stored in the module, or
3. first encrypted using the host user's stored symmetric encryption key and IV, and then authenticated by appending it a MAC computed using the host

user's stored MAC secret key, or

4. verified by verifying the MAC at the end of it using the host user's MAC secret key stored in the module, or
5. first decrypted using the host user's stored symmetric encryption key and IV, and then verified by verifying the MAC at the end of the decrypted data using the host user's stored MAC secret key, or
6. first verified by verifying the MAC at the end of it using the host user's stored MAC secret key, and then decrypted using the host user's symmetric encryption key and IV stored in the module,

according to various MAC and symmetric encryption options provided by the input protocol mechanism execution request message. Final status of the operation along with the authenticated and optionally encrypted, or optionally decrypted and verified (original) data are returned via a mechanism execution response message.

The designed security architecture should include renowned HMAC, Fast HMAC, CMAC and DAA message authentication methods, and MD5, SHA-1, RIPEMD-160 and Whirlpool hash methods, as a requisite to include this basic offloader mechanism.

3.1.5. Message_Digest_Operations Basic Offloader Mechanism

A host security server may have plaintext data authenticated or signed (using its hash code) and optionally encrypted on behalf a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the authenticated or signed and optionally encrypted data returned by the SM to the remote user with whom the host user is communicating. Also, a host security server may have authenticated or signed (using hash code) and optionally encrypted data (received previously from a remote user) optionally decrypted and verified on behalf of a host user by requesting the execution of this offloader mechanism from the SM. Consequently, the host security server will

obtain the original data verified by the SM as coming from the claimed remote user.

In the execution of this mechanism, the input data may be:

1. authenticated by appending its symmetrically encrypted (using the host user's symmetric encryption key and IV stored in the SM) message digest, or
2. authenticated and encrypted by first appending its message digest and then encrypting the input data and its message digest using the host user's symmetric encryption key and IV stored in the SM, or
3. signed by appending its asymmetrically encrypted (using the host user's private signing key stored in the module) message digest, or
4. signed as in the previous step and then encrypted using the host user's symmetric encryption key and IV stored in the module, or
5. authenticated by appending it a message digest calculated using the input data and the hash secret value shared by the host user and remote user and stored in the SM, or
6. authenticated as in the previous step and then encrypted using the host user's symmetric encryption key and IV stored in the SM, or
7. verified by first decrypting the encrypted message digest at its end using the host user's stored symmetric encryption key and IV and then verifying the recovered message digest, or
8. decrypted using the host user's stored symmetric encryption key and IV, and then verified by verifying the message digest at its end, or
9. verified by first asymmetrically decrypting the signed message digest at its end using the public signing key (certificate) of the remote user stored in the SM and then verifying the recovered message digest, or
10. decrypted using the host user's stored symmetric encryption key and IV, and then verified as in the previous step, or
11. verified by verifying the message digest at its end using the stored hash secret value shared by the host user and remote user, or

12. decrypted using the host user's stored symmetric encryption key and IV, and then verified as in the previous step,

according to various hash, symmetric encryption and public-key encryption parameters provided by the input protocol mechanism execution request message. Final status of the executed mechanism together with the authenticated or signed and optionally encrypted, or optionally decrypted and verified (original) output data are returned via a protocol mechanism execution response message.

3.1.6. Digital_Signature_Calculation Basic Offloader Mechanism

A host security server may have plaintext data signed, especially using the DSA or ECDSA method, on behalf of a host user by requesting the execution of this offloader mechanism from the security module. Later, the host security server will send the signed data returned by the SM to the remote user with who the host user is involved in a digitally signed data exchange. In addition, a host security server may have signed data (received previously from a remote user) verified on behalf of a host user by requesting the execution of this mechanism from the SM. Hence, the host security server will get the original data verified by the SM as having sent by the claimed remote user.

In the execution of this offloader mechanism, the input data may be:

1. signed using the host user's private signing key and global signing parameters stored in the module, or
2. verified by verifying the signature at its end using the public signing key (certificate) of the remote user and global signing parameters of the host user stored in the module,

according to the parameters supplied by the input protocol mechanism execution request message. Final status of the signing/verification operation along with the signed or verified data is returned via a mechanism execution response message.

The designed security architecture should include the DSA and ECDSA public-key signing methods as a requisite to include this offloader mechanism.

3.2. Administrational Offloader Mechanisms

To execute an administrational offloader mechanism, it is required that *more than one* administrator users have been authenticated by and connected to the security module. This is a precaution against a possibility that the SSL private key of one administrator user corresponding to her SSL public-key (identity) certificate, which is used in connecting to the security module to enable the SM identify the connecting host user, can be stolen by an attacker.

Before starting to execute an administrational offloader mechanism, the security module should pass to “maintenance” state from “operational” state by completing the processing of all current offloader mechanism execution requests and not accepting any new offloader mechanism execution requests or connection requests to the module. After the execution of an administrational offloader mechanism is finished, the SM should return to “operational” state and restart to accept new mechanism execution and connection requests.

3.2.1. Cryptographic_Keys_Parameters_Generation Administrational Offloader Mechanism

An administrator user of the security module may remotely, on a host system or security server connected to the SM, have cryptographic keys and parameters generated on behalf of a host user by requesting the execution of this offloader mechanism from the SM. Consequently, the SM will store the generated keys and parameters securely in its storage, and later these will be used in performing the offloader mechanisms to be requested.

In the execution of this mechanism:

1. global public-key parameters may be generated, and then a private and a public key may be computed from these parameters, or
2. a symmetric encryption key and an initialization vector may be generated, or
3. a MAC key may be generated,

for a specific host user of the SM and according to the key generation parameters provided by the input mechanism execution request message. Generated global public-key parameters and computed private and public keys, or generated symmetric encryption key and IV, or generated MAC key are/is securely stored in the SM associated with the identifier of the host user. Final status of the keys and parameters generation process is returned in a mechanism execution response message.

The security architecture designed should provide the key (and global parameters) generation methods for RSA, DH, ECC, ElGamal, DSA, ECDSA, DES, TDES, AES, RC2, RC5, RC4, IDEA, Blowfish, CAST-128, HMAC, Fast HMAC, CMAC and DAA cryptography methods, as a requisite to include this offloader mechanism.

3.2.2. Time_Time_Window_Adjustment Administrational Offloader Mechanism

An administrator user of the security module may remotely, using a host system or security server, adjust time information and/or modify time window value (denotes a time interval against which certain input data to the SM will be checked for their timeliness) of the SM by requesting the execution of this mechanism from the SM. As a result, timestamp values generated by the SM to be included in certain output data will be more accurate, and verification of the timestamp values found in certain input data by the SM will be more reliable.

In the execution of the mechanism, the time or time window value internal data of the SM or both of them may be updated using the corresponding values provided in the input protocol mechanism execution request message. Final status of the executed mechanism is returned in a mechanism execution response message.

3.2.3. Cryptographic_Keys_Parameters_Entry_Erasure_Backup Administrational Offloader Mechanism

An administrator user of the security module may remotely, on a connected host system or security server, perform cryptographic keys, parameters, public-key certificates and CRLs entry to, erasure and backup from the SM by requesting the execution of this mechanism from the SM. Thus, cryptographic keys and parameters the SM will need to use in executing the offloader mechanisms to be requested will be available, or they could be securely transferred to the host system or security server and backed up in another storage device. Also, cryptographic keys and parameters that are not needed any more will be discarded from the valuable module storage space.

In the execution of this mechanism, cryptographic keys, parameters, public-key certificates (for various host and remote users) and certificate revocation lists may be securely entered into the SM, erased from the SM, or backed up to the host system or security server the administrator user is using, according to various parameters provided via the input mechanism execution request message. Each *secret* key and parameter is transferred to the SM or backed up to the host system/security server as its confidentiality protected: its SHA-1 hash code appended and then encrypted using triple-DES with the storage master key and IV of the module, that is, in the format for *storing* secret keys and parameters in the SM. Refer to "Cryptographic Keys, Parameters and Public-Key Certificates Storage" section in this chapter for details. Final status of the executed mechanism together with the number of keys and parameters successfully processed by the

SM, until an error occurred or all of them were finished without any error, are returned via an offloader mechanism execution response message.

3.2.4. User_Role_Management Administrational Offloader Mechanism

An administrator user of the security module may remotely perform SM user role management by requesting the execution of this mechanism from the SM. By this way, non-administrator user access to the offloader mechanisms that require administrator user rights to execute will be prevented.

In the execution of this mechanism, the security module's user-role list entries, which are simple (user, role) pairs, may be modified, added to or deleted from the list according to the parameters provided by the input mechanism execution request message. Final status of the user role management operation along with the number of (user, role) entries successfully processed, until an error occurred or all of them were finished without any error, are returned via a mechanism execution response message. Currently, two roles are defined: "User" and "Administrator".

As a requisite to include this offloader mechanism, security module user-role list, which is SM internal data, should be securely stored as part of the designed security architecture as its integrity protected: its triple-DES encrypted (using the storage master key and IV of the SM) SHA-1 hash code appended.

3.2.5. Module_Code_Status_Receiving_Loading Administrational Offloader Mechanism

An administrator user of the security module may remotely demand digitally signed code status of the SM or update the firmware of the SM via loading digitally signed code to the SM by requesting the execution of this offloader mechanism. Thus, the module's firmware could be checked for any illegitimate modification, or a new firmware with patches, that eliminate vulnerabilities or

include extensions to the proposed offloader mechanisms, could be loaded to the module.

In the execution of this mechanism:

1. stored digitally signed code status, i.e. current firmware code pieces plus their signatures (possibly RSA-SHA-1), of the module may be returned, or
2. digitally signed code, i.e. new firmware code pieces plus their signatures, may be verified by the module and loaded to the module,

according to the parameters supplied by the input mechanism execution request message. Final status of the performed operation, and if the operation type is "Get Code Status" the digitally signed code status of the module are returned in a mechanism execution response message. If the operation type is "Load Signed Code" and new digitally signed firmware is verified by the module successfully, the module should restart to run this code afterwards. Hence, all current connections from host security servers to the module will be lost.

The module firmware should be signed in code pieces, because most hash algorithms that can be used in signing the firmware generally do not accept input data with size bigger than a limit. Each code piece should be signed by *at least two* administrator users successively as a precaution against a probable theft of private signing key of one administrator user.

3.3. Cryptographic Keys, Parameters, Public-Key Certificates and CRLs Storage

Symmetric encryption keys (including session keys), initialization vectors used in all modes of symmetric encryption except ECB, private and public keys, public-key certificates, certificate revocation lists, MAC secret keys (like HMAC or DAA key), hash secret values used for providing message authentication using hash codes but without encryption **[16]**, and global public-key parameters (like DH or DSA global parameters) are all stored securely in the storage of the SM, as part of

the designed security architecture. Each of them is stored along with the information about its *nature* (e.g. AES-192-skey denotes a 192-bit AES symmetric key, RSA-1024-priv denotes a 1024-bit RSA private key, etc..), its *usage* (encryption, signing, SSL-key exchange, SSL-authentication, etc..) and its *owner's identifier*. Symmetric encryption keys, initialization vectors, MAC secret keys and hash secret values are also stored along with the *identifier of the remote user* with who the owner (host user) of the key or parameter shares the key or parameter. CRLs are also stored along with their *publication date*.

To protect the integrity and confidentiality of the symmetric encryption keys, initialization vectors, private keys, MAC secret keys and hash secret values, each of them is stored as: its SHA-1 hash code appended and then encrypted using triple-DES in CBC mode with the storage master key and IV of the module, as illustrated in figure 6 below. Public keys and global public-key parameters are stored as their TDES-encrypted (with the storage master key and IV of the module) SHA-1 hash codes appended, for integrity protection *only*. Public-key certificates, CRLs and the storage master key and IV of the security module are stored in the clear.

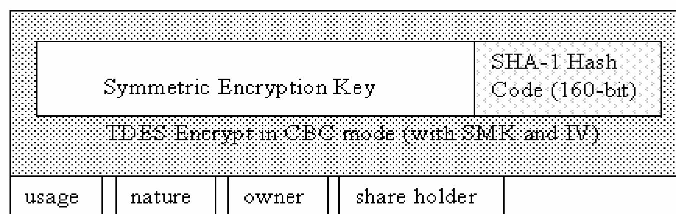

Figure 6 Secret Keys and Parameters Storage Format

Overall security architecture, after the inclusion of the basic and administrational offloader mechanisms, base cryptography methods they require, and cryptographic

keys, parameters, public-key certificates and CRLs they use, becomes as shown in figure 7:

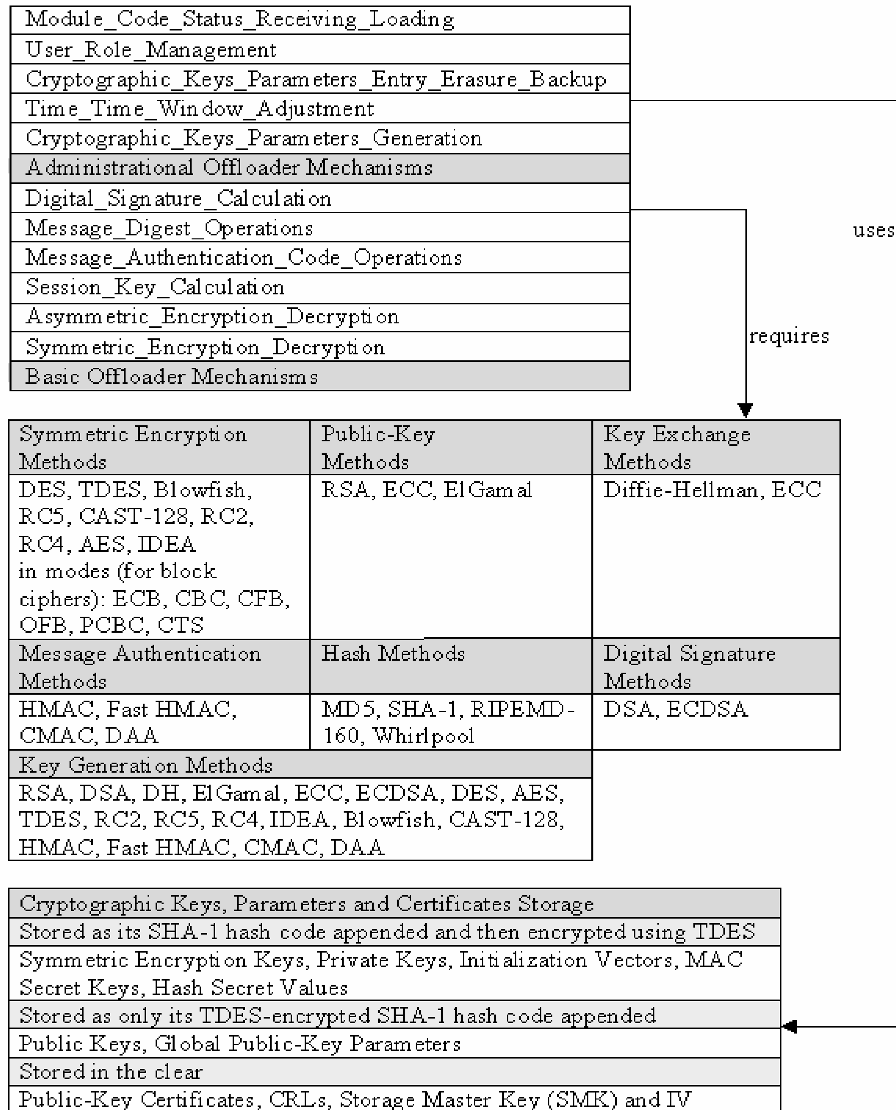

Figure 7 Overall Security Architecture (Initial)

3.4. SSLv3 Protocol Offloader Mechanisms

SSLv3 offloader mechanisms aim to reduce SSL protocol processing load on a host security server, usually an SSL server, connected to a security module which realizes the security architecture designed, by helping the host security server in preparing the contents of its outgoing and in recovering the contents of its incoming *SSL Record* protocol and *SSL Handshake* protocol messages. SSL Record protocol is used to exchange encrypted and authenticated SSL payloads between an SSL server and a client **[16]**. SSL Handshake protocol is used to setup an SSL connection between an SSL server and a client **[16]**.

3.4.1. SSL_Record_Preparation SSLv3 Offloader Mechanism

A host security server (an SSL server/client) may have the payload of an SSL Record protocol message prepared on behalf of a host user/system by requesting the execution of this offloader mechanism from the security module. Then, the host security server will add necessary SSL Record header to the SSL Record payload prepared and returned by the SM, and send the resulting SSL Record protocol message to the remote user/system who is an SSL client/server.

In the execution of this mechanism, an SSL Record protocol message payload is prepared from the input application-level data using:

- the SSL MAC secret key of the host user stored in the SM as associated with the *identifier of the specific SSL session* between the host user and remote user,
- the SSL symmetric encryption key and IV of the host user stored in the SM as associated with the *identifier of the specific SSL session* between the host user and remote user,

according to various parameters provided by the input mechanism execution request message. Final status of the SSL Record protocol payload preparation

operation, and if the operation is successful the prepared payload, are returned via a mechanism execution response message.

The security architecture designed should provide the SSLv3 MAC method and ZIP de/compression method, as a requisite to include this offloader mechanism.

3.4.2. SSL_Record_Verification SSLv3 Offloader Mechanism

A host security server (an SSL server/client) may have the payload of an SSL Record protocol message (received before from a remote user who is an SSL client/server) optionally decrypted and then verified on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server will obtain the original application-level data verified and returned by the SM, and send it to the host user.

In the execution of the mechanism, the input SSL Record protocol message payload is optionally decrypted using the SSL symmetric encryption key and IV of the host user stored in the SM, and then it is verified using the SSL MAC secret key of the host user stored in the SM, according to various parameters provided by the mechanism execution request message. Final status of the executed mechanism, and if the execution is completed successfully the recovered application-level data are returned via a mechanism execution response message.

3.4.3. Random_Value_Generation SSLv3 Offloader Mechanism

A host security server (an SSL server/client) may have an SSL pseudo-random value generated on behalf of a host user by requesting the execution of this offfoader mechanism from the SM. Then, the host security server will include the generated pseudo-random value in a “client hello” or “server hello” message to be sent to a remote user, who is an SSL client/server, during an SSL handshake.

In the execution of this offloader mechanism, a 28 byte **[16]** SSL pseudo-random value is generated by a cryptographically secure pseudo-random number generation method specified in the input mechanism execution request message. Final status of the operation along with the generated pseudo-random value, if the operation is successful, is returned via a mechanism execution response message.

The security architecture designed should provide DES used in OFB mode, ANSI X9.17 and Blum Blum Shub pseudo-random number generation methods, as a requisite to include this offloader mechanism.

3.4.4. Server_Certificate_Retrieval SSLv3 Offloader Mechanism

A host security server (an SSL server) may have SSL public-key certificate of a host system (usually a web server) retrieved from the SM by requesting the execution of this offloader mechanism from the SM. After that, the host security server will include the SSL public-key certificate returned by the SM in a "server certificate" message to be sent to a remote user, who is an SSL client, during an SSL handshake.

In the execution of the mechanism, the SSL public-key certificate of the host system is retrieved from the module storage according to the "SSL key exchange method" specified in the input mechanism execution request message:

- if the SSL key exchange method is RSA, the SSL RSA public key exchange key certificate of the host system is retrieved if it exists, otherwise the SSL RSA public signing key certificate of the host system is retrieved from the module storage,
- if the SSL key exchange method is fixed Diffie-Hellman, the SSL Diffie-Hellman public key exchange key certificate of the host system is retrieved from the module storage,
- if the SSL key exchange method is ephemeral Diffie-Hellman, the SSL RSA or DSA public signing key certificate of the host system is retrieved

from the module storage, according to the “Signing Method” parameter in the mechanism execution request message.

Other SSL public-key certificates, stored in the module, are added to the retrieved SSL public-key certificate of the host system to construct a public-key certificate chain starting from a root-level certification authority. Final status of the executed mechanism, and if the execution is completed successfully the constructed SSL public-key certificate chain for the host system are returned via a protocol mechanism execution response message.

3.4.5. SSL_Certificate_Verification SSLv3 Offloader Mechanism

A host security server (an SSL client/server) may have an SSL public-key certificate, which was received in a “server certificate” or “client certificate” message from a remote user (an SSL server/client) during an SSL handshake, verified on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server will gain access to the SSL public-key certificate of the remote user verified by and stored in the SM.

In the execution of this mechanism, each SSL public-key certificate contained in the input SSL public-key certificate chain for the remote user is verified and then stored in the module. Final status of the executed mechanism is returned via a mechanism execution response message.

3.4.6. Server_Key_Exchange_Preparation SSLv3 Offloader Mechanism

A host security server (an SSL server) may have SSL server key exchange data prepared on behalf of a host system by requesting the execution of this offloader mechanism from the SM. Later, the host security server will send the server key exchange data prepared and returned by the SM in a “server key exchange” message to a remote user, who is an SSL client, during an SSL handshake.

In the execution of this mechanism, SSL server key exchange data is prepared according to the "SSL key exchange method" specified in the input mechanism execution request message:

- if the SSL key exchange method is RSA,
  - a temporary RSA public key and private key for SSL key exchange are generated and stored in the module on behalf of the host system as associated with the *identifier of the specific SSL session* between the host system and remote user,
  - then server key exchange data is prepared using the generated temporary RSA public key and the SSL RSA private signing key of the host system already stored in the module,
- if the SSL key exchange method is anonymous DH, server key exchange data is prepared using the SSL Diffie-Hellman public key exchange key and global key exchange parameters of the host system stored in the module,
- if the key exchange method is ephemeral DH, server key exchange data is prepared using the SSL Diffie-Hellman public key exchange key and global key exchange parameters of the host system and the SSL RSA or DSA private signing key of the host system stored in the module.

Final status of the executed mechanism along with the prepared server key exchange data, if the execution is finished successfully, is returned via a protocol mechanism execution response message.

3.4.7. Server_Key_Exchange_Verification SSLv3 Offloader Mechanism

A host security server (an SSL client) may have SSL server key exchange data, which was received in a "server key exchange" message from a remote system (an SSL server) during an SSL handshake, verified on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server will gain access to the SSL key exchange parameters of the remote system verified by and stored in the SM.

In the execution of the mechanism, the input SSL server key exchange data is verified according to the "SSL key exchange method" specified in the input mechanism execution request message:

- o if the SSL key exchange method is RSA,
  - – the input server key exchange data is verified using the SSL RSA public signing key (certificate) of the remote system (SSL server) already stored in the SM,
  - – then *temporary* SSL RSA public key exchange key of the remote system contained in the verified key exchange data is securely stored in the SM,
- o if the SSL key exchange method is anonymous Diffie-Hellman, SSL DH public key exchange key and global key exchange parameters of the remote system, which are contained in the input server key exchange data, are securely stored in the SM,
- o if the SSL key exchange method is ephemeral DH,
  - – the input server key exchange data is verified using the SSL RSA or DSA public signing key (certificate) of the remote system,
  - – then SSL DH public key exchange key and global key exchange parameters of the remote system contained in the verified key exchange data are securely stored in the SM.

Final status of the verification operation is returned via a protocol mechanism execution response message.

3.4.8. Client_Certificate_Retrieval SSLv3 Offloader Mechanism

A host security server (an SSL client) may have SSL public-key certificate of a host user retrieved from the SM by requesting the execution of this offloader mechanism from the SM. Afterwards, the host security server will include the SSL public-key certificate, returned by the SM, in a "client certificate" message to be sent to a remote system, which is an SSL server, during an SSL handshake.

In the execution of this offloader mechanism, the SSL public-key certificate of the host user is retrieved from the module storage according to the “SSL key exchange method” specified by the input mechanism execution request message:

- o if SSL key exchange method is RSA or ephemeral DH, the SSL RSA or DSA public signing key certificate of the host user is retrieved from the SM storage according to the “Signing Method” parameter of the mechanism execution request message,
- o if SSL key exchange method is fixed Diffie-Hellman, the SSL DH public key exchange key certificate of the host user is retrieved from the module storage.

After that, other SSL public-key certificates, stored in the module, are added to the retrieved SSL public-key certificate of the host user to construct a public-key certificate chain starting from a root level certification authority. Final status of the operation together with the constructed SSL public-key certificate chain for the host user, if the operation is completed successfully, are returned via a mechanism execution response message.

3.4.9. Client_Key_Exchange_Preparation SSLv3 Offloader Mechanism

A host security server (an SSL client) may have SSL client key exchange data prepared on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the client key exchange data prepared and returned by the SM in a “client key exchange” message to a remote system, which is an SSL server, during an SSL handshake.

