FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

KMedia - Network Data Encryption Using a Hardware Crypto Engine

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Ondrej Čevan
Matrikelnummer 0226686

an der
Fakultät für Informatik der Technischen Universität Wien
ausgeführt am
Institut für Computertechnik

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Mitwirkung: Dipl.-Ing. Dr.techn. Andreas Gerstinger

Wien, 30.06.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Ondrej Čevan, Rázusova 2677/12, SK-05801 Poprad

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 30.06.2010

(Unterschrift Verfasser)

Kurzfassung

Es ist das Ziel jedes Netzwerksicherheitsprotokolls die Daten, die durch das nicht vertrauenswürdige Internet transportiert werden, zu sichern. Die verwendeten kryptografischen Algorithmen können allerdings einen gravierenden Einfluss auf die Leistung eines Prozessors haben. Deswegen wurden kryptografische Beschleuniger entwickelt, um den Prozessor von den kryptografischen Operationen zu entlasten. Diese Masterarbeit präsentiert eine Netzwerkanwendung, genannt *KMedia*, um die kryptografische Leistung eines Linux Kernels, der auf einem Kommunikations-Einchipsystem-Prozessor von *Freescale* mit einem eingebauten Verschlüsselungsbeschleuniger läuft, zu messen. Die "Authenticated Encryption" Algorithmen, die in dem Beschleuniger oder in dem Linux Kernel implementiert sind, wurden berücksichtigt und bezüglich ihrer Leistung verglichen. Diese kryptographischen Algorithmen ermöglichen die Geheimhaltung der Datenpakete, die Überprüfung der Datenherkunft und die Erkennung der Datenmanipulation. Die Tests wurden auf der Anwendungsschicht eines Netzwerkstacks ausgeführt und durch Werkzeuge unterstützt die entweder frei verfügbar waren oder im Rahmen der Masterarbeit entwickelt wurden. Die Ergebnisse zeigen, dass die Leistung des getesteten Netzwerkgeräts mit dem Krypto-Beschleuniger im Vergleich zur Leistung ohne Beschleuniger rund 1,3 mal besser mit Datenpaketen kleiner als 470 Bytes und bis zu 4 mal besser mit 32768 Bytes großen Paketen ist. Mit dem Beschleuniger wurden Datenraten nahe der maximalen Bandbreite des Testnetzwerkes erreicht. Die Ergebnisse zeigen auch, dass höhere Beschleunigungen erreicht werden können wenn die Ressourcen des Netzwerkgeräts nicht völlig ausgeschöpft sind, und, dass unterschiedliche Verschlüsselungsalgorithmen mit verschiedenen Datendurchsätzen in der Software ähnliche Durchsätze unter der Verwendung des Krypto-Beschleunigers erreichen.

Abstract

The goal of every network security protocol is to secure data transmitted through the untrustworthy Internet. However, the employed crypto algorithms can have a serious impact on a processor's performance and therefore cryptographic accelerators were developed to off-load cryptographic operations from the host processor. This master thesis presents a network application, called *KMedia*, for cryptographic performance measurements in the Linux kernel running on a communication system-on-chip processor from *Freescale* with an integrated security engine. Hardware and software implemented authenticated encryption algorithms that provide confidentiality, data origin authentication and manipulation detection security services are considered and compared on performance. Tests were executed on the application layer of a network stack and were supported by freely available benchmarking tools and by utilities developed in the scope of this thesis. The results show that the device's performance, when the cryptographic transformations are executed by the cryptographic accelerator instead of by the host processor, is around 1.3 times better for messages having less than 470 bytes, and up to 4 times better for messages with 32768 bytes. With the crypto accelerator data rates close to the link limit of the testing network are reached. The results show also that higher speedups are achieved when the system's resources are not fully exhausted and that different crypto algorithms, with various throughputs in software, exhibit similar throughputs on the security engine.

Acknowledgements

Foremost, I would like to thank my mum, dad and brother as well as all other members of my big family for giving me all kinds of support from the first second of my life.

I am heartily thankful to Andreas Gerstinger for making this master thesis possible. His encouragement, constructive feedbacks and friendly consultations helped a lot to improve the quality of the work.

Special thanks go to Andreas Reisenbauer and Horst Kronstorfer for their useful inputs and for enabling me to elaborate my work at the Frequentis company. My thanks also to all employees of this company who I had the honour to meet for making my stay at Frequentis very enjoyable.