In the execution of the mechanism, SSL client key exchange data is prepared according to the “SSL key exchange method” specified in the input mechanism execution request message:

- o if the SSL key exchange method is RSA, client key exchange data is prepared using the SSL pre-master secret of the host user (already generated by and stored in the module) and the SSL RSA public key

exchange key (certificate) of the remote system stored in the module, or if such a certificate does not exist, using the SSL RSA temporary public key exchange key of the remote system stored in the module,

- o if the SSL key exchange method is anonymous DH or is ephemeral DH but client authentication was not required by the remote system (SSL server) for this SSL handshake (determined by the "Certificate Sent" parameter of the input mechanism execution request message), client key exchange data is prepared using the SSL DH public key exchange key and global key exchange parameters of the host user stored in the module,
- o if the SSL key exchange method is ephemeral DH and if the remote system required client authentication for this SSL handshake, client key exchange data is prepared using the SSL DH public key exchange key and global key exchange parameters of the host user and the SSL RSA or DSA private signing key of the host user stored in the module.

Final status of the client key exchange data preparation operation, along with the prepared client key exchange data, if the operation is completed successfully, is returned via a mechanism execution response message.

As part of the designed security architecture, each SSL pre-master secret should be stored in the SM as both its integrity and confidentiality protected: its SHA-1 hash code appended and then encrypted using triple-DES with the storage master key and IV of the SM. Also, each SSL pre-master secret should be stored as associated with the identifiers of the host and remote users (SSL server and client) who share the pre-master secret, and with the identifier of the SSL session between them.

3.4.10. Client_Key_Exchange_Verification SSLv3 Offloader Mechanism

A host security server (an SSL server) may have SSL client key exchange data, which was received in a "client key exchange" message from a remote user (an SSL client) during an SSL handshake, decrypted/verified on behalf of a host system by requesting the execution of this offloader mechanism from the SM.

Afterwards, the host security server will gain access to the SSL key exchange parameters of the remote user recovered/verified and stored by the SM.

In the execution of the mechanism, the input SSL client key exchange data is verified according to the "SSL key exchange method" specified in the input mechanism execution request message:

- o if the SSL key exchange method is RSA,
  - – the input client key exchange data is decrypted using the SSL RSA private key exchange key of the host system stored in the module, or if such a key does not exist using the SSL RSA private temporary key exchange key of the host system stored in the module, to recover the SSL pre-master secret,
  - – The recovered SSL pre-master secret is securely stored in the module as associated with the identifier of the specific SSL session between the host system and remote user,
- o if the SSL key exchange method is anonymous DH or is ephemeral DH but client authentication was not required by the host system (SSL server) for this SSL handshake (determined by the "Certificate Received" field of the mechanism execution request message), SSL DH public key exchange key and global key exchange parameters of the remote user, contained in the input client key exchange data, are securely stored in the module,
- o if the SSL key exchange method is ephemeral DH and the host system required client authentication for this SSL handshake,
  - – the input client key exchange data is verified using the SSL RSA or DSA public signing key (certificate) of the remote user stored in the module,
  - – then SSL DH public key exchange key and global key exchange parameters of the remote user, contained in the verified key exchange data, are securely stored in the module.

Final status of the key exchange data verification operation is returned via a protocol mechanism execution response message.

3.4.11. Certificate_Verify_Preparation SSLv3 Offloader Mechanism

A host security server (an SSL client) may have the contents of a “certificate verify” SSL message prepared on behalf of a host user by requesting the execution of this mechanism from the SM. After that, the host security server will send the “certificate verify” message to a remote system, which is an SSL server, during an SSL handshake.

In the execution of the mechanism, “certificate verify” SSL message contents is prepared using the SSL master secret of the host user already computed by and stored in the SM, the SSL RSA or DSA private signing key of the host user stored in the SM, and using various parameters provided by the input protocol mechanism execution request message. Final status of the executed mechanism along with the prepared “certificate verify” message contents, if the execution is successfully finished, are returned via a mechanism execution response message.

As part of the security architecture designed, each SSL master secret should be stored in the SM as both its integrity and confidentiality protected: its SHA-1 hash code appended and then encrypted using triple-DES with the SMK and IV of the module. Also, each SSL master secret should be stored as associated with the identifiers of the host and remote users (SSL server and client) who share the master secret, and with the identifier of the SSL session between them.

3.4.12. Certificate_Verify_Verification SSLv3 Offloader Mechanism

A host security server (an SSL server) may have the contents of a “certificate verify” SSL message, which was received from a remote user (an SSL client) during an SSL handshake, verified on behalf of a host system by requesting the execution of this offloader mechanism from the security module.

In the execution of this mechanism, the input "certificate verify" message contents is verified using the SSL RSA or DSA public signing key (certificate) of the remote user stored in the SM, the SSL master secret of the host system previously computed by and stored in the SM, and using various parameters provided by the input mechanism execution request message. Final status of the verification operation is returned via a protocol mechanism execution response message.

3.4.13. Finished_Preparation SSLv3 Offloader Mechanism

A host security server (an SSL client/server) may have the contents of an SSL "finished" message prepared on behalf of a host user by requesting the execution of this mechanism from the security module. Then, the host security server will send the "finished" message to a remote user (an SSL server/client) during an SSL handshake.

In the execution of this mechanism, "finished" SSL message contents is prepared using the SSL master secret of the host user already computed by and stored in the SM, and using the parameters provided by the input protocol mechanism execution request message. Final status of the executed mechanism, along with the prepared "finished" SSL message contents, if the execution is finished successfully, are returned via a mechanism execution response message.

3.4.14. Finished_Verification SSLv3 Offloader Mechanism

A host security server (an SSL client/server) may have the contents of an SSL "finished" message, which was received from a remote user (an SSL server/client) during an SSL handshake, verified on behalf of a host user by requesting the execution of this mechanism from the SM. If the verification is successful, the host security server, hence the host user, will make sure that the ongoing SSL handshake will be completed successfully.

In the execution of this mechanism, the input "finished" SSL message contents is verified using the SSL master secret of the host user already computed by and stored in the SM, and the parameters provided by the input protocol mechanism execution request message. Final status of the verification operation is returned via a mechanism execution response message.

3.4.15. Pre-master_Secret_Computation SSLv3 Offloader Mechanism

A host security server (an SSL client/server) may have an SSL pre-master secret computed on behalf of a host user during an SSL handshake by requesting the execution of this mechanism from the security module. Later, the host security server will access to the pre-master secret computed by and securely stored in the SM to compute the SSL master secret and session keys.

In the execution of this mechanism, SSL pre-master secret is computed according to the "SSL key exchange method" specified in the input protocol mechanism execution request message:

- if the SSL key exchange method is RSA, a pseudo-random 48-byte **[16]** SSL pre-master secret is generated using the cryptographically secure pseudo-random number generation method specified in the mechanism execution request message,
- if the SSL key exchange method is anonymous DH, fixed DH or ephemeral DH, the SSL pre-master secret is computed using the SSL DH private key exchange key and global key exchange parameters of the host user stored in the SM and the stored SSL DH public key exchange key (certificate) of the remote user (an SSL server/client) with who the host user is performing the SSL handshake.

After that, the computed/generated SSL pre-master secret is securely stored in the SM as associated with the identifier of the specific SSL session being established between the host and remote users. Final status of the SSL pre-master secret computation operation is returned via a mechanism execution response message.

3.4.16. Master_Secret_And_Keys_Computation SSLv3 Offloader Mechanism

A host security server (an SSL client/server) may have an SSL master secret, SSL symmetric encryption keys and initialization vectors and SSL MAC secret keys for an SSL session computed on behalf of a host user by requesting the execution of this mechanism from the SM. After that, the host security server will gain access to the SSL session encryption and MAC keys computed by and securely stored in the SM to prepare/verify SSL Record protocol messages.

In the execution of this mechanism:

- first of all, a 48-byte **[16]** SSL master secret is computed using the SSL pre-master secret of the host user already computed by and stored in the SM, and securely stored in the SM on behalf of the host user,
- then, two SSL MAC secret keys for each of the host user and remote user (an SSL server/client with who the host user is performing the SSL handshake), two SSL symmetric encryption keys and IVs for each of the host user and remote user are computed using the SSL master secret computed in the previous step; and then they are securely stored in the SM,

making use of various parameters provided by the input mechanism execution request message. Final status of the SSL master secret and session keys computation operation is returned via a mechanism execution response message.

3.5. SET Protocol Offloader Mechanisms

SET offloader mechanisms aim to lessen the SET protocol processing load on a host security server connected to a security module which realizes the designed security architecture, by helping in the preparation/verification of the contents of various outgoing/incoming SET protocol messages for certain SET transactions. The SET transactions considered in designing these offloader mechanisms are: “Purchase Request”, “Payment Authorization” and “Payment Capture”.

3.5.1. Nonce_Generation SET Offloader Mechanism

A host security server may have a SET nonce value generated on behalf of a host user, who is a customer in a SET "Purchase Request" transaction, by requesting the execution of this mechanism from the security module. After that, the host security server will include the nonce value generated by the SM in an "Initiate Request" SET message to be sent to a remote system which is the merchant in this SET transaction.

In the execution of this mechanism, a 20-byte **[23]** SET pseudo-random nonce value is generated using the cryptographically secure pseudo-random number generation method specified in the input communications protocol mechanism execution request message. Final status of the operation along with the generated nonce value, if the operation is successful, is returned via a protocol mechanism execution response message.

3.5.2. Initiate_Response_Preparation SET Offloader Mechanism

A host security server may have the contents of an "Initiate Response" SET message prepared on behalf of a host system, which is a merchant in a SET "Purchase Request" transaction, by requesting the execution of this mechanism from the security module. After that, the host security server will send the "Initiate Response" message to a remote user who is the customer in this SET transaction.

In the execution of this mechanism, the contents of a "Initiate Response" SET message is prepared using:

- the SET private signing key of the host system stored in the module,
- the SET public signing key certificate of the host system stored in the module, and a number of other stored SET public-key certificates needed to construct a public signing key certificate chain for the host system starting from a root-level certification authority,

- the SET public key exchange key certificate of the payment gateway (in this SET transaction) stored in the module, and a number of other stored SET public-key certificates needed to construct a public key exchange key certificate chain for the payment gateway starting from a root-level CA,

according to the parameters provided via the input protocol mechanism execution request message. Final status of the executed mechanism together with the prepared "Initiate Response" SET message contents, if the execution is completed successfully, are returned via a protocol mechanism execution response message.

3.5.3. Initiate_Response_Verification SET Offloader Mechanism

A host security server may have the contents of an "Initiate Response" SET message, that was received from a remote system which is a merchant in a SET "Purchase Request" transaction, verified on behalf of a host user (the customer in this SET transaction) by requesting the execution of this mechanism from the SM. After that, the host security server, hence the host user, will get the transaction ID verified by the SM, and will gain access to the SET public key exchange key certificate of the payment gateway (in this SET transaction) verified by and stored in the SM.

In the execution of the mechanism:

- each SET public-key certificate in the SET public signing key certificate chain for the remote system, which is contained in the input "Initiate Response" SET message contents, is stored in the module after it is verified,
- each SET public-key certificate in the SET public key exchange key certificate chain for the payment gateway, which is contained in the input "Initiate Response" SET message contents, is stored in the module after it is verified,
- the input "Initiate Response" SET message contents is verified using the SET public signing key (certificate) of the remote system verified and

stored in the module in the first step,

using the parameters provided by the input mechanism execution request message. Final status of the executed mechanism, and if the execution is finished successfully the verified response block contained in the input "Initiate Response" SET message contents, are returned via a protocol mechanism execution response message.

3.5.4. Purchase_Request_Preparation SET Offloader Mechanism

A host security server may have the contents of a "Purchase Request" SET message prepared on behalf of a host user, who is a customer in a SET "Purchase Request" transaction, by requesting the execution of this mechanism from the security module. Afterwards, the host security server will send the "Purchase Request" message to a remote system that is the merchant in this SET transaction.

In the execution of the mechanism, the contents of a "Purchase Request" SET message is prepared using:

- the SET private signing key of the host user stored in the module,
- the SET public key exchange key (certificate) of the payment gateway (in this SET transaction) stored in the module,
- the SET public signing key certificate of the host user stored in the module, and a number of other stored SET public-key certificates needed to construct a public signing key certificate chain for the host user starting from a root-level CA,

and using the parameters supplied by the input mechanism execution request message. Final status of the executed mechanism along with the prepared "Purchase Request" SET message contents, if the execution is completed successfully, are returned via a mechanism execution response message.

3.5.5. Purchase_Request_Verification SET Offloader Mechanism

A host security server may have the contents of a “Purchase Request” SET message, that was received from a remote user who is a customer in a SET “Purchase Request” transaction, verified on behalf of a host system (the merchant in this SET transaction) by requesting the execution of this offloader mechanism from the SM. After that, the host security server, hence the host system, will get the order information of the remote user verified and returned by the SM.

In the execution of the mechanism:

- each SET public-key certificate, contained in the input SET public signing key certificate chain for the remote user, is stored in the module after it is verified,
- the input “Purchase Request” SET message contents (only the order-related information) is verified using the SET public signing key (certificate) of the remote user verified and stored in the module in the previous step,

using the parameters provided by the input mechanism execution request message. Final status of the “Purchase Request” SET message contents verification operation, and if the operation is successfully finished the order information verified as coming from the remote user, are returned in a protocol mechanism execution response message.

3.5.6. Purchase_Response_Preparation SET Offloader Mechanism

A host security server may have the contents of a “Purchase Response” SET message prepared on behalf of a host system, which is a merchant in a SET “Purchase Request” transaction, by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the “Purchase Response” message to a remote user who is the customer in this SET transaction.

In the execution of this offloader mechanism, the contents of a "Purchase Response" SET message is prepared using:

- the SET private signing key of the host system stored in the SM,
- the SET public signing key certificate of the host system stored in the SM, and a number of other stored SET public-key certificates needed to construct a public signing key certificate chain for the host system starting from a root-level CA,

and using the parameters provided via the input mechanism execution request message. Final status of the executed mechanism, and if the execution is successfully finished the prepared "Purchase Response" SET message contents, are returned via a mechanism execution response message.

3.5.7. Purchase_Response_Verification SET Offloader Mechanism

A host security server may have the contents of a "Purchase Response" SET message, that was received from a remote system which is merchant in a SET "Purchase Request" transaction, verified on behalf of a host user (the customer in this SET transaction) by requesting the execution of this mechanism from the SM. Thus, the host security server, and the host user, will get the response block, concerning her purchase request that was verified and returned by the SM.

In the execution of this offloader mechanism:

- each SET public-key certificate in the SET public signing key certificate chain for the remote system, which is contained in the input "Purchase Response" SET message contents, is stored in the SM after it is verified,
- the input "Purchase Response" SET message contents is verified using the SET public signing key (certificate) of the remote system which was verified and stored in the module in the previous step,

using the parameters provided by the input mechanism execution request message. Final status of the verification operation, and if the operation is completed

successfully the response block verified as coming from the remote system, are returned via a protocol mechanism execution response message.

3.5.8. Authorization_Request_Preparation SET Offloader Mechanism

A host security server may have the contents of an "Authorization Request" SET message prepared on behalf of a host system, which is a merchant in a SET "Payment Authorization" transaction, by requesting the execution of this offloader mechanism from the SM. Later, the host security server will send the "Authorization Request" message to a remote system that is the payment gateway in this SET transaction.

In the execution of this mechanism, the contents of an "Authorization Request" SET message is prepared using:

- the SET private signing key of the host system stored in the SM,
- the SET public key exchange key (certificate) of the remote system stored in the SM,
- the SET public signing key certificate of the customer (in this SET transaction) stored in the SM, and other stored SET public-key certificates needed to construct a public signing key certificate chain for the customer starting from a root-level CA,
- the SET public signing key certificate of the host system stored in the SM, and other stored SET public-key certificates needed to construct a public signing key certificate chain for the host system starting from a root-level CA,
- the SET public key exchange key certificate of the host system stored in the SM, and other stored SET public-key certificates needed to construct a public key exchange key certificate chain for the host system starting from a root-level CA,

and making use of the parameters provided by the input mechanism execution request message. Final status of the executed mechanism, and the prepared

"Authorization Request" SET message contents, if the execution is finished successfully, are returned via a protocol mechanism execution response message.

3.5.9. Authorization_Request_Verification SET Offloader Mechanism

A host security server may have the contents of an "Authorization Request" SET message, that was received from a remote system which is a merchant in a SET "Payment Authorization" transaction, verified on behalf of a host system (the payment gateway in this transaction) by requesting the execution of this offloader mechanism from the SM. Afterwards, the host security server, hence the host system, will get the payment information made by the customer of this transaction and the authorization block concerning her purchase, both of which are verified and returned by the SM.

In the execution of this mechanism:

- each SET public-key certificate in the SET public signing key certificate chain for the customer, which is contained in the input "Authorization Request" SET message contents, is stored in the module after it is verified,
- each SET public-key certificate in the SET public signing key certificate chain for the remote system, which is contained in the input "Authorization Request" SET message contents, is stored in the module after it is verified,
- each SET public-key certificate in the SET public key exchange key certificate chain for the remote system, which is contained in the input "Authorization Request" SET message contents, is stored in the module after it is verified,
- the input "Authorization Request" SET message contents is verified using:
  - the SET private key exchange key of the host system stored in the SM,
  - the SET public signing key (certificate) of the customer verified and stored in the first step,
  - the SET public signing key (certificate) of the remote system verified and stored in the second step,

making use of the parameters provided by the input mechanism execution request message. Final status of the verification operation along with the verified payment information and authorization block, if the operation is successfully completed, is returned via a protocol mechanism execution response message.

3.5.10. Authorization_Response_Preparation SET Offloader Mechanism

A host security server may have the contents of an "Authorization Response" SET message prepared on behalf of a host system, which is a payment gateway in a SET "Payment Authorization" transaction, by requesting the execution of this mechanism from the security module. Later, the host security server will send the "Authorization Response" message to a remote system that is the merchant in this SET transaction.

In the execution of the mechanism, the contents of an "Authorization Response" SET message is prepared using:

- the SET private signing key of the host system stored in the module,
- the SET public key exchange key (certificate) of the remote system stored in the module,
- the SET public signing key certificate of the host system stored in the module, and other stored SET public-key certificates needed to construct a public signing key certificate chain for the host system starting from a root-level CA,

and making use of the parameters provided by the input mechanism execution request message. Final status of the executed mechanism, and if the execution is completed successfully the prepared "Authorization Response" SET message contents, are returned via a protocol mechanism execution response message.

3.5.11. Authorization_Response_Verification SET Offloader Mechanism

A host security server may have the contents of an “Authorization Response” SET message, that was received from a remote system which is a payment gateway in a SET “Payment Authorization” transaction, verified on behalf of a host system (the merchant in this transaction) by requesting the execution of this mechanism from the security module. After that, the host security server, hence the host system, will obtain the authorization block and the capture token concerning the purchase made in this transaction, both of which are verified and returned by the SM.

In the execution of this mechanism:

- each SET public-key certificate in the SET public signing key certificate chain for the remote system, which is contained in the input “Authorization Response” SET message contents, is stored in the SM after it is verified,
- the input “Authorization Response” SET message contents is verified using the SET private key exchange key of the host system stored in the SM and the SET public signing key (certificate) of the remote system verified and stored in the previous step,

making use of the parameters supplied by the input mechanism execution request message. Final status of the verification operation, and if the operation is successfully completed the authorization block and capture token, verified as coming from the remote system, are returned via a mechanism execution response message.

3.5.12. Capture_Request_Preparation SET Offloader Mechanism

A host security server may have the contents of a “Capture Request” SET message prepared on behalf of a host system, which is a merchant in a SET “Payment Capture” transaction, by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the “Capture Request”

message to a remote system, which is the payment gateway in this transaction, to request its payment concerning the purchase made in this transaction.

In the execution of this offloader mechanism, the contents of a "Capture Request" SET message is prepared using:

- o the SET private signing key of the host system stored in the SM,
- o the SET public key exchange key (certificate) of the remote system stored in the SM,
- o the SET public signing key certificate of the host system stored in the SM, and other stored SET public-key certificates needed to construct a public signing key certificate chain for the host system starting from a root-level CA,
- o the SET public key exchange key certificate of the host system stored in the SM, and other SET public-key certificates needed to construct a public key exchange key certificate chain for the host system starting from a root-level certification authority,

and using various parameters provided via the input mechanism execution request message. Final status of the executed mechanism, and if the execution is completed successfully the prepared "Capture Request" SET message contents, are returned via a protocol mechanism execution response message.

3.5.13. Capture_Request_Verification SET Offloader Mechanism

A host security server may have the contents of a "Capture Request" SET message, that was received from a remote system which is a merchant in a SET "Payment Capture" transaction, verified on behalf of a host system (the payment gateway in this SET transaction) by requesting the execution of this offloader mechanism from the SM. After that, the host security server will get the capture request block and the capture token, which was previously prepared by the host system and returned as is by the remote system, verified and returned by the SM.

In the execution of this mechanism,

- o each SET public-key certificate in the SET public signing key certificate chain for the remote system, which is contained in the input "Capture Request" SET message contents, is stored in the SM after it is verified,
- o each SET public-key certificate in the SET public key exchange key certificate chain for the remote system, which is contained in the input "Capture Request" SET message contents, is stored in the SM after it is verified,
- o the input "Capture Request" SET message contents is verified using
  - – the SET private key exchange key of the host system stored in the module,
  - – the SET public signing key (certificate) of the remote system verified and stored in the first step,

using the parameters provided by the input mechanism execution request message. Final status of the verification operation, and if the operation is successfully completed the verified capture request block and capture token are returned via a mechanism execution response message.

3.5.14. Capture_Response_Preparation SET Offloader Mechanism

A host security server may have the contents of a "Capture Response" SET message prepared on behalf of a host system, which is a payment gateway in a SET "Payment Capture" transaction, by requesting the execution of this offloader mechanism from the SM. Later, the host security server will send the "Capture Response" message to a remote system that is the merchant in this transaction.

In the execution of this mechanism, the contents of a "Capture Response" SET message is prepared using:

- o the SET private signing key of the host system stored in the module,
- o the SET public key exchange key (certificate) of the remote system stored in the module,

- the SET public signing key certificate of the host system stored in the module, and other stored SET public-key certificates needed to construct a public signing key certificate chain for the host system starting from a root-level CA,

and using the parameters supplied by the input mechanism execution request message. Final status of the executed mechanism along with the prepared "Capture Response" SET message contents, if the execution is finished successfully, are returned via a mechanism execution response message.

3.5.15. Capture_Response_Verification SET Offloader Mechanism

A host security server may have the contents of a "Capture Response" SET message, that was received from a remote system which is a payment gateway in a SET "Payment Capture" transaction, verified on behalf of a host system (the merchant in this SET transaction) by requesting the execution of this offloader mechanism from the SM. Afterwards, the host security server, hence the host system, will get and store the capture response block verified and returned by the SM.

In the execution of this offloader mechanism:

- each SET public-key certificate in the SET public signing key certificate chain for the remote system, which is contained in the input "Capture Response" SET message contents, is stored in the module after it is verified,
- the input "Capture Response" SET message contents is verified using the SET private key exchange key of the host system stored in the module, and the SET public signing key (certificate) of the remote system which was verified and stored in the module in the previous step,

making use of the parameters supplied by the input protocol mechanism execution request message. Final status of the verification operation, and if the operation is successfully completed the verified capture response block and its signature,

contained in the verified "Capture Response" SET message contents, are returned via a protocol mechanism execution response message.

3.6. Kerberos Protocol Version 5 Offloader Mechanisms

The offloader mechanisms described in this section implements certain Kerberos network authentication protocol operations, each of which is performed by one of the four parties defined by the protocol: client, authentication server (AS), ticket-granting server (TGS), and server **[16]**.

3.6.1. Nonce_Generation Kerberos Offloader Mechanism

A host security server may have a Kerberos nonce value securely generated on behalf of a host user, who is a client in a Kerberos "Authentication Service" exchange or "Ticket-granting Service" exchange, by requesting the execution of this mechanism from the SM. After that, the host security server will send the nonce value returned by the SM to a remote system, which is the AS/TGS in this Kerberos exchange, to request a ticket-granting ticket/service-granting ticket.

In the execution of this mechanism, a pseudo-random Kerberos nonce value is generated using the cryptographically secure pseudo-random number generation method specified in the input protocol mechanism execution request message. Generated nonce value is returned along with the final status of the operation via a mechanism execution response message.

3.6.2. TGT_Preparation Kerberos Offloader Mechanism

A host security server, which is an authentication server in a Kerberos "Authentication Service" exchange, may have a ticket-granting ticket (TGT) and the associated encrypted response data prepared by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the

TGT and encrypted response data returned by the SM to a remote user who is the client in this Kerberos exchange.