Last but not least, I would like to thank all my friends and fellow students, especially Juraj Holták, Voin Legourski, Nikolaus Manojlovic and Yilin Huang for their support and the great time we spent together.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Basic Terminology . . . 4
- 1.4 Structure of the Thesis . . . 6

2 Methodology **7**
- 2.1 From Freescale's *t23x* Driver to *Talitos* . . . 7
- 2.2 IPsec . . . 8
- 2.3 Authenticated Encryption . . . 9
  - 2.3.1 Block Ciphers and Authentication Codes . . . 10
  - 2.3.2 Generic Composition . . . 12
  - 2.3.3 Combined Mode . . . 13
  - 2.3.4 IPsec, SRTP and Talitos Authenticated Encryption Algorithms . . . 14
  - 2.3.5 Cryptographic Algorithms' Input Data . . . 16
- 2.4 Related Work . . . 19
- 2.5 Concept . . . 21

3 *KMedia* Specification **23**
- 3.1 Module Architecture . . . 24
- 3.2 Network Functionality . . . 25
  - 3.2.1 *KMedia* Header Format . . . 26
  - 3.2.2 *KMedia* Address Field . . . 27
  - 3.2.3 *KMedia* Message Format in *Plaintext* and *Forward* State . . . 29
  - 3.2.4 *KMedia* Message Format in *Ciphertext* State . . . 29
  - 3.2.5 Message Size . . . 31
  - 3.2.6 Use Case Examples . . . 32
- 3.3 Cryptographic Functionality . . . 34
  - 3.3.1 The Crypto Algorithm Inputs in *KMedia* . . . 35
  - 3.3.2 *KMedia Crypto API* Constraints . . . 36
- 3.4 Configuration and Monitoring . . . 37
  - 3.4.1 "Kernel Module" Interface . . . 37
  - 3.4.2 "sysfs" Interface . . . 37
  - 3.4.3 "printk" Interface . . . 39
- 3.5 *KMedia* Execution . . . 40

3.6 Development Tactic . . . 41
3.7 Coding Style . . . 42
3.8 Possible Extensions . . . 42

4 *KMedia* Development: Problems and Solutions 43
4.1 Encryption . . . 43
4.2 Networking . . . 46
4.3 Threading and Monitoring . . . 47

5 Testing Tools, Environment and Methods 49
5.1 Test Tools . . . 49
5.1.1 Netperf . . . 50
5.1.2 Socat . . . 50
5.1.3 KMedia Message Checker . . . 50
5.1.4 KMedia Header Engine . . . 51
5.1.5 KMedia Statistics Reader . . . 51
5.1.6 KMedia CPU Top, KMedia CPU Looper . . . 51
5.2 Test Set Up . . . 52
5.2.1 DUT Set Up . . . 53
5.2.2 Tester Set Up . . . 54
5.2.3 Discussion . . . 55
5.3 Test Messages . . . 56
5.3.1 Message Sizes . . . 56
5.3.2 Message Formats . . . 57
5.4 Performance Characteristics . . . 58
5.5 Test Environment Parameters . . . 59
5.5.1 Security Parameters . . . 59
5.5.2 Data Rates Smaller Than Throughput . . . 60
5.5.3 Tested Combinations of Parameters . . . 61
5.6 Testing Methods . . . 62
5.6.1 Forwarding . . . 62
5.6.2 Encryption Transformation . . . 63
5.6.3 Decryption Transformation . . . 64

6 Results 67
6.1 Forwarding . . . 67
6.1.1 *KMedia* vs. NAT Forwarding . . . 68
6.1.2 *KMedia* Forwarding: All Message Sizes . . . 69
6.1.3 Summary . . . 70
6.2 Cryptographic Transformation Throughput . . . 70
6.2.1 Encryption vs. Decryption Throughput . . . 70
6.2.2 Encryption Throughput . . . 71
6.2.3 CPU Load at Encryption . . . 72
6.2.4 SHA1-AES and SHA256-DES3 Encryption Throughput . . . 73
6.2.5 Talitos Performance Gain . . . 74
6.2.6 Summary . . . 75
6.3 Cryptographic Transformation Latency . . . 75
6.3.1 Socket-to-Socket Latency . . . 75

6.3.2 Pure Cryptographic Transformation Latency . . . 76
6.3.3 Summary . . . 78
6.4 CPU Load at Various Message Rates . . . 78
6.4.1 Constant Message Size, Various Data Rates . . . 78
6.4.2 Constant Data Rate, Various Message Sizes . . . 80
6.4.3 Summary . . . 80