In the execution of this mechanism,

- a Kerberos ticket-granting ticket is prepared using the Kerberos symmetric encryption key of the host security server, which is stored in the module and shared with the TGS requested to be accessed by the remote user, and
- the associated encrypted response data is prepared using the remote user's Kerberos client password contained in the Kerberos client passwords file stored in the module as associated with the identifier of the host security server,

and using various parameters provided by the input protocol mechanism execution request message. Final status of the executed mechanism, and if the execution is successfully completed the prepared TGT and encrypted response data, are returned via a protocol mechanism execution response message.

The security architecture designed should provide the Kerberos client password-to-key transformation method, as a requisite to include this offloader mechanism. The security architecture designed should provide secure storage for Kerberos client passwords files belonging to Kerberos authentication servers, i.e. each such file should be stored in the SM as both its integrity and confidentiality protected: its SHA-1 hash code appended and then encrypted using triple-DES with the storage master key and IV of the SM.

3.6.3. TGT_Response_Verification Kerberos Offloader Mechanism

A host security server may have encrypted TGT response data, which was received from a remote system which is an AS in a Kerberos “Authentication Service” exchange, decrypted and verified on behalf of a host user, who is the client in this Kerberos exchange, by requesting the execution of this offloader mechanism from the SM. After a successful verification, the host security server hence the host user

will gain access to a Kerberos session key, stored by the SM that will be used in a subsequent "Ticket-granting Service" exchange with a TGS.

In the execution of this mechanism, the input encrypted TGT response data is decrypted and verified using the host user's Kerberos client password stored in the module and the parameters supplied by the input protocol mechanism execution request message. If the operation is successfully completed, the Kerberos session (symmetric) key, which is contained in the verified TGT response data and will be shared by the host user and the TGS requested to be accessed by the host user, is securely stored in the module on behalf of the host user. Final status of the executed mechanism is returned via a protocol mechanism execution response message.

As part of the designed security architecture, each *single* Kerberos client password is stored securely as both its integrity and confidentiality protected: its SHA-1 hash code appended and then encrypted using triple-DES with the storage master key and IV of the security module.

3.6.4. SGT_Authenticator_Preparation Kerberos Offloader Mechanism

A host security server may have a Kerberos authenticator value prepared on behalf of a host user, who is a client in a Kerberos "Ticket-granting Service" exchange, by requesting the execution of this mechanism from the SM. After that, the host security server will send the authenticator returned by the SM to a remote system, which is the TGS in this Kerberos exchange, to request a service-granting ticket.

In the execution of this mechanism, an authenticator block for requesting a SGT is prepared using the Kerberos session key, already stored in the module and shared by the host user and remote system, and the parameters provided by the input mechanism execution request message. Final status of the executed mechanism

along with the prepared authenticator block, if the execution is finished successfully, are returned via a protocol mechanism execution response message.

3.6.5. TGT_Verification Kerberos Offloader Mechanism

A host security server, which is a ticket-granting server in a Kerberos "Ticket-granting Service" exchange, may have a ticket-granting ticket and the associated authenticator block, that were received from a remote user who is the client in this Kerberos exchange, verified by requesting the execution of this mechanism from the security module. After a successful verification, the host security server will gain access to a Kerberos session key, stored by the SM that will be used for the subsequent message to be sent to the remote user in this exchange.

In the execution of this mechanism,

- the input TGT is decrypted and verified using the Kerberos symmetric encryption key of the host security server, stored in the module and shared with the AS in this Kerberos exchange,
- the Kerberos session (symmetric) key, which is contained in the verified TGT and will be shared by the host security server and remote user, is securely stored in the module on behalf of the host security server,
- the input authenticator block is decrypted and verified using the Kerberos session key obtained in the previous step,

making use of the parameters provided by the mechanism execution request message. Final status of the TGT verification operation is returned via a protocol mechanism execution response message.

3.6.6. SGT_Preparation Kerberos Offloader Mechanism

A host security server, which is a ticket-granting server in a Kerberos "Ticket-granting Service" exchange, may have a service-granting ticket and the associated encrypted response data prepared by requesting the execution of this offloader

mechanism from the SM. After that, the host security server will send the SGT and encrypted response data returned by the SM to a remote user who is the client in this Kerberos exchange.

In the execution of this offloader mechanism,

- o a Kerberos SGT is prepared using the Kerberos symmetric encryption key of the host security server, stored in the module and shared with the server which is requested to be accessed by the remote user,
- o the associated encrypted response data is prepared using the Kerberos session key, already stored in the SM and shared by the host security server and remote user,

making use of various parameters provided by the input protocol mechanism execution request message. Final status of the SGT preparation operation, and if the operation is completed successfully the prepared SGT and encrypted response data are returned in a protocol mechanism execution response message.

3.6.7. SGT_Response_Verification Kerberos Offloader Mechanism

A host security server may have encrypted SGT response data, that was received from a remote system which is a TGS in a Kerberos "Ticket-granting Service" exchange, decrypted and verified on behalf of a host user, who is the client in this Kerberos exchange, by requesting the execution of this mechanism from the SM. Afterwards, the host security server, hence the host user, will gain access to a Kerberos session key, stored by the SM, that will be used in a subsequent Kerberos "Client/server Authentication" exchange.

In the execution of the mechanism, the input encrypted SGT response data is decrypted and verified using the Kerberos session key, stored in the module and shared by the host user and remote system, and using the parameters provided by the input protocol mechanism execution request message. If the verification is successful, the Kerberos session (symmetric) key, which is contained in the

verified SGT response data and will be shared by the host user and the server requested to be accessed by the host user, is securely stored in the SM on behalf of the host user. Final status of the executed mechanism is returned via a protocol mechanism execution response message.

3.6.8. Service_Authenticator_Preparation Kerberos Offloader Mechanism

A host security server may have a Kerberos authenticator value prepared on behalf of a host user, who is a client in a Kerberos "Client/server Authentication" exchange, by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the authenticator value prepared and returned by the SM to a remote system, which is the server in this Kerberos exchange, to request service from the remote system.

In the execution of this mechanism, an authenticator block for requesting service is prepared using the Kerberos session key, already stored in the module and shared by the host user and remote system, according to the parameters provided by the input mechanism execution request message. Final status of the authenticator preparation operation along with the prepared authenticator block, if the operation is finished successfully, are returned via a protocol mechanism execution response message.

3.6.9. SGT_Verification Kerberos Offloader Mechanism

A host security server may have a service-granting ticket and the associated authenticator block, that were received from a remote user who is a client in a Kerberos "Client/server Authentication" exchange, verified on behalf of a host system (the server in this Kerberos exchange) by requesting the execution of this offloader mechanism from the security module. After a successful verification, the host security server hence the host system will gain access to a Kerberos session key, stored by the SM that will be used for the subsequent message to be sent to

the remote user in this Kerberos exchange. Also, the host security server hence the host system will gain access to a Kerberos subkey, stored by the SM, that will be used in later communication between the host system and remote user after Kerberos exchanges finish.

In the execution of this mechanism,

- the input service-granting ticket is decrypted and verified using the Kerberos symmetric encryption key of the host system stored in the module and shared with the TGS in this Kerberos exchange,
- the Kerberos session (symmetric) key, which is contained in the verified SGT and will be shared by the host system and remote user, is securely stored in the module on behalf of the host system,
- the input authenticator block is decrypted and verified using the Kerberos session key obtained in the previous step, and
- the Kerberos subkey (symmetric), which is contained in the verified authenticator block and will be shared by the host system and remote user, is securely stored in the module on behalf of the host system,

using the parameters supplied by the input protocol mechanism execution request message. Final status of the operation together with the sequence number obtained from the verified authenticator block, if the operation is completed successfully, are returned in a mechanism execution response message.

3.6.10. Service_Response_Preparation Kerberos Offloader Mechanism

A host security server may have encrypted service response data prepared on behalf of a host system, which is a server in a Kerberos "Client/server Authentication" exchange, by requesting the execution of this mechanism from the SM. After that, the host security server will send the service response data prepared and returned by the SM to a remote user who is the client in this Kerberos exchange.

In the execution of this mechanism, encrypted service response data is prepared using the Kerberos session key, stored in the SM and shared by the host system and remote user, and the parameters provided by the input protocol mechanism execution request message. Final status of the operation and the prepared service response data, if the operation is successfully finished, are returned in a protocol mechanism execution response message.

3.6.11. Service_Response_Verification Kerberos Offloader Mechanism

A host security server may have encrypted service response data, that was received from a remote system which is a server in a Kerberos "Client/server Authentication" exchange, verified on behalf of a host user, who is the client in this Kerberos exchange, by requesting the execution of this mechanism from the SM. After that, the host security server, hence the host user, will gain access to a Kerberos subkey, stored by the SM that will be used in later communication between the host user and remote system after Kerberos exchanges finish.

In the execution of the mechanism, the input encrypted service response data is decrypted and verified using the Kerberos session key, already stored in the SM and shared by the host user and remote system, and using the parameters provided by the input mechanism execution request message. If the verification is successful, the Kerberos subkey (symmetric), which is contained in the verified service response data and will be shared by the host user and remote system, is securely stored in the module on behalf of the host user. Final status of the executed mechanism together with the sequence number contained in the verified service response data are returned in a mechanism execution response message.

3.7. X.509 Certification Operations Offloader Mechanisms

The offloader mechanisms described in this section aim to lessen X.509 certification operations processing burden on a host security server, usually a

certification authority, connected to a security module which realizes the designed security architecture. The certification operations offloaded by the security module from the host security servers include X.509 certificate sign request (CSR) creation, X.509 CSR verification, X.509 v3 public-key certificate creation, X.509 certificate revocation list creation, and X.509 v3 public-key certificate and X.509 CRL verification.

3.7.1. Certificate_Sign_Request_Preparation X.509 Offloader Mechanism

A host security server may have an X.509 certificate sign request prepared on behalf of a host user, who is an X.509 public-key certificate requester, by requesting the execution of this offloader mechanism from the SM. After that, the host security server will send the CSR prepared and returned by the SM to a remote system that is the certification authority.

In the execution of the mechanism, an X.509 CSR block is constructed using the public key of the host user already stored in the module (and maybe her stored global public-key parameters if her public key is a DSA, DH public key etc..), and the corresponding private key of the host user stored in the module, according to various parameters provided via the input protocol mechanism execution request message. Final status of the CSR preparation operation and the prepared X.509 CSR block if the operation is successfully completed, are returned via a protocol mechanism execution response message.

3.7.2. Certificate_Sign_Request_Verification X.509 Offloader Mechanism

A host security server, which is a certification authority, may have an X.509 certificate sign request that was received from a remote user who is the certificate requester, verified by requesting the execution of this offloader mechanism from the SM. After that, the host security server will get the CSR block, whose

signature was verified by the SM, and become ready for a subsequent X.509 v3 public-key certificate generation operation.

In the execution of this offloader mechanism, the input X.509 CSR block is verified using the public key of the remote user contained in it and using the parameters provided by the input mechanism execution request message. Final status of the verification operation along with the verified X.509 CSR block if the operation is finished without any error, are returned via a protocol mechanism execution response message.

3.7.3. Certificate_Preparation X.509 Offloader Mechanism

A host security server, which is a certification authority, may have an X.509 v3 public-key certificate prepared by requesting the execution of this mechanism from the SM. Afterwards, the host security server will send the public-key certificate prepared and returned by the SM to a remote user, who is the certificate requester, in response to her CSR.

In the execution of the mechanism,

- an X.509 v3 public-key certificate for the remote user is prepared using the private certificate-signing key of the host security server stored in the module,
- the prepared X.509 v3 public-key certificate is stored in the module,
- a number of other X.509 public-key certificates, already stored in the module, are added to the prepared X.509 public-key certificate to construct an X.509 public-key certificate chain for the remote user starting from a root-level CA, and
- an X.509 certificate revocation list, stored in the module and previously published by the host security server, may be added to the constructed public-key certificate chain,

making use of the parameters provided by the input protocol mechanism execution

request message. Final status of the operation, and if the operation is successfully finished the constructed X.509 public-key certificate chain (may contain an X.509 CRL) are returned via a protocol mechanism execution response message.

3.7.4. Certificate_Revocation_List_Preparation X.509 Offloader Mechanism

A host security server, which is a certification authority, may have an X.509 CRL prepared by requesting the execution of this offloader mechanism from the SM. After that, the host security server may at any time access the X.509 CRL prepared by and stored in the SM, and send it to communicating parties to inform them about the public-key certificates revoked by the host security server.

In the execution of this mechanism, an X.509 CRL is prepared using the private CRL-signing key of the host security server stored in the SM and the stored X.509 public-key certificates that were revoked, and making use of various parameters provided by the input mechanism execution request message. The prepared X.509 CRL is stored in the SM. Final status of the operation, and if the operation is finished without any error the prepared X.509 CRL are returned via a mechanism execution response message.

3.7.5. Certificate_CRL_Retrieval X.509 Offloader Mechanism

A host security server may have an X.509 public-key certificate of a communicating party and/or an X.509 CRL, published by a certification authority, retrieved from the SM storage on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server may send the public-key certificate and/or the CRL returned by the SM to a remote user who is requesting them.

In the execution of this offloader mechanism,

- the X.509 public-key certificate of the communicating party, identified in

the input mechanism execution request message, is retrieved from the module storage,

- a number of other X.509 public-key certificates, stored in the module, are added to the retrieved certificate to construct a public-key certificate chain for the communicating party starting from a root-level CA,

and/or

- the stored X.509 CRL published (at a certain *date*) by the certification authority, identified in the input mechanism execution request message, is retrieved from the module storage.

Final status of the public-key certificate and/or CRL retrieval operation along with the constructed X.509 public-key certificate chain and/or the retrieved X.509 CRL, if the operation is successfully finished, are returned via a mechanism execution response message.

3.7.6. Certificate_CRL_Verification X.509 Offloader Mechanism

A host security server may have an X.509 public-key certificate chain and/or an X.509 CRL, which were received from a remote system (a CA) in response to a certificate sign request or from a remote user in response to a certificate/CRL request message, verified on behalf of a host user by requesting the execution of this offloader mechanism from the SM. After that, the host security server, hence the host user, will gain access to the public-key certificate and/or the CRL verified by and stored in the SM.

In the execution of this mechanism,

- each X.509 public-key certificate contained in the input X.509 public-key certificate chain is stored in the module after it is verified,
- the input X.509 CRL is verified using the stored public CRL-signing key (certificate) of the certification authority which published the CRL, and the verified CRL is stored in the module,

making use of the parameters provided by the input mechanism execution request

message. Final status of the verification operation is returned via a mechanism execution response message.

3.8. System Security Offloader Mechanisms

Regarding system security, system user passwords data and the two related operations: verifying an entered user password, and storing/deleting user password records, are assessed as the most appropriate data and operations for storing and implementing as part of the designed security architecture. In addition, system access control list or capability list data and the three related operations: verifying the legitimacy of an attempted access by a system user to a system resource, storing access rights entries, and getting stored access rights entries concerning given system users and resources, are also assessed as suitable data and operations for storing and implementing as part of the security architecture designed.

First of all, host system user passwords (e.g., user login passwords for an OS or a DBMS) does not constitute a large volume of data, that would otherwise easily use up the limited storage space of a security module which is an embedded SBC realizing the designed security architecture. Usually, there is only one (or two) password for each user of a host system. More importantly, system user passwords are in fact *secret* data, i.e. reveal of a user's password makes the password useless and the operations depending on this password vulnerable. These together make host system user passwords file/data an ideal choice for securely storing in the SM storage, as part of the designed security architecture.

Moreover, the two system security operations, mentioned above, which process system user passwords data, do not have high processing power requirements, that would otherwise severely slow down executions of other offloader mechanisms in the SM. To verify an entered user password, it is only needed to locate the password record, belonging to a given user, in the host system user passwords file, and to compare this record with the entered user password. Storing system user

password records operation refers to creating new password records for or modifying or deleting existing password records of a number of given host system users. A password checking procedure may be performed before the creation or modification of each password, however this may not require high processing power if the procedure depends on a previously created and stored data model (proactive password checking techniques, refer to **[3]**).

Secondly, host system access control list or capability list data also does not require a large storage space. An access rights entry is needed to be stored for each access relation, which indicates that a system user has certain access rights on a system object **[3]**, defined in the system. However, access control list or capability list data does *not* represent secret data, i.e. reveal of an access rights entry of the list does not make the revealed entry or the entire list useless nor does it make the operation of verification of an attempted access to a system object vulnerable. However, access control list or capability list represents critical data for a host system's security, hence their *integrity* should be protected. These together make them suitable to securely store in the SM storage.

Moreover, the three system security-related operations, mentioned above, which make use of either access control list or capability list data, require little processing power to execute. To verify an attempted access by a host system user to a host system object, it is only needed to locate the access rights entry, concerning the given host system user and object, in the access control or capability list, and to determine if the attempted type of access is among the access rights included in the located access rights entry. Storing an access rights entry in an access control or a capability list indicates creating, modifying or deleting the access rights entry, concerning a given system user and object, in/from the list. Getting an access rights entry from an access control or a capability list is an operation very similar to the verification of an attempted access to a system object, only it does not involve the step of determining whether an attempted type of access is allowable.

3.8.1. User_Password_Records_Storing System Security Offloader Mechanism

A host security server may have candidate system user passwords checked by and then stored in the security module, or may have existing system user password records deleted from the SM, on behalf of a host system (usually an OS or a DBMS) by requesting the execution of this offloader mechanism from the SM. Afterwards, the host security server, hence the host system, will gain access to the system user password records stored by the SM on behalf of the host system, when required for verification of an entered user password.

In the execution of this offloader mechanism,

- each input host system user password to be created or modified is checked using a proactive password checking procedure,
- each input host system user password, which has successfully passed the password checking procedure, is stored as associated with the identifier of the host system user, who owns this password, in a password record in the host system user passwords file which is stored in the SM as associated with the identifier of the host system and name of its system software,
- each host system user password record specified to be deleted is deleted from the host system user passwords file stored in the SM,

using the parameters provided by the input protocol mechanism execution request message. Final status of the operation along with the number of successfully checked and stored or deleted host system user passwords, until an error occurred or all of them are finished without any error, are returned via a mechanism execution response message.

As part of the security architecture designed, a proactive password checking procedure, which may well be based on both a rule enforcement system and a Markov model, should be included as a requisite to include this offloader mechanism. Also, as part of the designed security architecture, each host system

user passwords file should be stored securely as both its integrity and confidentiality protected: SHA-1 message digest of the file is appended at the end of the file, and then the file plus its message digest is encrypted using triple-DES with the storage master key and IV of the module.

3.8.2. Entered_Password_Verification System Security Offloader Mechanism

A host security server may have an entered user identifier and user password verified on behalf of a host system by requesting the execution of this offloader mechanism from the SM. It is required that the SM has already securely stored the system user passwords file belonging to the host system.

In the execution of this offloader mechanism, the input user password is verified using the password record (if such a record exists) associated with the user, whose identifier is provided in the input mechanism execution request message, and contained in the host system user passwords file stored in the SM as associated with the identifier of the host system and name of its system software. Final status of the verification operation is returned via a mechanism execution response message.

3.8.3. Access_Rights_Entries_Storing System Security Offloader Mechanism

A host security server may have several access rights entries stored in the security module or deleted from the SM on behalf of a host system (usually an OS or a DBMS) by requesting the execution of this mechanism from the SM. After that, the host security server, hence the host system, will gain access to the access rights entries securely stored by the SM on behalf of the host system, when required for verification of an attempted access to a host system object.

In the execution of this mechanism,

- o each input host system access rights to be created or modified is stored as associated with the identifier of the host system user, who will own the rights, in an access rights entry in the
    - host system access control list stored in the SM as associated with the identifier of the host system, name of its system software and the *identifier of the host system object* on which these access rights will be applied,

  or

  each input host system access rights to be created or modified is stored as associated with the identifier of the host system object, on which these rights will be applied, in an access rights entry in the
    - host system capability list stored in the SM as associated with the identifier of the host system, name of its system software and the *identifier of the host system user* who will own these access rights,

  depending on which data structure is preferred by the host system for storing its access rights entries,
- o each host system access rights entry specified to be deleted is deleted from
    - the host system access control list or
    - the host system capability list

  stored in the SM, depending on which one is preferred by the host system for storing its access rights entries,

using the parameters supplied by the input mechanism execution request message. Final status of the operation along with the number of successfully stored or deleted host system access rights entries, until an error occurred or all of them are finished without any error, are returned via a mechanism execution response message.

As part of the security architecture designed, each host system access control list and capability list data should be stored securely as its integrity protected: its

TDES-encrypted (with the storage master key and IV of the module) SHA-1 message digest appended at its end.

3.8.4. Attempted_Access_Verification System Security Offloader Mechanism

A host security server may have an attempted access by a system user to a system object verified, as to whether it is legitimate, on behalf of a host system by requesting the execution of this offloader mechanism from the SM. It is required that the SM has already stored the access control list or capability list data belonging to the host system.

In the execution of the mechanism, the input attempted type of access is verified using the access rights entry associated with the identifier of the host system user/object, which is provided in the input mechanism execution request message, and contained in the host system access control/capability list stored in the SM as associated with the identifier of the host system object/user, which is also provided in the mechanism execution request message. Final status of the verification operation is returned via a protocol mechanism execution response message.

3.8.5. Access_Rights_Entries_Retrieval System Security Offloader Mechanism

A host security server may have

- the access rights of a system user on a system object, or
- the access rights of a system user on all system objects, or
- the access rights of all system users on a system object,

retrieved from the security module, on behalf of a host system, by requesting the execution of this mechanism from the SM. It is required that the SM has already stored the access control list or capability list data belonging to the host system.

In the execution of this offloader mechanism,

- o the access rights entry associated with the identifier of the host system user/object, which is provided in the input mechanism execution request message, and contained in the host system access control/capability list stored in the module as associated with the identifier of the host system object/user, which is also provided in the mechanism execution request message,

or

- o the access rights entries associated with the identifier of the host system user and contained in *all* host system access control lists which are stored in the module as associated with the identifier of the host system; or all access rights entries in the host system capability list which is stored in the module as associated with the identifier of the host system user,

or

- o the access rights entries associated with the identifier of the host system object and contained in *all* host system capability lists which are stored in the module as associated with the identifier of the host system; or all access rights entries in the host system access control list which is stored in the module as associated with the identifier of the host system object,

is/are retrieved from the SM storage using the parameters provided by the input protocol mechanism execution request message. Final status of the executed mechanism, and if the execution is finished successfully the retrieved access rights entry(ies) are returned via a protocol mechanism execution response message.

3.9. Offloader Mechanism Access Controller

Offloader mechanism access controller is a server program located on top of the communication infrastructure that resides at the bottommost level of the designed security architecture. Its main functions are: to process connection requests from host security servers to the security module (which realizes the designed security architecture), and to direct offloader mechanism execution requests from

connected host security servers to the related offloader mechanisms for their processing.

When the offloader mechanism access controller receives connection requests from host security servers on behalf of host users, it controls *access to the security module* by host users. Only host users, who were *registered* before and have the required *SSL public-key certificate* (*identity certificate*) signed by a trusted certification authority, are allowed to connect to the security module. The offloader mechanism access controller also handles disconnection requests of host security servers from the SM to cleanly shutdown their connections.

The retrieval and initial processing of a protocol offloader mechanism execution request message, sent by a host security server to the SM, are performed by the offloader mechanism access controller. Most importantly, this initial processing involves determining whether the host user/system, on behalf of who the host security server requests execution of the offloader mechanism, is allowed to *access this offloader mechanism*. Only administrator host users may access/request the administrational offloader mechanisms. According to the outcome of this initial processing, the offloader mechanism access controller may transfer the contents of a valid (initially) request message and the control of processing of the request to the responsible offloader mechanism. After the responsible offloader mechanism processes the request, the mechanism returns its output (can also be an error code) to the offloader mechanism access controller for construction of an offloader mechanism execution response message that will be returned to the requesting host security server.

The offloader mechanism access controller server program should be capable of effectively handling many offloader mechanism execution and connection requests from a number of clients (host security servers on behalf of host users) simultaneously. To have this capability, the server program should be multi-threaded and allocate dedicated processing threads to serve each client connection,

and the program will have a separate main program thread to handle connection requests to the security module.

3.10. Communication Infrastructure

The communication between the security module (realizing the designed security architecture) and a host security server will be via the TCP/IP transport and network-level protocols combination, which will establish a proven and reliable communication infrastructure for accessing and exchanging data packets with the *network-attached* security module. To secure this communication, the SSLv3 protocol, which provides transport-level security over TCP/IP **[16]**, is considered appropriate.

Both the security module and the host security server/host user, which wants to connect to it, should send their *SSL public-key certificates*, signed by a trusted CA, to each other (instead of only the security module sends) during the SSL handshake performed for connecting to the SM. Hence, **peer authentication** will be achieved by *both the security module and the host security server/host user*. This means that a man-in-the-middle attack (as in the case of SSL handshake using anonymous Diffie-Hellman where no certificates sent **[16]**) will, theoretically, not be possible. SSL RSA key exchange (but with a mandatory SSL client certificate sending) may well be used to do the SSL handshake for performance reasons (refer to chapter 4 Reference Implementation).

Since, the authentication and integrity of the messages exchanged are the primary concerns regarding the security of communication between the SM and a host security server, and the confidentiality of the messages exchanged is not that critical in many cases, message exchanges between the SM and a host security server may be realized *only with the authentication option* of the SSLv3 protocol for performance reasons. However, in cases where the requested offloader mechanism involves the transfer of cryptographic keys, parameters or other *secret*

data to/from the security module (like the “Entered_Password_Verification” and “User_Password_Records_Storing” system security offloader mechanisms), message exchanges between the SM and the host security server should be realized with *both the authentication and encryption options* of the SSLv3 protocol to also protect the confidentiality of the transferred secret keys, parameters and data.

Figure 8 below illustrates peer authentication, message exchange between the SM and a host security server/host user, and access control to the security module and offloader mechanisms.

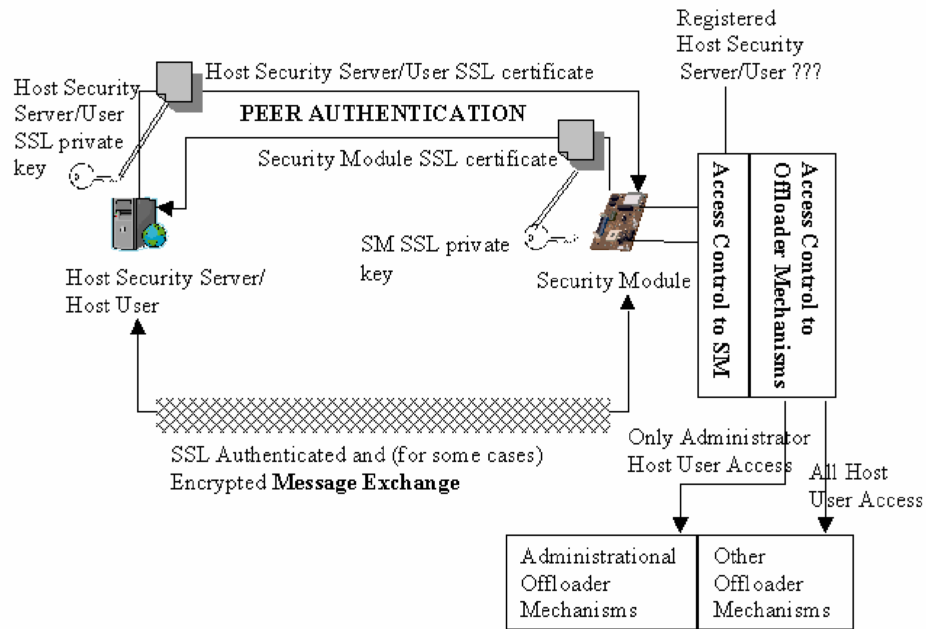

Figure 8 Peer Authentication, Message Exchange and Access Control

After the inclusion of all offloader mechanisms, base cryptography methods and other auxiliary functions they *require*, cryptographic keys and parameters they *use*, and the offloader mechanism access controller server program and communication

infrastructure elements, the overall security architecture becomes as shown in figure 9 below.

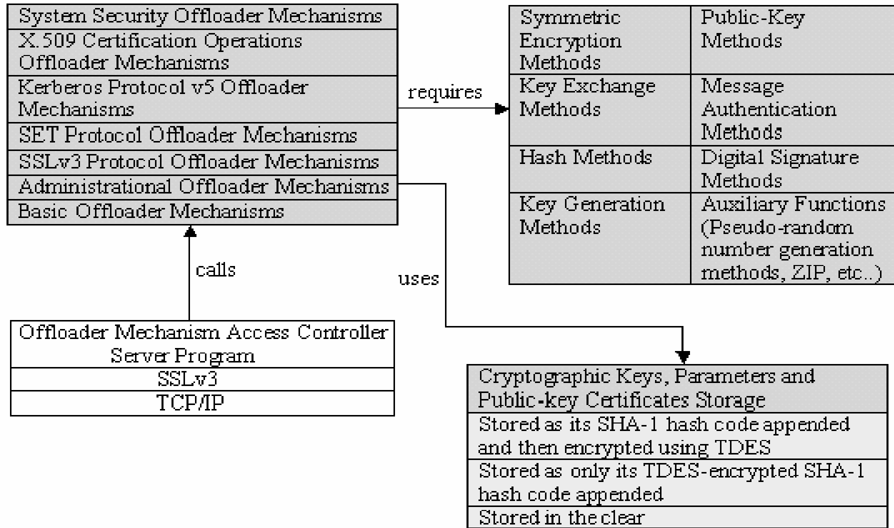

Figure 9 Overall Security Architecture

CHAPTER 4

REFERENCE IMPLEMENTATION

The aim of implementation phase is to demonstrate the practicability of the designed security architecture and its associated concise communications protocol (*even*) on a software-based security module. Hence, a software-based security module was built within the scope of the implementation, in which all operations are performed by the software running on the general-purpose PC/104-plus embedded computer (SBC) without any special cryptography hardware. To achieve the aim, some conventional performance parameters of the built security module were measured.

The built security module implements only the basic offloader mechanisms, defined by the security architecture designed, with their following options:

- "Symmetric_Encryption_Decryption" basic offloader mechanism using DES and AES encryption methods for ECB, CBC (with PKCS padding), CFB and OFB (with unit of transmission value fixed as 8) modes,
- "Asymmetric_Encryption_Decryption" basic offloader mechanism, including all of its six operations, using only RSA method (with PKCS #1 v1.5 padding) for both public-key encryption and signing,
- "Session_Key_Calculation" basic offloader mechanism using only Diffie-Hellman key exchange method,
- "Message_Authentication_Code_Operations" basic offloader mechanism, including all of its six operations, using only HMAC method which can make use of one of MD5, SHA-1 and RIPEMD-160 hash algorithms,
- "Message_Digest_Operations" basic offloader mechanism, including all of its twelve operations, using MD5, SHA-1 and RIPEMD-160 hash algorithms,

- o “Digital_Signature_Calculation” basic offloader mechanism using only DSA digital signature method.

4.1. Reference Algorithms for Basic Offloader Mechanisms

The reference algorithms to implement the basic offloader mechanisms were coded in “C” language, and built linking with the standard “C” language development library (the GNU “C” library glibc) for the chosen OS, namely Linux, and with the SSL library chosen for the implementation, namely OpenSSL. Then, the implemented algorithms were tested to verify that they are working correctly in accordance with the requirements defined by the designed security architecture.

Implemented basic offloader mechanisms were built as a ‘dynamic’ library to be able to upgrade the mechanisms without the necessity to rebuild (and deploy) the offloader mechanism access controller server program, which is linked with the implemented mechanisms.

4.1.1. Symmetric_Encryption_Decryption Basic Offloader Mechanism

The algorithm steps for implementing this offloader mechanism are:

i. first of all, the symmetric encryption key and initialization vector of the host user (on behalf of who the host security server requests execution of this mechanism) are located in the security module storage using the identifiers of the host user and remote user, and using the encryption method, the method’s key length, block length and number of rounds parameters all of which are supplied in the input mechanism execution request message contents,
    - if no such symmetric encryption key can be located, “Encrypt/Decrypt Error” status code is returned as output,
    - if the input encryption mode is not ECB and no such initialization

vector can be located, "Encrypt/Decrypt Error" status code is returned as output,

ii. the located symmetric encryption key and IV of the host user are first decrypted using triple-DES in CBC mode with the storage master key and IV of the module, and then their integrity are verified using their SHA-1 hash codes,
   - if an error occurs during the decryption or integrity verification of the symmetric encryption key or IV, "Encrypt/Decrypt Error" status code is returned as output,

iii. then, according to the requested operation type (encryption or decryption), the input data is encrypted or decrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user,
   - if the input encryption mode is ECB and the length of the input data is not equal to the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
   - if the input encryption mode is CBC and the length of the input data to be *decrypted* is not equal to a multiple of the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
   - if the input encryption mode is CBC, the input data to be *encrypted* is padded using PKCS padding,
   - if the input encryption mode is CBC, after the *decryption* of the input data, the PKCS padding of the decrypted data is verified and the padding bytes are removed if the verification is successful. Otherwise, "Encrypt/Decrypt Error" status code is returned,

iv. finally, the encrypted or decrypted data is returned as the output of the mechanism along with the "OK" status code.

The implementations of DES and AES symmetric encryption methods for ECB, CBC, CFB and OFB modes of operation are provided by the chosen SSL library, OpenSSL. The PKCS padding for CBC mode was implemented separately.

4.1.2. Asymmetric_Encryption_Decryption Basic Offloader Mechanism

The steps of the algorithm for implementing this offloader mechanism are:

i. if the requested operation type involves signing (e.g. "Sign and Encrypt"), the private signature key of the host user is located in the security module storage using the identifier of the host user, the public-key signing method and this method's key size all of which are provided in the input mechanism execution request message contents,
   - if no such private key can be located, "Sign/Verify Error" status code is returned as output,

ii. if the requested operation type involves encryption (e.g. "Encrypt"), the public encryption key certificate of the remote user is located in the SM storage using the identifier of the remote user, the public-key encryption method and this method's key size all of which are provided in the input mechanism execution request message contents,
   - if no such public-key certificate can be located, "Encrypt/Decrypt Error" status code is returned as output,

iii. if the requested operation type involves decryption (e.g. "Decrypt and Verify"), the private encryption key of the host user is located in the SM storage using the input identifier of the host user, public-key encryption method and this method's key size,
   - if no such private key can be found, "Encrypt/Decrypt Error" status code is returned,

iv. if the requested operation type involves verification (e.g. "Verify"), the public signature key certificate of the remote user is located in the SM storage using the input identifier of the remote user, public-key signing method and this method's key size,
   - if no such public-key certificate can be found, "Sign/Verify Error" status code is returned as output,

v. if the requested operation type involves signing or decryption, the located private signature or encryption key of the host user is first decrypted using

triple-DES in CBC mode with the storage master key and IV of the module, and then its integrity is verified using its SHA-1 hash code,

- if an error occurs during the decryption or integrity verification of the private key, "Sign/Verify Error" status code is returned as output if the private key is a signature key, "Encrypt/Decrypt Error" status code is returned if the private key is an encryption key,

vi. afterwards, if the operation type involves signing, the input data is asymmetrically encrypted (signed) using the input public-key signing method and with the private signature key of the host user,

- if the input public-key signing method is RSA, the input data to be signed is padded using PKCS#1 v1.5 padding. Therefore, if the length of the input data (in bytes) is *bigger than the size of the private signature key of the host user minus 12*, "Sign/Verify Error" status code is returned,

vii. if the operation type involves encryption, the input data (*possibly after signed* in the previous step) is asymmetrically encrypted using the input public-key encryption method and with the public encryption key of the remote user contained in her located public encryption key certificate,

- if the input public-key encryption method is RSA, the input data to be encrypted is padded using PKCS #1 v1.5 padding. Therefore, if the length of the input data is bigger than the size of the public encryption key of the remote user minus 12, "Encrypt/Decrypt Error" status code is returned,

viii. if the input operation type involves decryption, the input data is asymmetrically decrypted using the input public-key encryption method and with the private encryption key of the host user,

- if the input public-key encryption method is RSA and the length of the input data is not equal to the size of the private encryption key of the host user, "Encrypt/Decrypt Error" status code is returned,
- if the input public-key encryption method is RSA, during the decryption of the input data, its PKCS #1 v1.5 padding is verified and

the padding bytes are removed if the verification is successful. Otherwise, “Encrypt/Decrypt Error” status code is returned,

ix. if the input operation type involves verification, the input data (*possibly after decrypted* in the previous step) is asymmetrically decrypted using the input public-key signing method and with the public signature key of the remote user contained in her located public signature key certificate,
   - if the input public-key signing method is RSA and the length of the input data is not equal to the size of the public signature key of the remote user, “Sign/Verify Error” status code is returned,
   - if the input public-key signing method is RSA, during the verification of the input data, its PKCS #1 v1.5 padding is verified and the padding bytes are removed if this verification is successful. Otherwise, “Sign/Verify Error” status code is returned,

x. finally, the encrypted, signed, signed and encrypted, decrypted, verified, or decrypted and verified data is returned as output of this mechanism along with the “OK” status code,

The implementation of the RSA public-key method using PKCS #1 v1.5 padding is provided by the OpenSSL library.

4.1.3. Session_Key_Calculation Basic Offloader Mechanism

The steps of the reference algorithm to implement this offloader mechanism are:

i. firstly, the private key exchange key of the host user is located in the security module storage using the identifier of the host user, the public-key key exchange method and this method’s key size all of which are provided in the input mechanism execution request message contents,
   - if such a private key cannot be located, “Key Exchange Error” status code is returned as output,

ii. the located private key exchange key of the host user is first decrypted using triple-DES in CBC mode with the storage master key and IV of the

module, and then its integrity is verified using its SHA-1 hash code,

- if an error occurs during the decryption or integrity verification of the private key, "Key Exchange Error" status code is returned as output,

iii. the global key exchange parameters of the host user are located in the security module storage using the input identifier of the host user, input public-key key exchange method and this method's key size,

- if these parameters cannot be located in the SM storage, "Key Exchange Error" status code is returned as output,

iv. the public key exchange key certificate of the remote user is located in the SM storage using the input identifier of the remote user, input public-key key exchange method and this method's key size,

- if such a public-key certificate cannot be located in the module, "Key Exchange Error" status code is returned as output,

v. afterwards, a session (symmetric) key is computed using the input public-key key exchange method and with the private key exchange key and global key exchange parameters of the host user and the public key exchange key of the remote user contained in her located certificate, *according to the input encryption method key length which refers to the desired length of the session key*,

- if the private key exchange key of the host user and the public key exchange key of the remote user were not generated using the same (host user's) global key exchange parameters, "Key Exchange Error" status code is returned as output,
- if the input public-key key exchange method is Diffie-Hellman and the input encryption method key length is bigger than the input public-key key exchange method (Diffie-Hellman) key size (which determines the *maximum length for the session key* to be computed), "Key Exchange Error" status code is returned as output,

vi. lastly, the computed session key is stored in the SM as its SHA-1 hash code appended and then encrypted using triple-DES in CBC mode with the storage master key and IV of the module; and as associated with the

identifiers of the host and remote users, input encryption method, encryption method key length, block length and number of rounds parameters. “OK” status code is returned as output.

The implementation of the Diffie-Hellman public-key key exchange method is provided by the OpenSSL library.

4.1.4. Message_Authentication_Code_Operations Basic Offloader Mechanism

The steps of the reference algorithm to implement this offloader mechanism are:

i. the MAC secret key of the host user is located in the security module storage using the identifiers of the host user and remote user, the MAC method and the MAC method key size all of which are provided in the input mechanism execution request message contents,
   - if such a MAC secret key cannot be located in the module, “Authenticate/Verify Error” status code is returned as output,

ii. if the requested operation type involves encryption or decryption (e.g. “Authenticate and Encrypt”), the symmetric encryption key and IV of the host user are located in the SM storage using the input identifiers of the host user and remote user, the encryption method, this method’s key length, block length and number of rounds parameters all of which are provided in the input mechanism execution request message contents,
   - if such a symmetric encryption key cannot be located in the module, “Encrypt/Decrypt Error” status code is returned as output,
   - if the input encryption mode is not ECB and no such initialization vector can be located, “Encrypt/Decrypt Error” status code is returned,

iii. the located MAC secret key, symmetric encryption key and IV of the host user are first decrypted using triple-DES in CBC mode with the storage master key and IV of the SM, and then their integrity are verified using their SHA-1 hash codes,
   - if an error occurs during the decryption or integrity verification of the

MAC secret key, "Authenticate/Verify Error" status code is returned; if an error occurs during the decryption or integrity verification of the symmetric encryption key or IV, "Encrypt/Decrypt Error" status code is returned,

iv. afterwards, if the requested operation type is "Authenticate" or "Authenticate and Encrypt", the input data is authenticated by appending it a message authentication code computed using the input MAC method and hash method and with the MAC secret key of the host user,

v. if the requested operation type is "Authenticate and Encrypt", the input data, after authenticated in the previous step, is symmetrically encrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user,

- if the input encryption mode is ECB and the length of the input data (*after authentication*) is not equal to the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC, the input data (after authentication) is padded using PKCS padding before encryption,

vi. if the requested operation type is "Encrypt and Authenticate", the input data is first symmetrically encrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user, and then the encrypted input data is authenticated by appending it a MAC computed using the input MAC method and hash method and with the MAC secret key of the host user,

- if the input encryption mode is ECB and the length of the input data to be encrypted is not equal to the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned as output,
- if the input encryption mode is CBC, the input data is padded using PKCS padding before encryption,

vii. if the requested operation type is "Verify" or "Verify and Decrypt", the input data is verified by verifying the message authentication code at its

end using the input MAC method and hash method and with the MAC secret key of the host user,

- if the input MAC method is HMAC, there must exist a MAC at the end of the input data with *length equal to the output length of the input hash method*, otherwise "Authenticate/Verify Error" status code is returned,
- if the verification of the MAC of the input data is successful, the MAC is thrown off from the input data,

viii. if the requested operation type is "Verify and Decrypt", the input data, after verified in the previous step, is symmetrically decrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user,

- if the input encryption mode is ECB and the length of the input data (*after verification*) is not equal to the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC and the length of the input data (*after verification*) is not equal to a multiple of the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC, after the decryption of the input data, its PKCS padding is verified and the padding bytes are removed if this verification is successful. Otherwise, "Encrypt/Decrypt Error" status code is returned,

ix. if the requested operation type is "Decrypt and Verify", the input data is first symmetrically decrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user, and then the decrypted input data is verified by verifying the MAC at its end using the input MAC method and hash method and with the MAC secret key of the host user,

- if the input encryption mode is ECB and the length of the input data is not equal to the block length of the input encryption method,

"Encrypt/Decrypt Error" status code is returned as output,

- if the input encryption mode is CBC and the length of the input data is not equal to a multiple of the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC, after the decryption of the input data, its PKCS padding is verified and the padding bytes are removed if this verification is successful. Otherwise, "Encrypt/Decrypt Error" status code is returned,
- if the input MAC method is HMAC, there must exist a MAC at the end of the *decrypted* input data with length equal to the output length of the input hash method, otherwise "Authenticate/Verify Error" status code is returned,
- if the verification of the MAC of the decrypted input data is successful, the MAC is thrown off from the decrypted input data,

x. finally, the authenticated, authenticated and encrypted, encrypted and authenticated, verified, decrypted and verified, or verified and decrypted data is returned as output of this offloader mechanism along with the "OK" status code,

The implementations of the HMAC message authentication method and MD5, SHA-1, RIPEMD-160 hash methods are provided by the OpenSSL library.

4.1.5. Message_Digest_Operations Basic Offloader Mechanism

The steps of the reference algorithm to implement this offloader mechanism are:

i. if the requested operation type involves encryption or decryption (e.g. "Authenticate and Encrypt") or is "Authenticate" or "Verify", the symmetric encryption key and IV of the host user are located in the SM storage using the identifiers of the host user and remote user, the encryption method, this method's key length, block length and number of rounds parameters all of which are provided in the input mechanism

execution request message contents,

- if such a symmetric key cannot be located, “Encrypt/Decrypt Error” status code is returned as output,
- if the input encryption mode is not ECB and no such IV can be located, “Encrypt/Decrypt Error” status code is returned as output,

ii. if the requested operation type involves signing operation (e.g. “Sign”), the private signing key of the host user is located in the SM storage using the input identifier of the host user, the public-key signing method and this method’s key size both of which are provided in the input mechanism execution request message contents,

- if such a private key cannot be located in the SM storage, “Sign/Verify Signed Data Error” status code is returned as output,

iii. if the requested operation type involves verification of signed data (e.g. “Verify Signed Data”), the public signing key certificate of the remote user is located in the SM storage using the input identifier of the remote user, public-key signing method and this method’s key size,

- if no such public-key certificate can be located in the module storage, “Sign/Verify Signed Data Error” status code is returned,

iv. if the requested operation type involves authentication or verification using hash secret value (e.g. “Authenticate with Secret Value”), the hash secret value of the host user is located in the SM storage using the input identifiers of the host user and remote user,

- if no such hash secret value can be located in the module storage, “Authenticate/Verify Error” status code is returned,

v. then, the located symmetric encryption key and IV, private signing key and hash secret value of the host user are first decrypted using triple-DES in CBC mode with the storage master key and IV of the module, and then their integrity are verified using their SHA-1 hash codes,

- if an error occurs during the decryption or integrity verification of the symmetric encryption key or IV, “Encrypt/Decrypt Error” status code is returned; if an error occurs during the decryption or integrity

verification of the private signing key, “Sign/Verify Signed Data Error” status code is returned; if an error occurs during the decryption or integrity verification of the hash secret value, “Authenticate/Verify Error” status code is returned,

vi. afterwards, if the requested operation type is “Authenticate”, “Authenticate and Encrypt”, “Sign” or “Sign and Encrypt”, the message digest of the input data is computed using the input hash method, and appended to the input data,

vii. if the requested operation type is “Authenticate with Secret Value” or “Authenticate with Secret Value and Encrypt”, the message digest of the *input data and hash secret value of the host user combined* is computed using the input hash method, and appended to the input data,

viii. if the requested operation type is “Sign” or “Sign and Encrypt”, the message digest of the input data, computed and appended to the input data in step vi., is asymmetrically encrypted (signed) using the input public-key signing method and with the private signing key of the host user,

- if the input public-key signing method is RSA, the message digest is padded using PKCS #1 v1.5 padding during signing. Therefore, if the length (in bytes) of the message digest is bigger than the size of the private signing key of the host user minus 12, “Sign/Verify Signed Data Error” status code is returned as output,

ix. if the requested operation type is “Authenticate”, the message digest of the input data, computed and appended to the input data in step vi., is symmetrically encrypted using the input encryption method and encryption mode with the symmetric encryption key and IV of the host user,

- if the input encryption mode is ECB and the length of the message digest is not equal to the block length of the input encryption method, “Authenticate/Verify Error” status code is returned as output,
- if the input encryption mode is CBC, the message digest is padded using PKCS padding before encryption,

x. if the requested operation type is “Authenticate and Encrypt”, “Sign and

Encrypt" or "Authenticate with Secret Value and Encrypt", the input data and the message digest, computed and appended to the input data in previous steps (and *maybe signed*), is symmetrically encrypted using the input encryption method and encryption mode with the symmetric encryption key and IV of the host user,

- if the input encryption mode is ECB and the length of the input data along with the message digest is not equal to the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC, the input data along with the message digest is padded using PKCS padding before encryption,

xi. if the requested operation type is "Verify", the encrypted message digest at the end of the input data is symmetrically decrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user,

- if the input encryption mode is ECB, there must exist an encrypted message digest at the end of the input data *with length equal to the block length of the input encryption method*, otherwise "Authenticate/Verify Error" status code is returned as output,
- if the input encryption mode is CBC, there must exist an encrypted message digest at the end of the input data *with length equal to the output length of the input hash method complemented to a multiple of the block length of the input encryption method*, otherwise "Authenticate/Verify Error" status code is returned,
- if the input encryption mode is CFB or OFB, there must exist an encrypted message digest at the end of the input data *with length equal to the output length of the input hash method*, otherwise "Authenticate/Verify Error" status code is returned,
- if the input encryption mode is CBC, after the decryption of the encrypted message digest, its PKCS padding is verified and the padding bytes are removed if this verification is successful. Otherwise,

"Authenticate/Verify Error" status code is returned,

xii. if the requested operation type is "Decrypt and Verify", "Decrypt and Verify Signed Data" or "Decrypt and Verify with Secret Value", the input data is symmetrically decrypted using the input encryption method and encryption mode and with the symmetric encryption key and IV of the host user,

- if the input encryption mode is ECB and the length of the input data is not equal to the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC and the length of the input data is not equal to a multiple of the block length of the input encryption method, "Encrypt/Decrypt Error" status code is returned,
- if the input encryption mode is CBC, after the decryption of the input data, its PKCS padding is verified and the padding bytes are removed if this verification is successful. Otherwise, "Encrypt/Decrypt Error" status code is returned,

xiii. if the requested operation type is "Verify Signed Data" or "Decrypt and Verify Signed Data", the signed message digest at the end of the input data (*possibly decrypted* in the previous step) is first asymmetrically decrypted using the input public-key signing method and with the public signing key of the remote user contained in her located certificate,

- if the input public-key signing method is RSA, there must exist a signed message digest at the end of the input data with length equal to the size of the public signing key of the remote user, otherwise "Sign/Verify Signed Data Error" status code is returned,
- if the input public-key signing method is RSA, during asymmetric decryption of the signed message digest, its PKCS#1 v1.5 padding is verified and the padding bytes are removed if this verification is successful. Otherwise, "Sign/Verify Signed Data Error" is returned,

xiv. if the requested operation type is "Verify", "Decrypt and Verify", "Verify Signed Data" or "Decrypt and Verify Signed Data", the message digest

(*possibly recovered* in step xi. or step xiii.) at the end of the input data (*possibly decrypted* in step xii.) is verified using the input hash method,

- if the requested operation type is "Verify", "Verify Signed Data" or "Decrypt and Verify Signed Data", and if the length of the message digest, *as recovered* in step xi. or step xiii., is not equal to the output length of the input hash method, "Authenticate/Verify Error" status code is returned,
- if the requested operation type is "Decrypt and Verify", there must exist a message digest at the end of the input data (decrypted in step xii.) with length equal to the output length of the input hash method, otherwise "Authenticate/Verify Error" status code is returned,
- if the verification of the message digest is successful, the message digest is thrown off from the input data,

xv. if the requested operation type is "Verify with Secret Value" or "Decrypt and Verify with Secret Value", the message digest at the end of the input data (possibly decrypted in step xii.) is verified using the input hash method and the hash secret value of the host user,

- there must exist a message digest at the end of the input data with length equal to the output length of the input hash method, otherwise "Authenticate/Verify Error" status code is returned,
- if the verification of the message digest is successful, the message digest is thrown off from the input data,

xvi. finally, the authenticated, authenticated and encrypted, signed, signed and encrypted, verified, or decrypted and verified data is returned as the output of this offloader mechanism along with the "OK" status code.

4.1.6. Digital_Signature_Calculation Basic Offloader Mechanism

The steps of the reference algorithm to implement this offloader mechanism are:

i. if the requested operation type is "Sign", the private signing key of the host user is located in the security module storage using the identifier of the host

user, public-key signing method and this method's key size all of which are provided in the input mechanism execution request message contents,

- if such a private key cannot be located in the SM storage, "Sign/Verify Error" status code is returned as output,

ii. if the requested operation type is "Verify", the public signing key certificate of the remote user is located in the SM storage using the input identifier of the remote user, public-key signing method and this method's key size,

   - if such a public-key certificate cannot be located in the module storage, "Sign/Verify Error" status code is returned as output,

iii. the global signing parameters of the host user are located in the SM storage using the input identifier of the host user, public-key signing method and this method's key size,

   - if the input public-key signing method is DSA and these parameters cannot be located, "Sign/Verify Error" status code is returned,

iv. the located private signing key of the host user is first decrypted using triple-DES in CBC mode with the storage master key and IV of the module, and then its integrity is verified using its SHA-1 hash code,

   - if an error occurs during the decryption or integrity verification of the private signing key, "Sign/Verify Error" status code is returned,

v. afterwards, if the requested operation type is "Sign", the input data is signed using the input public-key signing method and with the private signing key and global signing parameters of the host user to obtain the signature of the input data, which is then appended to the input data,

   - if the input public-key signing method is DSA and the private signing key of the host user was not generated using her global signing parameters, "Sign/Verify Error" status code is returned as output,

vi. if the requested operation type is "Verify", the signature at the end of the input data is verified using the input public-key signing method and with the public signing key of the remote user contained in her located certificate and with global signing parameters of the host user,

- if the input public-key signing method is DSA and the public signing key of the remote user was not generated using the global signing parameters of the host user, "Sign/Verify Error" status code is returned,
- if the input public-key signing method is DSA, there must exist a signature at the end of the input data with 48 bytes length (required by DSA),
- if the verification of the signature is successful, it is thrown off from the input data,

vii. finally, the signed or verified data is returned as the output of this offloader mechanism along with the "OK" status code,

The implementation of the DSA public-key digital signature method is provided by the OpenSSL library.

4.2. Offloader Mechanism Access Controller Server Program

The offloader mechanism access controller server program was also coded in "C" language and built linking with the GNU "C" library glibc, and the OpenSSL library. The server program was, in addition, linked with the basic offloader mechanisms dynamic library. The program is multi-threaded and consists of a main program thread and dedicated threads for each host security server/host user connection. The program uses the pthreads library, included in the glibc library, to have the multi-threading functionality.

The main program thread

i. locates the storage master key and IV of the security module in the module storage and reads them to memory,

ii. initializes the SSL context by
   - setting the context cipher list to "NULL-SHA" for performing *SSL RSA key exchange* with host security servers when connecting to the

SM, and for protecting the integrity of the messages exchanged between the SM and connected host security servers with *SHA-1* hash method,

  - loading the security module's SSL public-key certificate into the context to enable host security servers to *identify* the SM (*peer authentication*) when connecting to the SM,

iii. waits for and accepts a TCP/IP connection from a host security server,

iv. tries to set up an SSL connection ("SSL_connect") with the TCP/IP-connected host security server,

  - if the SSL public-key certificate of the host user (on behalf of who the host security server will request offloader mechanism executions) cannot be obtained by the SM during the SSL handshake to *identify* the host user (*peer authentication*), the main program thread disconnects the host security server from the module and resumes its execution from step iii.,
  - if the SSL public-key certificate of the host user, obtained during the SSL handshake, is not signed by a trusted (accepted as trusted by the security module) CA, the main program thread disconnects the host security server from the SM and resumes its execution from step iii.,

v. extracts the user information of the host user from her SSL public-key certificate, and searches in the user-role list of the module whether the host user is a registered module user (*access control to the security module*),

  - before the stored user-role list is searched, the integrity of the list is verified by first decrypting the encrypted SHA-1 hash code at its end using triple-DES in CBC mode with the storage master key and IV of the SM, and then verifying the recovered hash code using the SHA-1 hash code of the list. If an error occurs during the integrity verification of the user-role list, the main program thread finishes execution of the server program,
  - if the host user is not a registered module user, the main program thread disconnects the host security server from the module and

resumes its execution from step iii.,

vi. allocates and starts to run a dedicated thread for processing offloader mechanism execution requests from the host security server/host user; and passes to the dedicated thread user information of the host user,

vii. decides whether the quota of connections (it is determined as 64 after performance tests) to the security module is filled up,
   - if the quota of connections is filled up, the main program thread waits to deallocate dedicated threads which have already finished their execution,

viii. returns to step iii.

A dedicated thread

i. waits for and receives (blocking "SSL_read") an offloader mechanism execution request message from the SSL-connected host security server,
   - if there is an error occurred during the "SSL_read" or the host security server closed its SSL connection to the SM, the dedicated thread disconnects the host security server from the SM and then finishes its execution,

ii. decides which offloader mechanism is requested by the host security server from the "Operation Type" field of the received mechanism execution request message; and determines whether the host user is allowed to request this offloader mechanism using her user information (*access control to offloader mechanisms*),
   - if the host user is not allowed to request this offloader mechanism, "Access Control Error" status code is returned and the execution of the dedicated thread resumes from step i.,

iii. calls the requested offloader mechanism passing the contents of the mechanism execution request message and user information of the host user as parameters (passing of the user information to the offloader mechanism enables the mechanism to verify that the host user supplied her identifier and noone else's in the mechanism execution request message),

iv. gets the output parameters returned by the executed offloader mechanism, prepares an offloader mechanism execution response message from these parameters, and sends (blocking "SSL_write") this response message back to the host security server,
   - if there is an error occurred during the "SSL_write" or the host security server closed its SSL connection to the SM, the dedicated thread disconnects the host security server from the SM and then finishes its execution,

v. returns to step i.

4.3. Communication Infrastructure - SSL Library

The OpenSSL 0.9.7d library is used to provide the implementation of the SSLv3 protocol. Also, the OpenSSL library provides base cryptography methods and other auxiliary functions (e.g. PEM-formatted private key file read) required by offloader mechanisms. Moreover, the tools and functions provided by the OpenSSL library were used to generate all sample symmetric encryption keys, IVs, private keys, public-key certificates, MAC secret keys, global public-key parameters, hash secret values, storage master key and IV of the SM, SSL public-key certificates etc.., needed for the implementation.

SSL RSA key exchange was used during the SSL handshake to connect a host security server to the security module. The reason for this is that SSL RSA key exchange does not involve time-consuming Diffie-Hellman session key computation contrary to SSL fixed DH and ephemeral DH key exchange schemes. During the SSL handshake, the security module sends its *SSL (RSA) public-key certificate*, which contains its identity information and signed by a trusted CA, to the host security server to enable the server to *identify* the SM; and also requests from the host security server the *SSL (RSA) public-key certificate* of the host user, containing her user information and signed by a trusted CA, to *identify* the host

user. By this way, both the host security server/host user and the security module verify each other's identity.

For authentication and integrity-protection of the messages exchanged between the security module and a host security server, SSLv3 MAC method was used with SHA-1 hash method (with 160-bit output length **[16]**). The messages exchanged are *always* authenticated and integrity-protected.

For encryption of the messages exchanged, the strongest symmetric encryption method allowed by SSLv3, triple-DES with 168-bit key length **[16]**, would be used, but the basic offloader mechanisms implemented do not involve exchange of secret keys or data between the SM and a host security server hence do not require encrypted message exchange.

4.4. Embedded Linux OS

Qplus embedded Linux development toolkit was used to build an embedded Linux OS that will run on the embedded SBC of the built security module. Both the kernel and root filesystem of the built embedded Linux OS include only the definitely needed features and libraries not to use up a large part of the storage space of the embedded computer. Also, only the required kernel modules and programs are configured to load and run at the startup of the OS to ensure that the running OS has a small memory footprint on the embedded computer.

Some important features included in the built Linux kernel (version 2.4.17) are mentioned below along with the reasons why they were included:

- since the main server program (offloader mechanism access controller) will communicate with host security servers via TCP/IP protocol, built kernel included the TCP/IP networking protocols and Unix domain sockets implementation,

- since the embedded computer has a Realtek®RTL8139D 10/100 Mbps PCI Ethernet controller as its networking device, built Linux kernel included network device support, Ethernet support and the driver for this Ethernet controller,
- since cryptographic keys, parameters and other security-related data belonging to host and remote users, and the security module internal data (including its signed firmware) will be stored in the storage device of the embedded computer, which is a Type I CompactFlash® Card working over EIDE interface, built Linux kernel included ATA/(E)IDE/ATAPI low cost mass storage units support and EIDE disk/cdrom/floppy support,
- since administrator users of the security module may transfer cryptographic keys, parameters and other security-related data to/from the SM *manually* via USB storage devices (e.g. flash disks), built Linux kernel included support for hot-pluggable devices, USB support, Memory Technology Device support (for flash chips and solid state devices), and Common Flash Interface support (for detection of flash chips),
- loadable kernel module support is enabled in the built kernel to be able to dynamically load kernel modules at runtime when they are actually needed,
- since the embedded SBC is a PC/104-plus CPU board having both PCI and ISA buses, support for PCI and ISA bus hardware, and Plug and Play support (for configuring ISA devices via software) are included,
- support for terminal devices (display and keyboard) is included together with VGA text console support, to enable administrator users of the SM to add, remove, modify kernel modules, programs or other files to/from the OS manually from the console in text mode.

Some important software libraries and other features included in the built root filesystem, which was formatted as an ext3 filesystem, are mentioned below:

- GNU “C” library, glibc 2.3.3, is included whose various libraries are used by the OpenSSL library, main server program and basic offloader mechanisms,

- Basic "/etc" files and initialization scripts are included to enable configuring the network interface and programs that will run at the startup of the OS after the kernel is loaded,
- Busybox software package is included which contains a small version of the bash shell "ash", and various useful command-line utilities like "insmod" for dynamically loading kernel modules, "chown" for adjusting ownership of folders and files, "ifconfig" for configuring the network interface, "rm", "cp", "vi", "syslogd", "mount", "rsync", "ping", etc..
- Tinylogin software package is included which contains command-line utilities "adduser", "addgroup", "deluser", "delgroup" for adding and deleting users and groups, a small login program "login", a utility for changing user passwords "passwd", "su", etc..
- Procps software package is included which contains "ps", "top", "kill" and other command-line tools for listing, killing and adjusting priorities of running processes. The "ps" tool was also used in performance testing of the built security module,
- Lilo bootloader is included for loading the Linux OS when the security module is booted,
- Support for ISO 9660 CD-ROM filesystem is included which will enable copying of files in a CD to the security module.

The implemented main server program and basic offloader mechanisms, generated sample cryptographic keys, parameters and public-key certificates, and the OpenSSL library were all copied onto the root filesystem after it was customized. Then, an initialization script was created in the "/etc/rc.d/init.d" directory of the root filesystem to run the main server program as a system daemon at the startup of the OS after the kernel is loaded. Finally, the built kernel image and root filesystem were transferred to the embedded SBC using the Etherboot protocol, which makes use of the DHCP and TFTP protocols **[31]**, via the Ethernet.

4.5. Embedded Computer

The embedded SBC chosen for the implementation is the PCM-3370F PC/104-plus CPU board manufactured by Advantech Corporation. Refer to the related section in the "Literature Survey" chapter for its detailed specifications. Some important features of the SBC that led the writer to choose it are mentioned below.

First of all, PC/104 has been the most commonly used embedded board standard for years, and PC/104 boards have small size (96 x 115 mm) considering their low cost. Also, most PC/104 boards are made with standard desktop and laptop chipsets, hence they work fine under Linux especially without the need to write device drivers. **[32]**

Since the embedded computer is a PC/104-plus board, it has a PCI bus, in addition to the old ISA bus, that supports faster data transfers to/from peripheral devices and new easy-to-configure peripheral devices. **[32]**

More importantly, the embedded board has a 10/100 Mbps PCI Ethernet controller, which implies more or less speedy communication and robust data transfers under TCP/IP between the *network-attached* security module and host security servers. Also, the embedded board does not have any special hardware for accelerating certain cryptography operations, therefore it conforms to the aim of building a *software-based* security module.

The SBC has a sufficiently powerful Celeron processor to perform complex cryptography operations (like RSA) fast. In addition, a 256 MB SDRAM memory was purchased and plugged on the board. They together are adequate for both running the Linux kernel smoothly and handling many connection and offloader mechanism execution requests from a number of clients simultaneously.

A 1 GB Type I CFC was purchased and plugged on the embedded board, that provides storage with sufficient size for storing cryptographic keys, parameters and other security-related data of a number of host and remote users. Assuming a user has cryptographic keys, parameters and other security-related data, which have on average the size of 64 1024-bit public keys; those of more than 100.000 users can be stored on the CFC.

Moreover, the SBC has two USB ports which enable an administrator host user to transfer cryptographic keys, parameters and other security-related data, programs or other files to/from the security module manually via USB flash disks.

4.6. Implementation Issues About Client-side API Functions

All client-side API functions were developed for Linux OS, which requires host security servers to use Linux OS to call these functions. They were coded in "C" language and built linking with glibc (version 2.3.3) library and OpenSSL (version 0.9.7d) library, which requires host security servers to have both of these libraries with the same or higher versions installed.

Client-side API functions were built as a 'dynamic' library to enable many client programs of a host security server share a single instance of the client-side API functions library and to be able to upgrade client-side API functions without the need to rebuild the client programs linked with the client-side API functions library.

Two client-side API functions were created for each basic offloader mechanism: one is "Send_*the name of the basic offloader mechanism*" and the other is "Get_*the name of the basic offloader mechanism*". Apart from these, two important client-side API functions "Connect_to_SM" and "Disconnect_from_ SM" were created.

"Connect_to_SM" client-side API function can be called by a client program to connect to a security module with two input parameters:

- local directory path for the host user's SSL public-key certificate (which will be sent to the SM during the SSL handshake),
- IP address of the security module to be connected to.

"Disconnect_from_SM" client-side API function can be called by a client program to cleanly disconnect from a security module with an input parameter which points to the SSL connection data structure returned by the "Connect_to_SM" client-side API function called before by the client program to connect to this security module.

A "Send_*the name of the basic offloader mechanism*" client-side API function can be called by a client program to prepare and send ("SSL_write") to the security module the offloader mechanism execution request message to request execution of the related basic offloader mechanism. Input parameters to this client-side API function are the same as the fields of the offloader mechanism execution request message sent to the SM by calling this function. The "SSL_write" call in this function is *non-blocking*, preventing the client program from waiting idly for the security module to finish its processing and accept the "SSL_write".

A "Get_*the name of the basic offloader mechanism*" client-side API function can be called by a client program to get ("SSL_read") and process the offloader mechanism execution response message returned by the security module as a result of executing the related basic offloader mechanism. Output parameters of this client-side API function are the same as the fields of the offloader mechanism execution response message received from the SM by calling this function. The "SSL_read" call in this function is also *non-blocking*, which prevents the client program from waiting idly for the security module to finish its processing of the requested basic offloader mechanism and send the response message.

4.7. Performance Tests

Conventionally, four parameters are considered more important than others in evaluating the performance of security modules. First one is the number of executions of a symmetric encryption method, usually DES and maybe AES, by the security module per second. Second one is similar but concerns the number of executions of a public-key signing method, usually RSA, by the security module per second. Third parameter is the amount of memory space used by the software of the security module at runtime. Similarly, last parameter is the percentage of CPU power consumed by the SM software at runtime. A less important performance parameter concerns the amount of disk space used up by the SM software that should be small enough for the SM software to run directly from the local disk of the module.

Both the security architecture and the accompanying communications protocol are designed considering these performance parameters. Simple, easy-to-process, yet sufficiently inclusive communications protocol helps increase number of executions of any offloader mechanism, including the ones for symmetric encryption and public-key signing, by the security module per second. It also contributes to reducing the general memory space usage and CPU power consumption of the security module, and to fast serving of the requests.

Different methods were used for measuring the mentioned four performance parameters. To estimate the number of executions of DES and AES symmetric encryption operations by the security module per second, a client program was designed which:

i. records the current time after first achieving connection to the SM,
ii. requests execution of the "Symmetric_Encryption_Decryption" basic offloader mechanism from the SM by calling the "Send_Symmetric_ Encryption_Decryption" client-side API function with the input encryption method parameter set to DES or AES, encryption mode parameter set to

CBC, encryption method key length parameter set to *192* for AES, and each time with varying lengths of input data to be encrypted: 20, 100, 200, 500, 1000-bytes,

- if the security module is busy and cannot handle the mechanism execution request, the program requests execution of the mechanism again,

iii. tries to get the result of the requested basic offloader mechanism by calling the "Get_Symmetric_Encryption_Decryption" client-side API function,

- if the security module is busy with processing of the requested mechanism and cannot return the result, the program tries to get the result again,
- if the returned result indicates that an error occurred during the execution of the offloader mechanism, the client program terminates,

iv. the number of successfully completed "Symmetric_Encryption_ Decryption" basic offloader mechanism executions is increased by one,

v. takes the current time and subtracts it from the recorded starting time to determine if 10 seconds have just passed since the starting time of observation,

- if 10 seconds have not yet passed since the starting time, the execution of the program resumes from step ii.,

vi. calculates the number of executions of the "Symmetric_Encryption_ Decryption" basic offloader mechanism per second approximately by dividing the total number of successfully completed mechanism executions by 10,

vii. finally, disconnects from the security module.

To estimate the number of executions of RSA public-key signing operation by the security module per second, the same client program described above was used with minor modifications. The execution of the "Asymmetric_Encryption_ Decryption" basic offloader mechanism is requested from the SM by calling the "Send_Asymmetric_Encryption_ Decryption" client-side API function with the

input public-key signing method parameter set to RSA, public-key signing method key length parameter set to *1024* or *2048*, and each time with varying lengths of input data to be signed: 20, 50, 100-bytes for 1024-bit key, and 20, 100, 200-bytes for 2048-bit key. Also, the result of the executed offloader mechanism is obtained by calling the "Get_Asymmetric_Encryption_ Decryption" client-side API function.

For each of these two performance parameters, the measurements were performed each time with an *increasing number* of simultaneous client connections to the security module pushing the limits of the SM. Each connected client runs the same client program above.

The observation time is determined as 10 seconds (instead of 1) to enable more accurate measurements by making the difference between the actual observation time (e.g. 10.05 sec.) and observation time (10 sec.) negligible.

To measure both the amount of memory space used and the percentage of CPU power consumed by the module software (Linux OS and application software), two programs, which run at the client side, and a program, which run at the module side, were developed. The cryptography operation, which uses the most amount of memory space and consumes the most percentage of CPU power, is after some experiments, decided to be public-key signing and then encryption with RSA.

The structure of the first client-side program looks like the structure of the client program used for estimating the first two performance parameters. This program repeatedly requests execution of the "Asymmetric_Encryption_Decryption" basic offloader mechanism from the SM by calling the "Send_Asymmetric_Encryption_ Decryption" client-side API function with the input operation type parameter set to "*Sign and Encrypt*", public-key signing method and public-key encryption method parameters set to RSA, public-key signing method key length set to *1024*, public-

key encryption method key length set to *2048*, and with a *100*-byte length input data to be signed and then encrypted. Also, the observation time is determined as 5 seconds, because this is long enough for the program, running at the module side, to measure memory usage and CPU power utilization values of the module software sufficient number of times.

The other client-side program is executed in parallel with the client-side program described above and:

i. connects to the program running at the module side via TCP/IP,
ii. reads (blocking read) physical and virtual memory space usage amounts and CPU power utilization percents of the OS and application software (main server program) of the security module separately, which are measured and sent by the module-side program,
    - if the module-side program has closed the connection, the program disconnects from the SM and finishes its execution,
iii. updates maximum values of physical and virtual memory space usage amounts and CPU power utilization percents of the OS and application software separately, if the read amounts/percents are greater than maximum values.

The program running at the module side is executed in parallel with the main server program and:

i. waits for and accepts TCP/IP connection from the above reader client-side program,
ii. gathers physical (non-swapped) and virtual memory space usage amounts and CPU power utilization percentages information of the running OS and application software separately by using the ‘ps’ utility provided by the embedded Linux,
iii. transmits them (blocking write) to the reader client-side program via the TCP/IP connection,
iv. determines if all dedicated threads, created by the main server program for

each connected client, have finished their executions,

- if a dedicated thread still continues its execution, the execution of the program resumes from step ii.,

v. disconnects from the reader client-side program and finishes its execution.

To determine if all dedicated threads, created by the main server program for each connected client, have finished their executions, Linux shared memory implementation is used. When a dedicated thread starts its execution, it sets its part in the shared memory created by the module-side program described above. When a dedicated thread finishes its execution, it unsets its part in the shared memory. The module-side program scans all parts of the shared memory in step iv. above, if it finds a part that is set, it decides that a dedicated thread still continues its execution.

Also for these two performance parameters, the measurements were performed each time with an increasing number of simultaneous client connections to the security module pushing its limits. Each connected client runs the same client program, which is the first client-side program.

For all performance parameters, to ensure reliability of measurements, each measurement was run three times and the resulting values from three runs were averaged.

As the last performance parameter, the amounts of disk space used up by the Linux kernel and root filesystem deployed to the SM were observed from the Qplus toolkit.

In the figures below, the results of measurements for each performance parameter, for **100**-bytes of input data, are shown. Same results are also listed in tables in Appendix B. The figures showing the measurements results for other lengths (20, 50, 200, 500, 1000-bytes) of input data for the first two performance parameters

are given in Appendix B. Table 1, below the figures, lists disk space usage amounts for the kernel and root filesystem of the built SM.

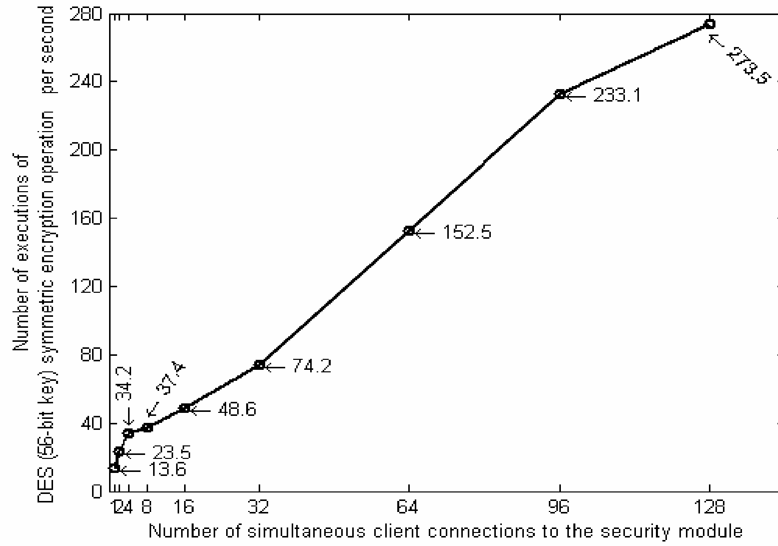

Figure 10 Performance Measurements Results - DES Symmetric Encryption, 100-bytes of Input Data

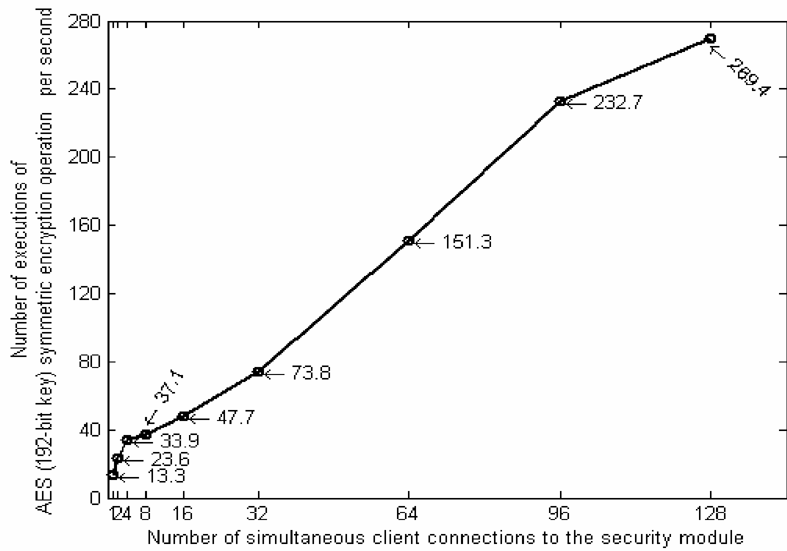

Figure 11 Performance Measurements Results - AES Symmetric Encryption, 100-bytes of Input Data

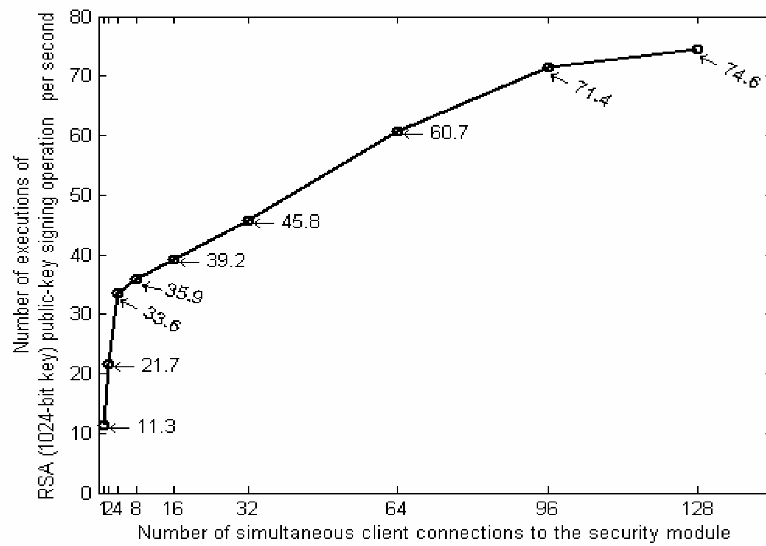

Figure 12 Performance Measurements Results - RSA 1024-bit Signing, 100-bytes of Input Data

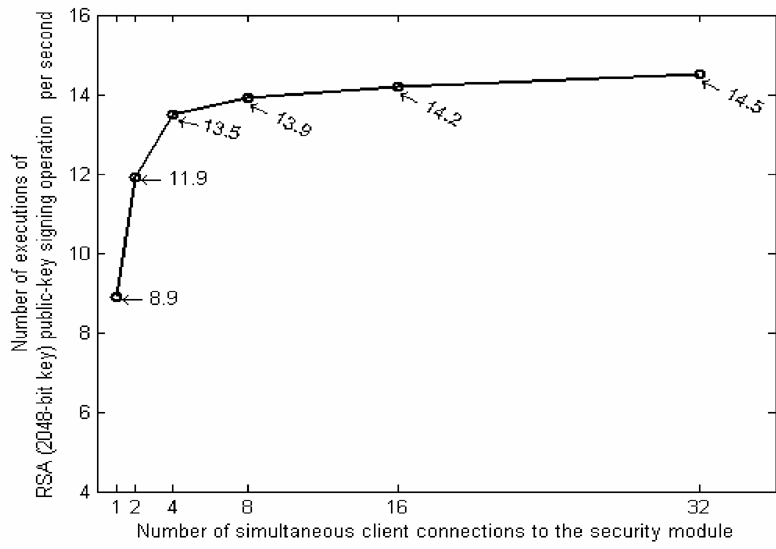

Figure 13 Performance Measurements Results - RSA 2048-bit Signing, 100-bytes of Input Data

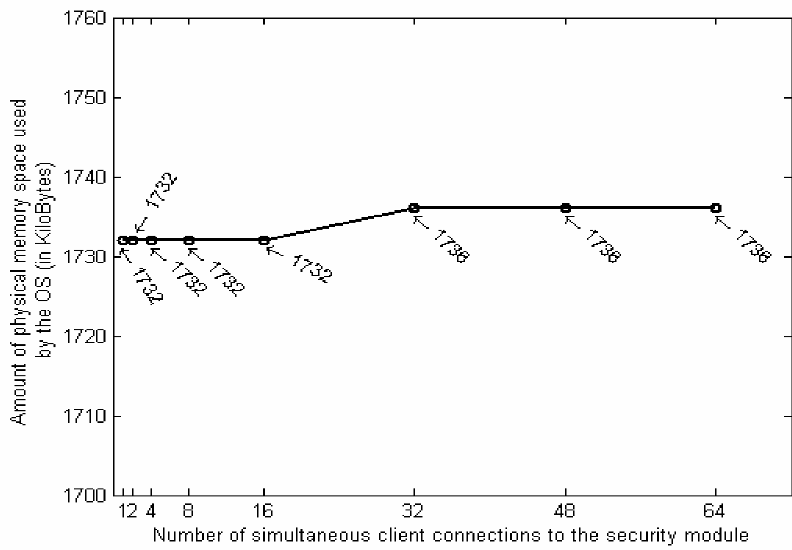

Figure 14 Performance Measurements Results - Physical Memory Space Usage by the OS

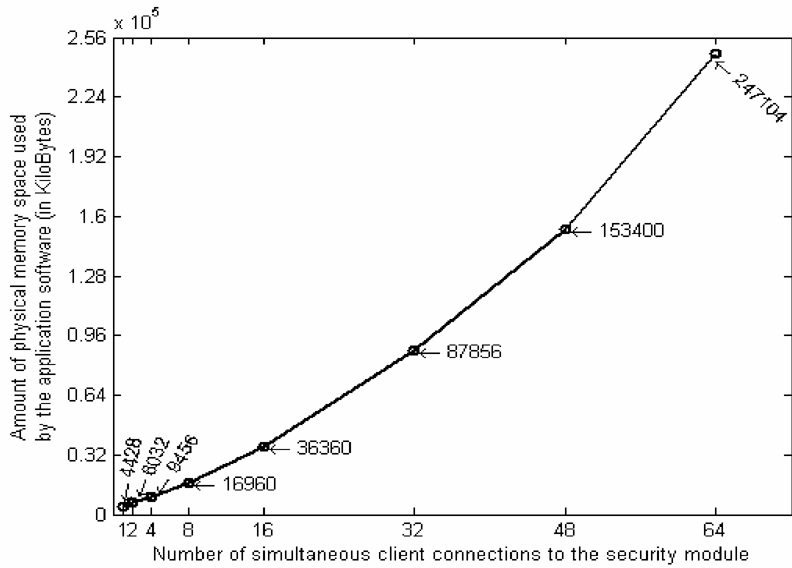

Figure 15 Performance Measurements Results - Physical Memory Space Usage by the Application Software

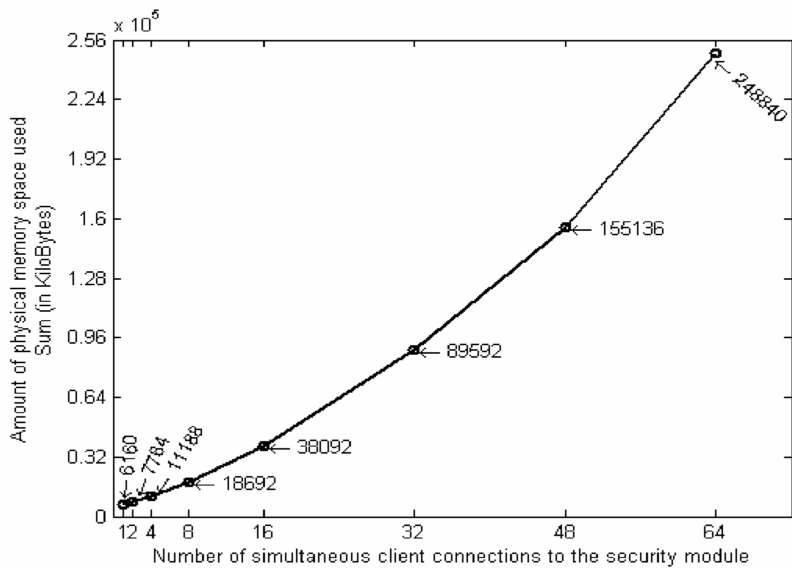

Figure 16 Performance Measurements Results - Physical Memory Space Usage Sum

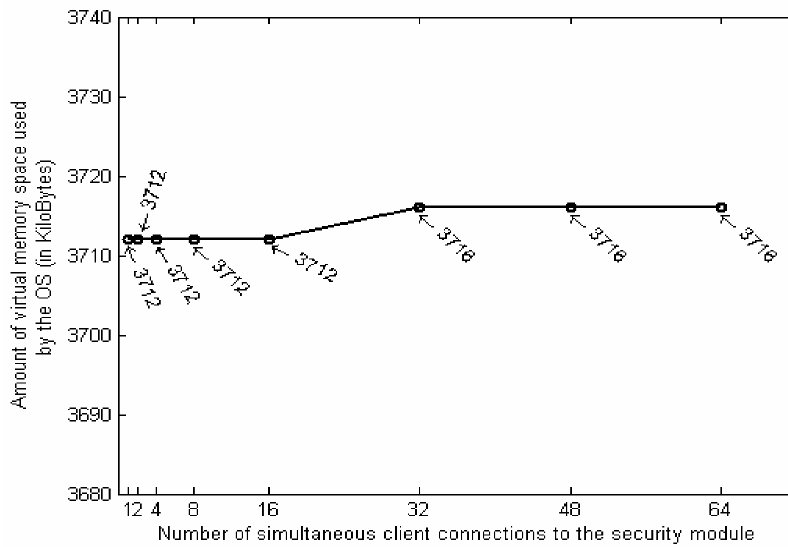

Figure 17 Performance Measurements Results - Virtual Memory Space Usage by the OS

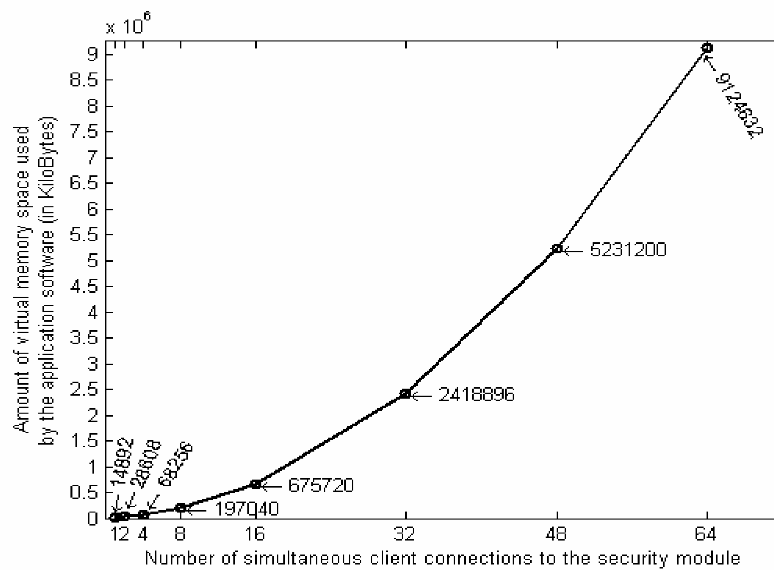

Figure 18 Performance Measurements Results - Virtual Memory Space Usage by the Application Software

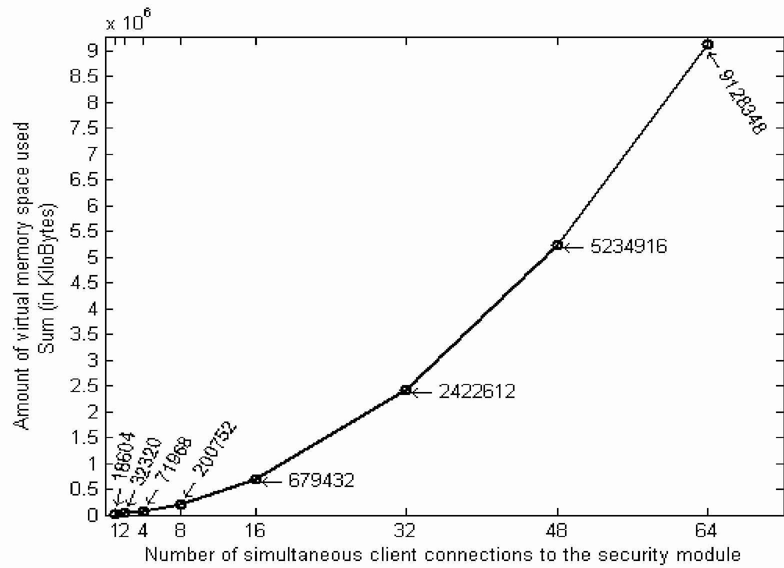

Figure 19 Performance Measurements Results - Virtual Memory Space Usage Sum

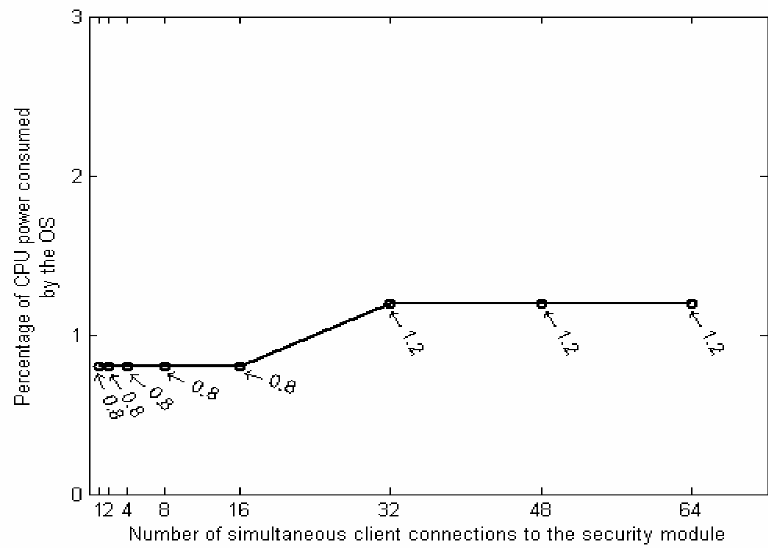

Figure 20 Performance Measurements Results - CPU Power Consumption by the OS

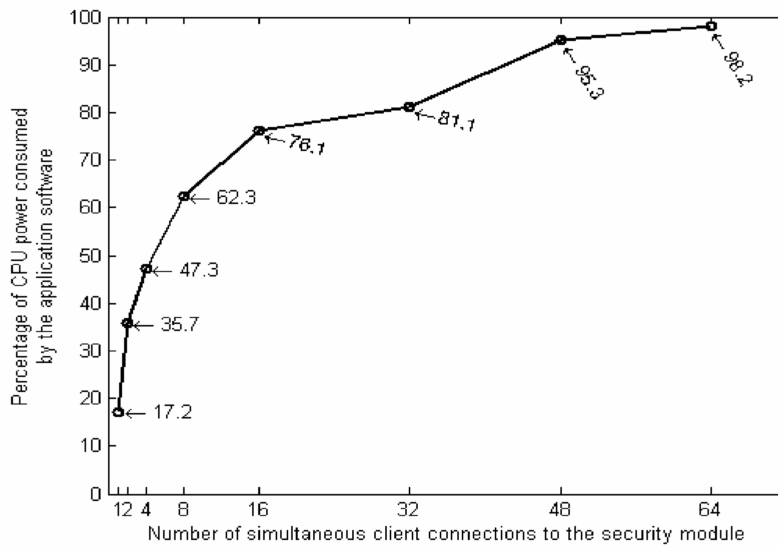

Figure 21 Performance Measurements Results - CPU Power Consumption by the Application Software

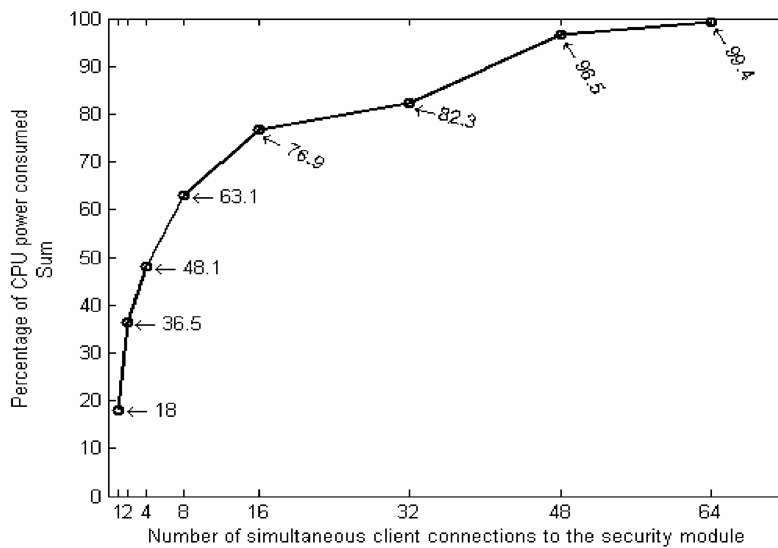

Figure 22 Performance Measurements Results - CPU Power Consumption Sum

Table 1 Performance Measurements Results - Disk Space Usage

Amount of disk space used by the Linux kernel and root filesystem, and their sum	Kernel	Root Filesystem	Sum
	634.2 KB	10.8 MB	11.4 MB

As expected and can be seen in figures 10, 11, 12 and 13 above, as the number of client connections to the security module is increased, both the number of executions of DES and AES symmetric encryption operations by the SM per second and that of RSA (either 1024 or 2048-bit key) public-key signing operation by the SM per second increase. This is *probably* because a CPU-intensive part of the offloader mechanism executed in one dedicated thread can be performed concurrently with a memory-intensive part of the offloader mechanism executed in another dedicated thread. When the number of client connections to the SM is increased beyond a certain approximate threshold, 128 (for RSA-2048 this threshold is 32), values of all these performance parameters start to *decrease*, probably because a large part of the available processing power of the SM is used for management (scheduling, switching, etc..) of many dedicated threads. Therefore, it can be said that *maximum number of executions* of

- DES symmetric encryption operation by the SM per second is 273.5,
- AES-192 symmetric encryption operation by the SM per second is 269.4,
- RSA-1024 public-key signing operation by the SM per second is 74.6,
- RSA-2048 public-key signing operation by the SM per second is 14.5.

As expected and can be seen in figures 15 and 21 above, as the number of client connections to the security module is increased, both the amount of memory space usage and percentage of CPU power consumption by the application software increase. When the number of client connections is increased beyond a certain threshold, 64, total physical memory space usage amount and CPU power consumption percentage by the OS and application software together approaches 256 MB and 100 percent respectively, which could be hazardous for continuous

proper operation of the SM. Therefore, *64* is determined as the *maximum allowable number* of simultaneous client connections to the security module.

Also, as expected and can be seen in figures in Appendix B, as the length of the input data is increased, both the number of executions of DES and AES symmetric encryption operations by the SM per second and that of RSA (either 1024 or 2048-bit key) public-key signing operation by the SM per second decrease. However, these decreases are *small*: for example with 128 simultaneous client connections, the SM executes approximately 274 DES operations and 271 AES operations per second for *20*-bytes of input data (see figures 23 and 24) and executes approximately 268 DES operations and 264 AES operations per second for *1000*-bytes of input data (see figures 29 and 30). Maximum decrease amounts are between 6 and 7. For RSA signing operations, these decrease amounts are very small, even smaller than 1. These suggest that increasing of the length of input data to an operation (either symmetric encryption or public-key signing) to be executed does not greatly increase SM CPU power consumption of the operation.

Since PKCS#1 v1.5 padding is used with RSA public-key signing operation, the length of input data cannot be greater than 116 bytes for 1024-bit RSA signing key and cannot be greater than 244 bytes for 2048-bit RSA signing key (refer to section 4.1.2. Asymmetric_Encryption_Decryption Basic Offloader Mechanism). For DES and AES symmetric encryption operations, if the length of the input data is increased above 1350 bytes, number of executions of these operations by the SM per second increases abnormally beyond 2500 operations. This is *probably* caused by the limited (*embedded*) processing capability of the SM, and not observed in the desktop computer using the same application source code and software libraries.

The results of the performance tests seem satisfactory for a security module that is software-based and network-attached instead of a hardware-based coprocessor add-in card. The security module built in this thesis work can perform upto 75

1024-bit RSA signing operations per second only by relying on its 650 MHz Intel Celeron processor. For example, a famous hardware-based coprocessor SM, IBM PCI Cryptographic Coprocessor 4758, supports upto 175 1024-bit RSA private key operations per second [6]. From two famous hardware-based but network-attached SMs, nCipher netHSM performs upto 2000 RSA signing operations per second [12], and SafeNet Luna SA performs over 1200 1024-bit RSA decrypt operations per second [15]. Also, a renowned hardware-based coprocessor SM, SafeNet Luna PCI, supports over 1200 1024-bit RSA operations per second in its low-end models [14]. Though it is not cited in the literature any information about the types of CPUs and chipsets used in these commercial security modules, they are probably advanced and powerful microprocessors. If the security module built in this work would have one of the recently available powerful CPUs, which has a higher clock frequency (say around 2 GHz) and front-side bus speed and is a quad-core microprocessor with a multiprocessing capability, for example Intel Xeon “Clovertown” series, maximum number of RSA signing operations per second may increase upto 12 times its current value (i.e. near 900) linearly though still without having any cryptographic accelerator hardware.

CHAPTER 5

SUMMARY CONCLUSION

The aim of this thesis work as elaborated in the "Proposed Security Architecture" chapter: to design an extensive, modular and extendable (i.e. open) security architecture whose desired parts could be realized *even* on a software-based security module, is reached as more or less proven by the results of the performance tests done on the built security module.

The most powerful aspect of the designed architecture, that mainly enables the architecture to be feasible to realize on a software-based security module, is its *easy-to-process* communications protocol, which provides access to the offloader mechanisms, yet covering as many options for these mechanisms as possible. Another powerful aspect of the architecture, which also enables the architecture to have the mentioned feasibility, is its *software-only* protection method which provides high security, integrity and confidentiality for stored cryptographic keys, parameters and other secret data even if the security module (realizing the designed architecture) does not have any hardware-based protection or tamper-response mechanism for stored data.

The trust relationship between the security module and a host security server, which wants to connect to it, i.e. each of them identifying that it is connecting with the right party (peer authentication), is ensured by *SSL certificate exchange* hence depends on the SSL protocol. A man-in-the-middle attack is *theoretically* not possible during connecting to the SM, because SSL RSA key exchange is used and the SSL server (SM) *requires* the client (host security server) to send its SSL certificate during the SSL handshake. However, breaking or circumventing of the SSL protocol, which can be defined as the "root of trust" here, and its certificate-

based authentication mechanism may give an attacker administrator user access to the SM. This may be the weakest link of the designed security architecture.

The types of attacks on the SSL protocol cited in the literature can be examined in two groups: man-in-the-middle attacks and cryptographic attacks. Man-in-the-middle attacks in the literature are usually about attacking client machines by impersonating legitimate server sites which use SSL (like banking or e-commerce sites). However, the opposite of this is also possible, i.e. a client (a host security server/user in this work) can attack an SSL-capable server (the SM) by impersonating a legitimate client.

As an example, an attacker can easily create a *self-signed* SSL certificate, which looks plausible for the client being spoofed, and send it to the server (SM) for identification by the SM **[34]**. To mitigate this attack, the SM should not accept self-signed identity certificates from clients. This option is enabled by default in SSLv3.

Or, an attacker may have obtained an SSL certificate (belonging to anyone) signed by a CA *accepted as trusted by the SM* and the private key corresponding to this certificate, and send it to the SM for identification **[34]**. To protect against this, the SM should check if the common name field (plus other required fields) of the received client certificate matches the name of a *registered* host user. The SM built in this work performs this, which is termed in the work as *access control to the SM*.

Alternatively, an attacker may somehow physically access to the SM, load her own trusted root CA certificate to the SM, and later remotely connect to the SM by sending its identity certificate which is *signed by this (virtual) root CA* and whose common name field matches the name of any registered host user **[34]**. To protect against this attack, physical/manual access to the SM should be restricted to only administrator users and the OS of the SM should provide a login mechanism from

terminal for administrator users. It should be noted that *neither* of the three attacks mentioned above stem from a deficiency in the SSL protocol, rather they stem from not properly implemented or configured applications using the SSL protocol.

A famous and recently identified man-in-the-middle attack on applications using SSL is MD5 chosen-prefix attack, which exploits a weakness in the MD5 hash function that allows construction of different messages with the same MD5 hash, i.e. "MD5 collision". An attacker may request and get a legitimate certificate from a trusted CA that uses MD5 function to generate signature of the certificates. Then, the attacker creates a second forged certificate for a *virtual* intermediate CA, whose MD5 hash hence the signature is the same as that of the legitimate certificate. Lastly, the attacker uses this forged certificate to issue a forged identity certificate for any host user and presents this identity certificate to the SM together with the intermediate CA certificate for identification by the SM. The SM will verify the signatures of both of these forged certificates and give the attacker access to the SM. Here, the vulnerability is *not in the SSL protocol*, but in the Public Key Infrastructure. An effective countermeasure is to stop using MD5 for the creation of certificates altogether. **[35]**

The cryptographic attacks on SSL, cited in the literature, are usually *timing-based attacks* related to vulnerabilities in cryptographic method implementations of the OpenSSL library. The first attack takes advantage of the padding used in CBC mode block encryption in an SSL conversation. The versions of OpenSSL before 0.9.6i finishes processing of a data block early when it detects an error in the padding and this creates a measurable time difference that can be used to extract the plaintext from an encrypted block. OpenSSL team fixes this attack by verifying other aspects of the received data block even if its padding is wrong. **[36]** In this work OpenSSL version 0.9.7d was used in the implementation phase, so this attack is not possible. Even if this attack would be possible, the attacker could not view the keys or passwords sent to the SM, because they are sent *in the format in which they are stored in the SM*, i.e. in authenticated and encrypted form.

A second attack takes advantage of the PKCS padding used in RSA encryption of the SSL pre-master secret. Older OpenSSL implementations reveal information (either an error message or an early finish time difference) about whether the pre-master secret is decrypted to a valid PKCS header, and this may enable an attacker to guess the SSL RSA private key of the SSL server (the SM) and hence obtain session keys for all subsequent SSL sessions between the SM and host users. A similar attack builds on the information (time difference or error message) revealed by an SSL server when validating the SSL version number contained in the first two bytes of an SSL payload. This may enable an attacker to guess the session keys for the SSL session. The OpenSSL team published patches for these two vulnerabilities in version 0.9.6j. **[36]** Hence, these attacks are not possible for the implementation in this work.

Another timing-based attack exploits slight differences in the time that it takes to decrypt the data/pre-master secret with the SSL server's RSA private decryption key, when the value of the data is varied. This attack works best in a local network, but is possible over any good network; it takes an attacker around a million queries to extract the SSL RSA private key of the server (SM). The most robust solution to this problem is *"RSA blinding"*: the server multiplies the data by a random number before RSA decryption and reverses this after decryption, so any timing variations for the decryption operation will depend on this random number *unknown to the attacker*. RSA blinding is enabled by default in versions of OpenSSL starting from 0.9.7b. **[36]** Therefore, this attack is not possible for the SM built in this work, which uses OpenSSL 0.9.7d.

As can be deduced from all these discussions and other sources in the literature, the *SSLv3* protocol is *theoretically* very secure and more secure than most of the similar protocols providing application or transport layer security. However, there may exist vulnerabilities *in practice* of its implementation, but the most commonly used SSL implementation, OpenSSL, continuously improves its code to remove the vulnerabilities. Rather than using/trusting the SSLv3 protocol, a new protocol

may be designed from scratch that enables remote authenticated access and secure data transfer to the SM by host security servers/users. However, this could be a duplication of most of the efforts made in designing the SSL protocol and it cannot be guaranteed that the new protocol will not have any vulnerabilities for a long period of time.

A more reasonable way to strengthen the weakest link may consider utilizing the idea of threshold cryptography: to protect information or *computation* by fault-tolerantly distributing it among a cluster of cooperating computers **[37]**. For example, the SSL handshake, which involves certificate exchange-based authentication of the peers (the SM and a host user) and is a highly sensitive operation, can be performed by several cooperating SMs so that an attacker would need to corrupt authentication mechanisms of *a threshold number* of SMs to gain administrator user access. Similarly, the task of SSL authenticating and encrypting the SSL payloads transferred between the SM and host security servers can also be divided among several cooperating SMs to improve its security. Fault-tolerant, secure, efficient protocols will be required to communicate the cooperating SMs **[37]**. In the same manner, the *pieces* of the SSL RSA private key or SSL session keys of the SM can be distributed among several cooperating SMs to prevent reveal/reconstruction of these keys to/by an attacker if the attacker has not invaded *a threshold number* of cooperating SMs.

Another type of attack to the designed architecture, which does not stem from relying on the SSL protocol or its certain implementation and may happen in any *PKI-based system*, concerns the case if the SSL private key of a host user corresponding to its SSL certificate, used to connect to the SM, is stolen by an attacker. This would give the attacker illegitimate host user access to the SM, and the attacker may access to the cryptographic keys and parameters of the host user stored in the SM via offloader mechanism execution requests. However, possibility of this attack is removed for *administrational* operations to be performed on the SM. Since administrational offloader mechanisms require *at*

least two administrator host users have connected to the SM to execute, stealing of the SSL private key of just one administrator user by an attacker *cannot* give the attacker administrator user access to the SM. This can also be considered a certain practice of threshold cryptography, and by this way the architecture eliminates single-point-of-failure over an administrator user. Moreover, an attacker must obtain module code signing private keys of *at least two* administrator users to load a new signed firmware to the security module.

Since the security architecture designed is intended to be realized on an embedded SBC and these computers usually have limited processing capability and memory space compared to laptops or server machines, the security module can easily be short of processing power or memory when fulfilling many offloader mechanism execution and connection requests from a number of clients. Therefore, the SM should be able to delegate/redirect some of its incoming offloader mechanism execution and connection requests to other security modules on the same local network. Also, when the SM cannot find in its storage cryptographic keys and parameters of a host user required to fulfill her offloader mechanism execution request, it should be able to probe other security modules on the same local network for these keys and parameters. Consequently, an important improvement to the designed security architecture may be the addition of new offloader mechanisms and related communications protocol messages needed for communicating several security modules on the same local network.

The security architecture applies role-based authorization of connected host security servers/users to control their access to offloader mechanisms. Currently, only two roles are defined: “User” and “Administrator”. As another improvement to the architecture, these roles could be extended to include others such as “SET Merchant”, “SET Payment Gateway”, “Kerberos Authentication Server”, etc.. By this way, for instance a host user who is a SET merchant could not access the offloader mechanisms intended to be requested by SET payment gateways. In the current state of the architecture, although such an access is possible for this host

user, who is the SET merchant, she cannot access a cryptographic key or parameter of any SET payment gateway stored in the module and hence cannot execute these offloader mechanisms without errors. The communications protocol by design *cannot allow* a host security server/user, requesting an offloader mechanism, to access cryptographic keys and parameters of any other host security server/user stored in the SM.

As stated in previous chapters, an administrator user can also manually login to the security module and transfer secret keys and parameters or SM internal data files to the SM via USB storage devices. In this case, authentication of the administrator user by the SM and secure transfer of the mentioned files to the SM should be guaranteed by the OS of the built security module. For instance, the OS of the SM may provide a highly secure login mechanism and may check for the existence of any unwanted software in the USB storage device plugged to the SM. However, this issue is completely related with *implementation* and not in the scope of the designed security architecture, since the architecture is independent of any OS or hardware that could be used in implementation.

One of two further works may be addition of new offloader mechanisms and related data, implementing operations of other network security protocols (like S/MIME and IPSec), to the designed architecture owing to the fact that the architecture is extendable. The other work may be providing reference implementations/algorithms for existing offloader mechanisms other than the implemented basic offloader mechanisms.

REFERENCES

[1] Hardware Security Module - Wikipedia the free encyclopedia, http://en.wikipedia.org/w/index.php?title=Hardware_Security_Module&oldid=218886319, last visited on 12 June 2008

[2] CREN - Hardware Security Modules, http://www.cren.net/crenca/onepagers/hsm2.html, last visited on 14 July 2008

[3] William Stallings, Network Security Essentials: applications and standards, Prentice Hall, 2000

[4] Peter Gutmann - University of Auckland New Zealand, An Open-source Cryptographic Coprocessor, August 2000

[5] IBM PCI Cryptographic Coprocessor - CCA support, http://www-03.ibm.com/security/cryptocards/pcicc/overcca.shtml, last visited on 6 April 2008

[6] IBM PCI Cryptographic Coprocessor - Product Summary, http://www-03.ibm.com/security/cryptocards/pcicc/overproduct.shtml, last visited on 6 April 2008

[7] IBM 4764 product and PCIXCC feature overview, http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml, last visited on 7 April 2008

[8] nCIPHER Corporation, nCIPHER SECURITY WORLD WHITE PAPER, April 2001

[9] IBM Corporation, CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors, July 2006

[10] nCIPHER Corporation, SECURE EXECUTION ENGINE WHITE PAPER, April 2001

[11] Cryptographic Hardware Platform - nCipher (miniHSM), http://www.ncipher.com/cryptographic_hardware/hardware_security_modules/72/minihsm, last visited on 10 May 2008

[12] Cryptographic Hardware Platform - nCipher (netHSM), http://www.ncipher.com/cryptographic_hardware/hardware_security_modules/10/nethsm, last visited on 10 May 2008

[13] nCIPHER Corporation, NCIPHER netHSM TECHNICAL ARCHITECTURE - WHITE PAPER, August 2006

[14] Luna PCI, http://www.safenet-inc.com/products/pki/lunaPCI.asp, last visited on 12 May 2008

[15] Luna SA, http://www.safenet-inc.com/products/pki/lunaSA.asp, last visited on 12 May 2008

[16] William Stallings, Cryptography and network security: principles and practice, Prentice Hall, 2nd edition 1999

[17] Advanced Encryption Standard - Wikipedia the free encyclopedia, http://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=250648121, last visited on 9 November 2008

[18] C. Neuman - USC-ISI, T. Yu - MIT, S. Hartman - MIT, K. Raeburn - MIT, The Kerberos Network Authentication Service (V5) - Request for Comments: 4120, Network Working Group - The Internet Society, July 2005

[19] R. Housley - RSA Laboratories, W. Polk - NIST, W. Ford - VeriSign, D. Solo - Citigroup, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile - Request for Comments: 3280, Network Working Group - The Internet Society, April 2002

[20] J. Schaad - Soaring Hawk Consulting, Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF) - Request for Comments: 4211, Network Working Group - The Internet Society, September 2005

[21] Alan O. Freier - Netscape Communications, Philip Karlton - Netscape Communications, Paul C. Kocher - Independent Consultant, The SSL Protocol Version 3.0 - Internet Draft, Transport Layer Security Working Group - Internet Engineering Task Force (IETF), November 1996

[22] SET Writing Team, SET Secure Electronic Transaction Specification Book 1: Business Description, MasterCard and VISA, Version 1.0 May 1997

[23] SET Writing Team, SET Secure Electronic Transaction Specification Book 3: Formal Protocol Definition, MasterCard and VISA, Version 1.0 May 1997

[24] Simplifying Embedded Linux Development with Graphical Tools, http://linuxdevices.com/articles/AT4574262276.html, last visited on 16 June 2008

[25] Using Linux in Embedded Systems and Smart Devices, http://www.linuxdevices.com/articles/AT3155773172.html, last visited on 16 June 2008

[26] Embedded Linux Distributions Quick Reference Guide (Part 2), http://www.linuxdevices.com/articles/AT9952405558.html, last visited on 24 June 2008

[27] Embedded Linux Distributions Quick Reference Guide (Part 3), http://www.linuxdevices.com/articles/AT4525882120.html, last visited on 24 June 2008

[28] Creating embedded Linux filesystems with graphical tools, http://linuxdevices.com/articles/AT3617019494.html, last visited on 25 June 2008

[29] Migrating to Linux kernel 2.6 -- Part 1: Customizing a 2.6-based kernel, http://linuxdevices.com/articles/AT3855888078.html, last visited on 25 June 2008

[30] A developer's review of Qplus - an open source embedded Linux toolkit, http://www.linuxdevices.com/articles/AT5640843706.html, last visited on 26 June 2008

[31] ETRI (Electronics and Telecommunications Research Institute - Korea), Qplus-P Target Builder User's Guide, Version 1.2 December 2002

[32] PC/104 Embedded Systems FAQ, EBX, Linux, Real Time, data acquisition, http://www.controlled.com/pc104faq/, last visited on 4 April 2008

[33] Advantech Corporation Limited, PCM-3370 Data Sheet, February 2007

[34] Peter Burkholder, SSL Man-in-the-Middle Attacks, SANS Institute InfoSec Reading Room, Version 2.0 February 2002

[35] MD5 considered harmful today, http://www.win.tue.nl/hashclash/rogue-ca/, last visited on 21 May 2009

[36] Recent Attacks against OpenSSL likely to be applicable to other SSL implementations - Netcraft, http://news.netcraft.com/archives/2003/04/21/recent_attacks_against_openssl_likely_to_be_applicable_to_other_ssl_implementations.html, last visited on 22 May 2009

[37] CIS: Threshold Cryptography, http://groups.csail.mit.edu/cis/cis-threshold.html, last visited on 23 May 2009

APPENDIX A

COMMUNICATIONS PROTOCOL MESSAGE FORMATS

In the tables below, the fields of offloader mechanism execution request and response messages for basic and administrational offloader mechanisms are given with their possible values and explanations.

Table 2 Symmetric_Encryption_Decryption Offloader Mechanism Execution Request Message

Fields	Operation Type	Encryption Method	Encryption Mode	Unit of Transmission
Explanation	Encrypt Decrypt	DES TDES BlowFish RC5 CAST-128 RC2 RC4 AES IDEA	ECB CBC (with PKCS padding) CFB OFB PCBC	Present if encryption mode is CFB or OFB. It is smaller than or equal to the block length of the encryption method **[16]**.

Fields	Key Length	Block Length	Number of Rounds
Explanation	Refers to the key length of the encryption method.	Refers to the block length of the encryption method.	Refers to the number of rounds of the encryption method.

Fields	$ID_{Host\ User}$	$ID_{Remote\ User}$	Input Data
Explanation	Identifier of the host user on behalf of who the host security server requests execution of this offloader mechanism.	Identifier of the remote user with whom the host user is communicating and shares its symmetric key and IV.	Input data to be encrypted or decrypted.

Table 3 Symmetric_Encryption_Decryption Offloader Mechanism Execution Response Message

Fields	Status	Output Data
Explanation	OK Internal Error Encrypt/Decrypt Error	

Table 4 Asymmetric_Encryption_Decryption Offloader Mechanism Execution Request Message

Fields	Operation Type	Encryption Method
Explanation	Encrypt Sign Sign and Encrypt Decrypt Verify Decrypt and Verify	RSA ECC ElGamal Present if the operation type includes “Encrypt” or “Decrypt”.
Fields	Encryption Key Size	Signing Method
Explanation	Present if the operation type includes “Encrypt” or “Decrypt”. Key size of the public-key encryption method. For RSA: 512, 1024, 2048, 3072, 4096 bits	RSA Present if the operation type includes “Sign” or “Verify”.
Fields	Signing Key Size	$ID_{\text{Host User}}$
Explanation	Present if the operation type includes “Sign” or “Verify”. Key size of the public-key signing method.	Identifier of the host user on behalf of who the host security server requests execution of this mechanism. Present if the operation type includes “Sign” or “Decrypt”.
Fields	$ID_{\text{Remote User}}$	Input Data
Explanation	Identifier of the remote user with who the host user is communicating. Present if the operation type includes “Encrypt” or “Verify”.	Input data to be - encrypted, - signed, - signed and encrypted, - decrypted, - verified, or - decrypted and verified.

Table 5 Asymmetric_Encryption_Decryption Offloader Mechanism Execution Response Message

Fields	Status	Output Data
Explanation	OK Internal Error Encrypt/Decrypt Error Sign/Verify Error	

Table 6 Session_Key_Calculation Offloader Mechanism Execution Request Message

Fields	Operation Type	Key Exchange Method	Key Size
Explanation	Key Exchange	Diffie-Hellman ECC	Key size for the chosen key exchange method. For DH: 512, 1024, 2048, 4096 bits For ECC: 128, 160, 224, 256 bits
Fields	$ID_{\text{Host User}}$	$ID_{\text{Remote User}}$	Encryption Method
Explanation	Used to locate in the SM storage the stored private key exchange key and global key exchange parameters of the host user.	Used to locate in the SM storage the stored public key exchange key certificate of the remote user.	Determines the type (nature) of the session key to be generated. DES, TDES, BlowFish, RC5, CAST-128, RC2, RC4, AES, IDEA
Fields	Key Length	Block Length	Number of Rounds
Explanation	They refer to the key length, block length and number of rounds parameters of the encryption method, respectively. They all determine the nature of the session key that will be generated and stored in the SM.		

Table 7 Session_Key_Calculation Offloader Mechanism Execution Response Message

Fields	Status
Explanation	OK Internal Error Key Exchange Error

Table 8 Message_Authentication_Code_Operations Offloader Mechanism Execution Request Message

Fields	Operation Type	MAC Method
Explanation	Authenticate Authenticate and Encrypt Encrypt and Authenticate Verify Decrypt and Verify Verify and Decrypt	HMAC Fast HMAC CMAC (using DES) DAA

Fields	MAC Key Size	Hash Method
Explanation	Key size for the MAC method.	Hash method is present only if MAC method is HMAC or Fast HMAC. MD5, SHA-1, RIPEMD-160, Whirlpool

Fields	Encrypt. Method	Encrypt. Mode	Unit of Transmis.	Key Length	Block Length	Number of Rounds
Explanation	They are the same fields as in the first basic offloader mechanism "Symmetric_Encryption_Decryption". All of these six fields are present if "Encrypt" or "Decrypt" is included as part of the operation type.					

Fields	$ID_{Host\ User}$	$ID_{Remote\ User}$	Input Data
Explanation	Identifier of the host user, that will be used in locating in the SM storage her MAC secret key, symmetric encryption key and IV shared with the remote user.	Identifier of the remote user, that will also be used in locating in the SM storage the shared MAC secret key, symmetric encryption key and IV.	Input data to be authenticated and optionally encrypted, or optionally decrypted and verified.

Table 9 Message_Authentication_Code_Operations Offloader Mechanism Execution Response Message

Fields	Status	Output Data
Explanation	OK Internal Error Authenticate/Verify Error Encrypt/Decrypt Error	

Table 10 Message_Digest_Operations Offloader Mechanism Execution Request Message

Fields	Operation Type	Hash Method
Explanation	Authenticate Authenticate and Encrypt Sign Sign and Encrypt Authenticate with Secret Value Authenticate with Secret Value and Encrypt Verify Decrypt and Verify Verify Signed Data Decrypt and Verify Signed Data Verify with Secret Value Decrypt and Verify with Secret Value	MD5 SHA-1 RIPEMD-160 Whirlpool

Fields	Encrypt Method	Encrypt Mode	Unit of Transmis.	Key Length	Block Length	Number of Rounds
Explanation	They are the same fields as in the first basic offloader mechanism “Symmetric_Encryption_Decryption”. All of these fields are present if “Encrypt” or “Decrypt” is included as part of the operation type, or if the operation type is “Authenticate” or “Verify”.					

Fields	Signing Method	Signing Method Key Size
Explanation	RSA Signing Method field refers to the public-key method that will be used to asymmetrically encrypt (sign) or decrypt (verify) the message digest of the input data. Key Size field refers to the size of the private/public key to be used with this method. These fields are present only if “Sign” or “Verify Signed Data” is included as part of the operation type.	

Fields	$ID_{\text{Host User}}$	$ID_{\text{Remote User}}$	Input Data
Explanation	Present if the operation type is *not* “Verify Signed Data”. It will be used in locating (in the SM storage) the symmetric encryption key and IV and hash secret value of the host user shared with the remote user. Also, it will be used in locating the private signing key of the host user.	Present if the operation type is *not* “Sign”. It will also be used in locating (in the SM storage) the symmetric encryption key and IV and hash secret value shared by the host user and remote user. Also, it will be used in locating the public signing key certificate of the remote user.	

Table 11 Message_Digest_Operations Offloader Mechanism Execution Response Message

Fields	Status	Output Data
Explanation	OK Internal Error Authenticate/Verify Error Sign/Verify Signed Data Error Encrypt/Decrypt Error	

Table 12 Digital_Signature_Calculation Offloader Mechanism Execution Request Message

Fields	Operation Type	Signing Method	Key Size
Explanation	Sign Verify	DSA ECDSA RSA-SHA-1	DSA: 512 - 1024 (in 64-bit increments **[16]**) ECDSA: 128/192/256 bits
Fields	$ID_{Host\ User}$	$ID_{Remote\ User}$	Input Data
Explanation	It will be used in locating the private signing key and global signing parameters of the host user in the SM storage.	Present only if the operation type is "Verify". It will be used in locating the public signing key certificate of the remote user in the SM storage.	Input data to be signed or verified.

Table 13 Digital_Signature_Calculation Offloader Mechanism Execution Response Message

Fields	Status	Output Data
Explanation	OK Internal Error Sign/Verify Error	

Table 14 Cryptographic_Keys_Parameters_Creation Offloader Mechanism Execution Request Message

Fields	Operation Type	Key Generation Method
Explanation	Key Generate	RSA, ECC, ElGamal, DH, DSA, ECDSA, DES, TDES, AES, RC2, RC5, IDEA, RC4, Blowfish, CAST-128, HMAC, Fast HMAC, CMAC, DAA

Fields	Key Size	ID_{Owner}
Explanation	Key size for the key generation method.	Identifier of the *host user* for who the keys and parameters will be generated.

Fields	Key Usage	$ID_{Remote\ User}$
Explanation	Encryption, Signing, Key Exchange, Authentication, Kerberos-SessionKey, SSL-MessageAuthentication, etc..	This field is present if a symmetric encryption key and IV, or a MAC key will be generated. Identifier of the remote user with who the host user will share the generated key and parameter.

Table 15 Cryptographic_Keys_Parameters_Creation Offloader Mechanism Execution Response Message

Fields	Status
Explanation	OK KO

Table 16 Time_Time_Window_Adjustment Offloader Mechanism Execution Request Message

Fields	Operation Type	Time Value	Time Window Value
Explanation	Adjust Time Adjust Time Window Adjust Both	Present if the operation type is “Adjust Time” or “Adjust Both”.	Present if the operation type is “Adjust Time Window” or “Adjust Both”.

Table 17 Time_Time_Window_Adjustment Offloader Mechanism Execution Response Message

Fields	Status
Explanation	OK Time Adjust Error Time Window Adjust Error

Table 18 Key_Entry_Erasure_Backup Offloader Mechanism Execution Request Message

Fields	Operation Type	ID_{Owner1}
Explanation	Key Entry Key Erasure Key Backup	Identifier of the *host or remote user* to who the following key, parameter, public-key certificate or CRL belongs.

Fields	Key/Parameter/Certificate/CRL$_1$ Type	Key/Parameter/Certificate/ CRL$_1$
Explanation	RSA-1024-priv DSA-512-cert DH-1024-cert AES-192-skey DES-ivec HMAC-mkey CRL-*pub. date* . . .	Present only if the operation type is "Key Entry". Secret keys and parameters are in TDES-encrypted (with the storage master key of the SM) form. They were encrypted after their SHA-1 hash codes appended.

Fields	Key/Parameter/ Certificate/CRL$_1$ Usage	$ID_{Remote\ User1}$
Explanation	Encryption, Signing, Authentication, Key exchange, Kerberos-Subkey, SSL-KeyExchange, SSL-Signing, etc..	The field is present for symmetric encryption keys and IVs, MAC keys, and hash secret values. Identifier of the remote user with who the key or parameter owner (host user) shares the key or parameter.

Fields	ID_{Owner2}	Key/Parameter/Certificate/ CRL$_2$ Type	Key/Parameter/ Certificate/CRL$_2$	• • •
Explanation				

Table 19 Key_Entry_Erasure_Backup Offloader Mechanism Execution Response Message

Fields	Status
Explanation	OK Internal Error Key/Parameter/Certificate/CRL Process Error
Fields	Number of Successfully Processed Keys/Parameters/ Certificates/CRLs
Explanation	Until an error occurred during processing or all are processed without an error.

Table 20 User_Role_Administration Offloader Mechanism Execution Request Message

Fields	Operation Type	$ID_{Host\ User1}$	$Role_1$
Explanation	User-Role Modify User-Role Add User-Role Delete	Identifier of the host user whose role will be managed.	Present if the operation type is *not* “User-Role Delete”. Currently, two roles are defined: “User” and “Administrator”.
Fields	$ID_{Host\ User2}$	$Role_2$	• • •
Explanation			

Table 21 User_Role_Administration Offloader Mechanism Execution Response Message

Fields	Status	Number of User-Role Entries Successfully Processed
Explanation	OK Internal Error User-Role Entry Process Error	Until an error occurred during processing or all are processed without an error.

Table 22 Module_Code_Status_Receiving_Loading Offloader Mechanism Execution Request Message

Fields	Operation Type	Code Piece$_1$	Code Piece Signature$_1$
Explanation	Get Code Status Load Signed Code	Present if operation type is “Load Signed Code”. Each code piece has a fixed size of 100 KB.	RSA-SHA-1 signature made using the private code signing keys of at least two administrator users successively. Present if the operation type is “Load Signed Code”.
Fields	Code Piece$_2$	Code Piece Signature$_2$	• • •
Explanation			

Table 23 Module_Code_Status_Receiving_Loading Offloader Mechanism Execution Response Message

Fields	Status	Code Piece$_1$	Code Piece Signature$_1$
Explanation	OK Internal Error Verify Error - (in loading the new signed firmware to the SM)	Present if the operation type is “Get Code Status”. Each code piece has a fixed size.	Present if the operation type is “Get Code Status”. Stored signature block for the code piece.
Fields	Code Piece$_2$	Code Piece Signature$_2$	• • •
Explanation			

APPENDIX B

RESULTS OF PERFORMANCE MEASUREMENTS

In the figures below, the performance measurements results (averages) for the two performance parameters of the built SM:

1. number of executions of DES and AES-192 symmetric encryption operations per second,
2. number of executions of RSA (1024 and 2048-bit key) public-key signing operation per second,

for various lengths (20, 50, 200, 500, 1000-bytes) of input data, other than 100-bytes (which are shown in the text), are shown. After the figures, performance measurements results for all of the four performance parameters for 100-bytes of input data are listed in tables.

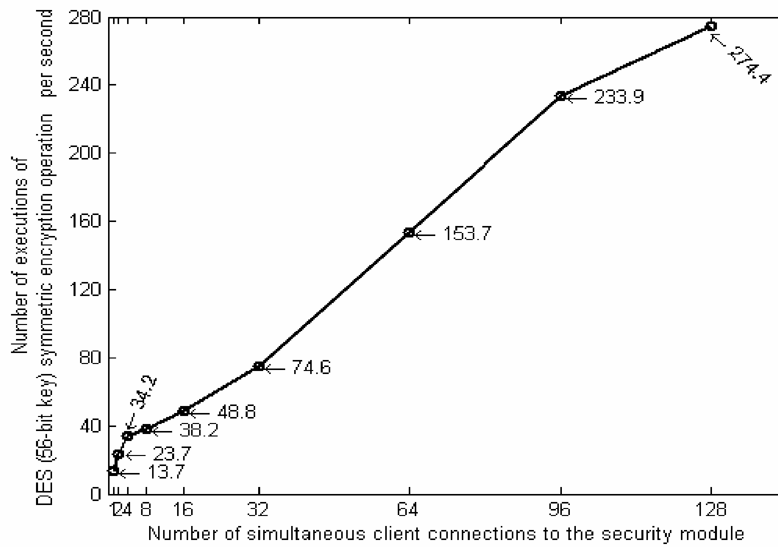

Figure 23 Performance Measurements Results - DES Symmetric Encryption, 20-bytes of Input Data

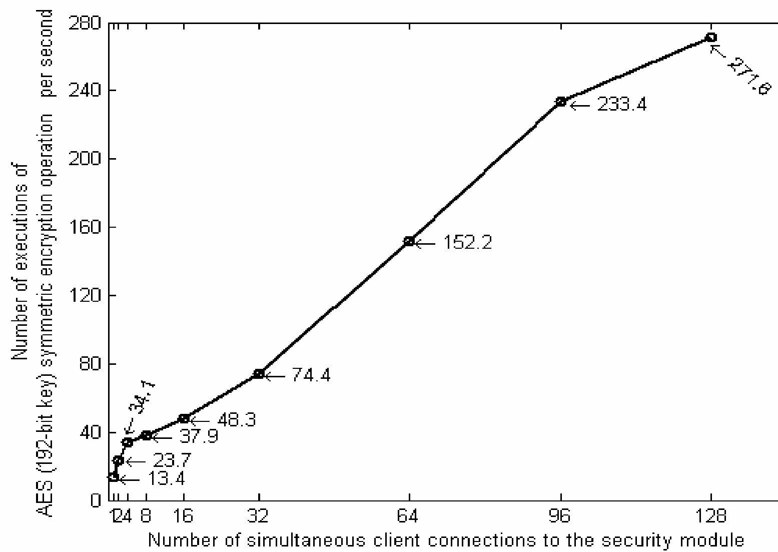

Figure 24 Performance Measurements Results - AES Symmetric Encryption, 20-bytes of Input Data

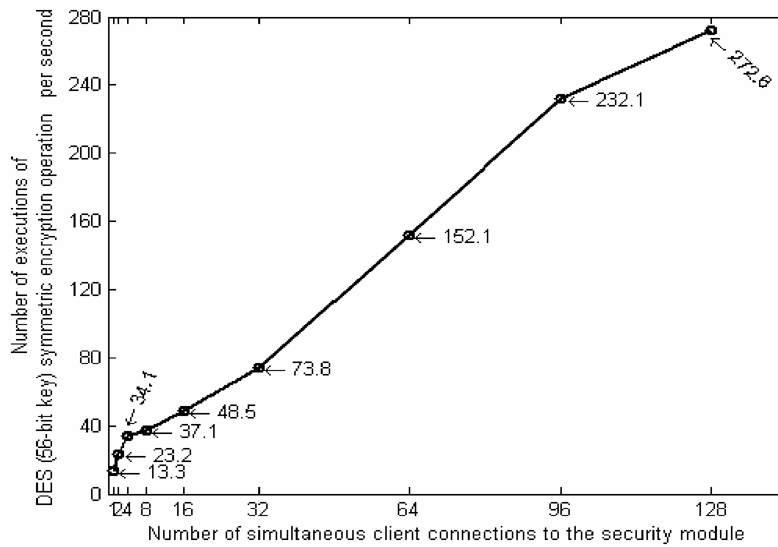

Figure 25 Performance Measurements Results - DES Symmetric Encryption, 200-bytes of Input Data

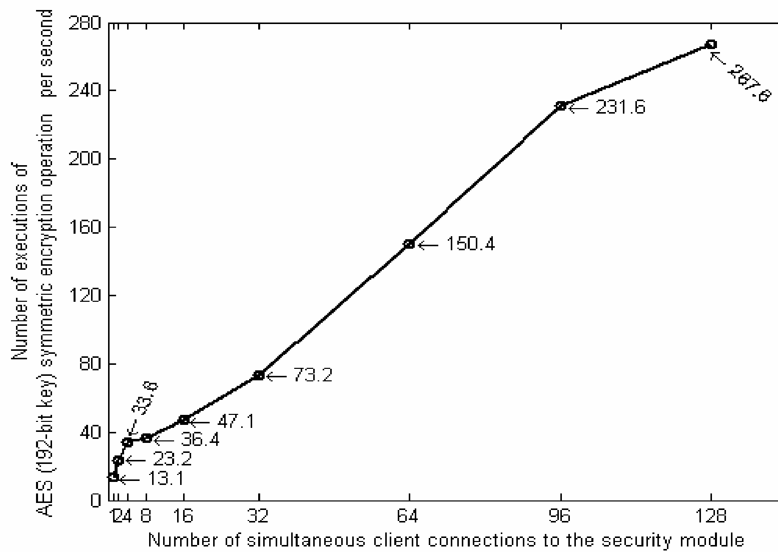

Figure 26 Performance Measurements Results - AES Symmetric Encryption, 200-bytes of Input Data

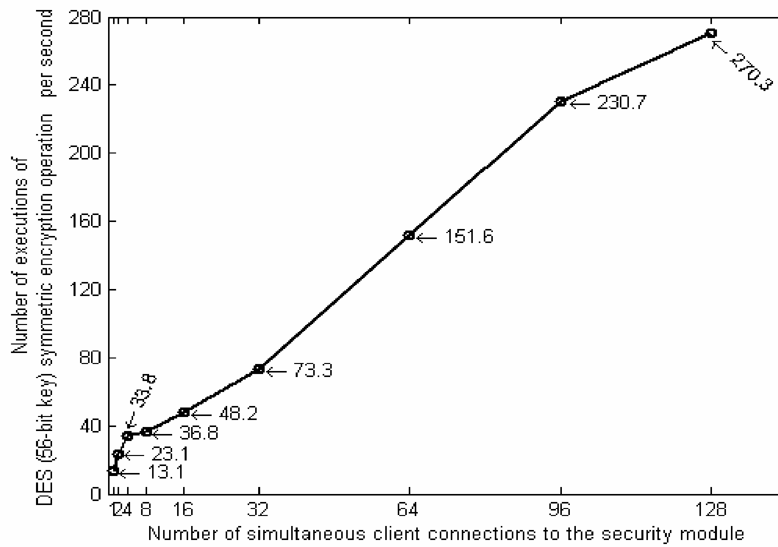

Figure 27 Performance Measurements Results - DES Symmetric Encryption, 500-bytes of Input Data

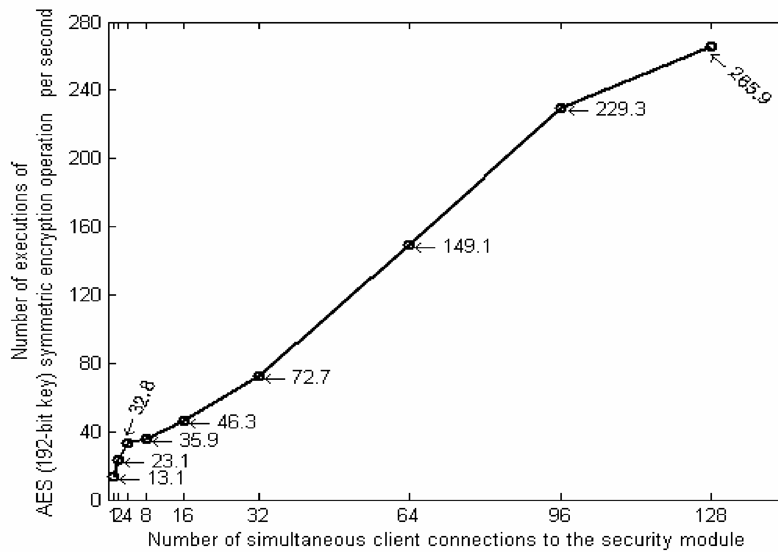

Figure 28 Performance Measurements Results - AES Symmetric Encryption, 500-bytes of Input Data

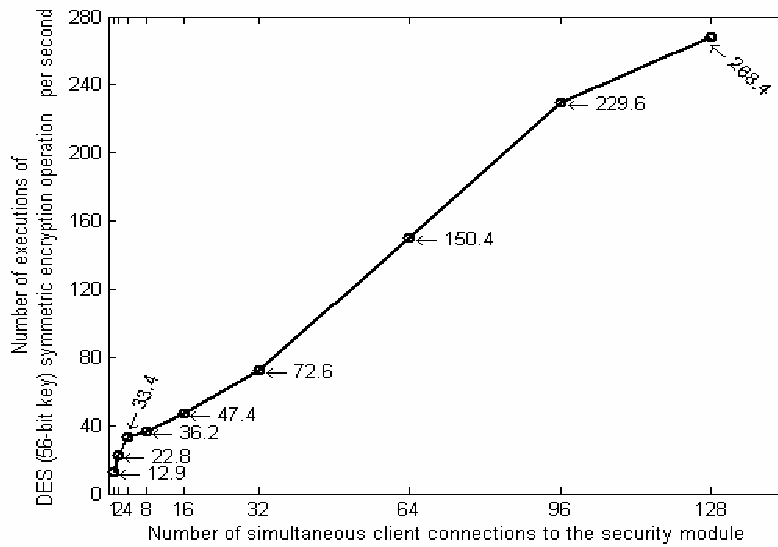

Figure 29 Performance Measurements Results - DES Symmetric Encryption, 1000-bytes of Input Data

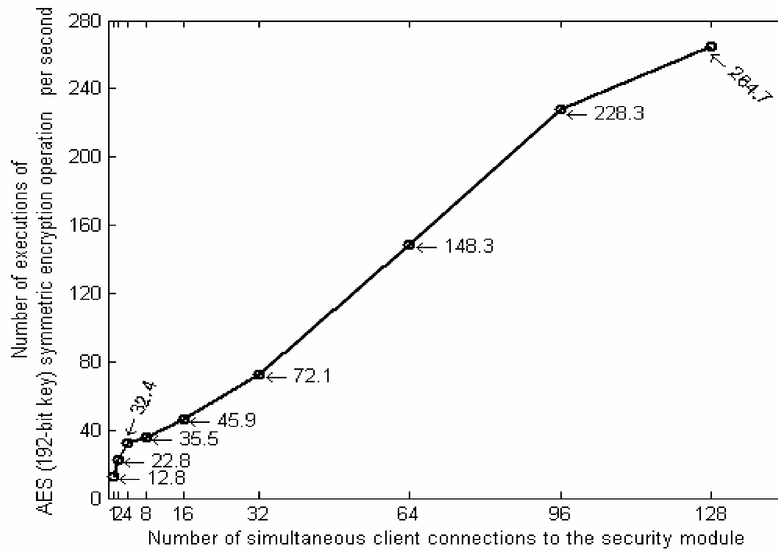

Figure 30 Performance Measurements Results - AES Symmetric Encryption, 1000-bytes of Input Data

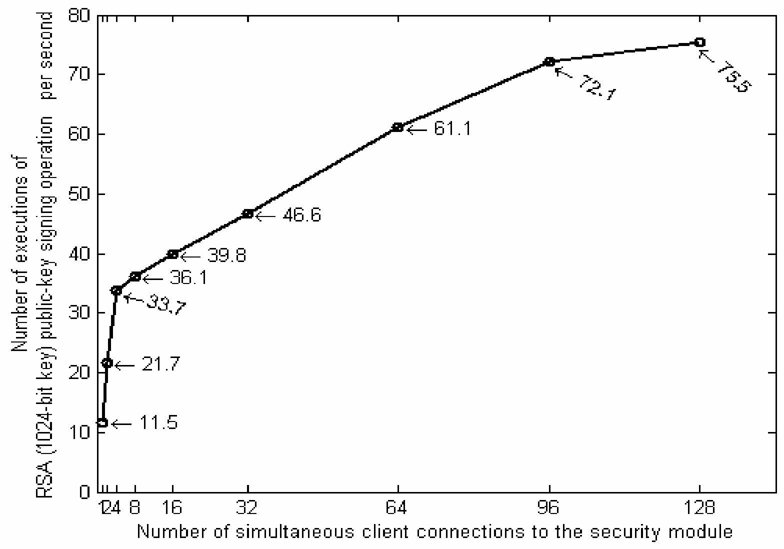

Figure 31 Performance Measurements Results - RSA 1024-bit Signing, 20-bytes of Input Data

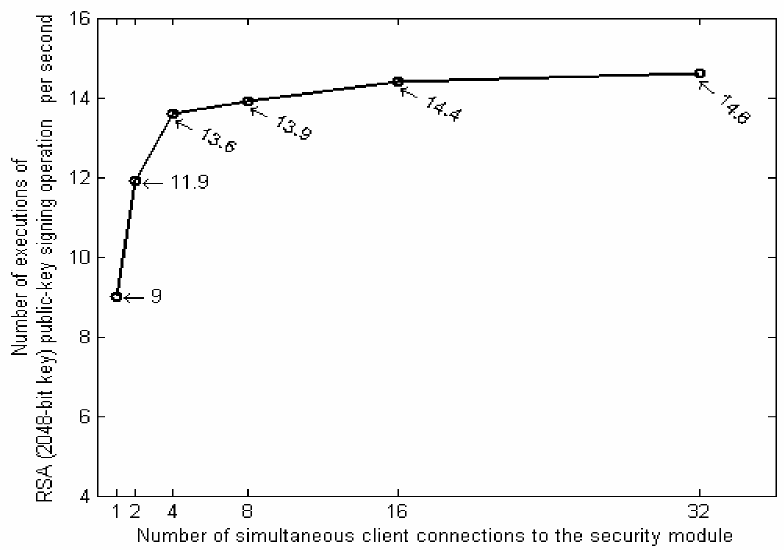

Figure 32 Performance Measurements Results - RSA 2048-bit Signing, 20-bytes of Input Data

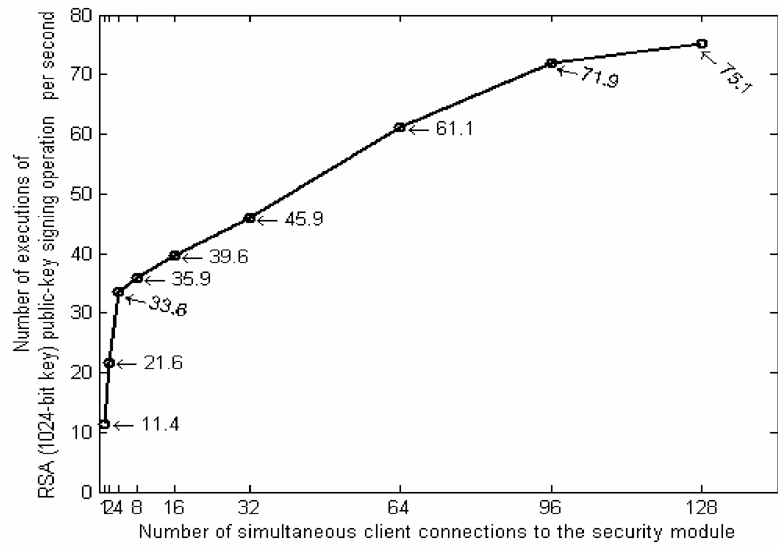

Figure 33 Performance Measurements Results - RSA 1024-bit Signing, 50-bytes of Input Data

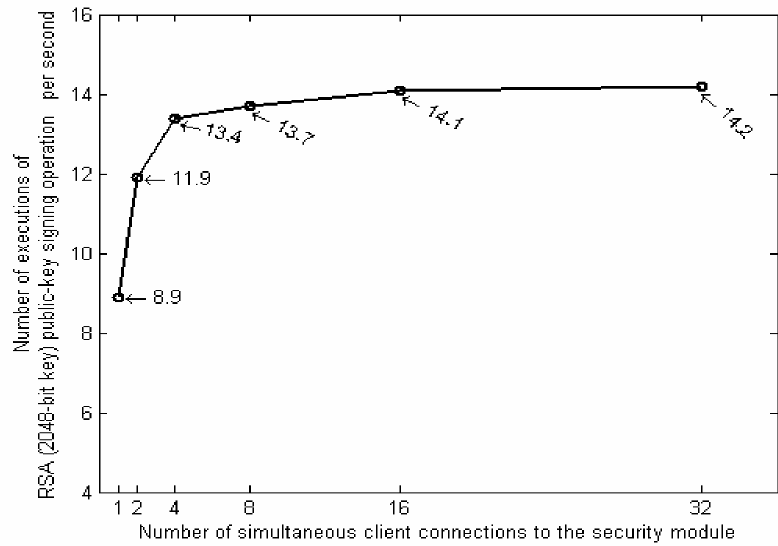

Figure 34 Performance Measurements Results - RSA 2048-bit Signing, 200-bytes of Input Data

Table 24 Performance Measurements Results - Symmetric Encryption, 100-bytes of Input Data

	Number of simultaneous client connections to the security module				
Number of executions of DES (56-bit key) symmetric encryption operation per second	1	2	4	8	16
	13.6	23.5	34.2	37.4	48.6
	32	64	96	128	
	74.2	152.5	233.1	**273.5**	
	Number of simultaneous client connections to the security module				
Number of executions of AES (192-bit key) symmetric encryption operation per second	1	2	4	8	16
	13.3	23.6	33.9	37.1	47.7
	32	64	96	128	
	73.8	151.3	232.7	**269.4**	

Table 25 Performance Measurements Results - Signing, 100-bytes of Input Data

	Number of simultaneous client connections to the security module				
Number of executions of RSA (1024-bit key) public-key signing operation per second	1	2	4	8	16
	11.3	21.7	33.6	35.9	39.2
	32	64	96	128	
	45.8	60.7	71.4	**74.6**	
	Number of simultaneous client connections to the security module				
Number of executions of RSA (2048-bit key) public-key signing operation per second	1	2	4	8	16
	8.9	11.9	13.5	13.9	14.2
	32				
	14.5				

Table 26 Performance Measurements Results - Memory Space Usage

Amount of memory space used, physical and virtual memories, (in KiloBytes)	Number of simultaneous client connections to the security module							
	1	2	4	8	16	32	48	64
by the OS	(phy.) 1732 (virt.) 3712	(phy.) 1732 (virt.) 3712	(phy.) 1732 (virt.) 3712	(phy.) 1732 (virt.) 3712	(phy.) 1732 (virt.) 3712	(phy.) 1736 (virt.) 3716	(phy.) 1736 (virt.) 3716	(phy.) 1736 (virt.) 3716
by the application software	(phy.) 4428 (virt.) 14892	(phy.) 6032 (virt.) 28608	(phy.) 9456 (virt.) 68256	(phy.) 16960 (virt.) 197040	(phy.) 36360 (virt.) 675720	(phy.) 87856 (virt.) 2418896	(phy.) 153400 (virt.) 5231200	(phy.) 247104 (virt.) 9124632
and their sum	(phy.) 6160 (virt.) 18604	(phy.) 7764 (virt.) 32320	(phy.) 11188 (virt.) 71968	(phy.) 18692 (virt.) 200752	(phy.) 38092 (virt.) 679432	(phy.) 89592 (virt.) 2422612	(phy.) 155136 (virt.) 5234916	(phy.) **248840** (virt.) 9128348

Table 27 Performance Measurements Results - CPU Power Consumption

<table>
<tr><td rowspan="4">Percentage of CPU power consumed by the OS</td><td colspan="15">Number of simultaneous client connections to the security module</td></tr>
<tr><td colspan="3">1</td><td colspan="3">2</td><td colspan="3">4</td><td colspan="3">8</td><td colspan="3">16</td></tr>
<tr><td colspan="3">0.8</td><td colspan="3">0.8</td><td colspan="3">0.8</td><td colspan="3">0.8</td><td colspan="3">0.8</td></tr>
<tr><td colspan="5">32
1.2</td><td colspan="5">48
1.2</td><td colspan="5">64
1.2</td></tr>
<tr><td rowspan="3">by the application software</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="3">4</td><td colspan="3">8</td><td colspan="3">16</td></tr>
<tr><td colspan="3">17.2</td><td colspan="3">35.7</td><td colspan="3">47.3</td><td colspan="3">62.3</td><td colspan="3">76.1</td></tr>
<tr><td colspan="5">32
81.1</td><td colspan="5">48
95.3</td><td colspan="5">64
98.2</td></tr>
<tr><td rowspan="3">and their sum</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="3">4</td><td colspan="3">8</td><td colspan="3">16</td></tr>
<tr><td colspan="3">18.0</td><td colspan="3">36.5</td><td colspan="3">48.1</td><td colspan="3">63.1</td><td colspan="3">76.9</td></tr>
<tr><td colspan="5">32
82.3</td><td colspan="5">48
96.5</td><td colspan="5">64
99.4</td></tr>
</table>