7 Conclusion **83**
7.1 Summary . . . 83
7.2 Outlook . . . 85

A Benchmarking Tools **87**
A.1 Netperf . . . 87
A.1.1 Control vs. Test Data Connection . . . 88
A.1.2 *Netperf* Test Example . . . 89
A.1.3 Modifications in the Source Code . . . 90
A.1.4 Summary . . . 91
A.2 Socat . . . 91
A.3 KMedia Message Checker . . . 92
A.3.1 Command Line . . . 93
A.3.2 Use Case And Output . . . 94
A.4 KMedia Header Engine . . . 94
A.4.1 Configuration File . . . 95
A.4.2 Command Line . . . 97
A.4.3 Use Case Examples . . . 98
A.4.4 Using Netcat with *KMedia Header Engine* . . . 99
A.5 KMedia Statistics Reader . . . 101
A.5.1 Command Line . . . 102
A.5.2 Configuration With Macros . . . 102
A.5.3 Use Case And Output . . . 103
A.5.4 Constraints . . . 104
A.6 Tools for CPU Utilization Measurement . . . 105
A.6.1 KMedia CPU Top . . . 105
A.6.2 KMedia CPU Looper . . . 107
A.7 *Netperf*'s Patch File . . . 111
A.8 *KMedia Header Engine* Configuration File Example . . . 113
A.9 *KMedia Header Engine* Configuration File for Netcat Use Case . . . 114

B Message Generation **115**
B.1 Forward Messages . . . 115
B.2 Plaintext Messages . . . 116
B.3 Ciphertext Messages . . . 116

C *Freescale* MPC8349E Processor **118**

D *Freescale*'s Security Engines **124**
D.1 SEC 1.x . . . 125
D.1.1 SEC 1.0 . . . 125
D.1.2 SEC 1.2 . . . 126
D.2 SEC 2.x . . . 126
D.2.1 SEC 2.0 . . . 127
D.2.2 SEC 2.1 . . . 127
D.2.3 SEC 2.2 . . . 128
D.2.4 SEC 2.4 . . . 129
D.3 SEC 3.x . . . 130
D.3.1 SEC 3.0 . . . 131
D.3.2 SEC 3.1 . . . 132
D.3.3 SEC 3.3 . . . 133

E Securing Real-Time Multimedia Applications **134**

F Performance of Some Authenticated Encryption Modes **136**

G *KMedia* Development **138**
G.1 System Requirements . . . 138
G.2 Development Environment . . . 138
G.3 A Short Analysis of the Linux IPsec Implementation . . . 139
G.4 Transparent *KMedia* . . . 140

H Source Code and *KMedia* Webpage **142**
H.1 *KMedia* . . . 142
H.2 *KMedia Message Checker* . . . 142
H.3 *KMedia Header Engine* . . . 142
H.4 *KMedia Statistics Reader* . . . 142
H.5 *KMedia CPU Top* . . . 142
H.6 *KMedia CPU Looper* . . . 142

I Test Environment **143**
I.1 *KMedia* Test Message Sizes . . . 143
I.2 *KMedia* Extended Header Content Used in the Tests . . . 145
I.3 *Creator of File with KMedia Test-Payload* Source Code . . . 146
I.4 Iptables Set Up . . . 147

J Results **149**

Literature References **161**

Web References **166**

Acronyms

AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AH	Authentication Header
API	Application Programming Interface
B	Byte
BSD	Berkeley Software Distribution
CCM	Counter with CBC-MAC
CBC	Cipher Block Chaining mode
CTR	Counter Mode
CPU	Central Processing Unit
DDR	Double Data Rate DRAM
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
DES	Data Encryption Standard
ESP	Encapsulating Security Payload
FIFO	First In-First Out Pipe
GCM	Galois Message Authentication Code
GHz	Gigahertz
GNU	GNU's Not Unix
HMAC	keyed-Hash Message Authentication Code
KHMAC	keyed-Hash Message Authentication Code
HW	Hardware
ICV	Integrity Check Value
ID	Identification
IEEE	Institute of Electric and Electronic Engineers
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IP	Internet Protocol
IPsec	Internet Protocol Security
ITU-T	International Telecommunication Union-Telecommunication standardization sector
IV	Initialization Vector
KiB	Kibibyte (2^{10} Bytes)
KMedia	Kernel Module for Encryption and Decryption Inclusive Authentication
MAC	Message Authentication Code (In Section 4.2 it denotes "Medium Access Control" address used on Ethernet networks.)
MB	Megabyte (10^6 Bytes)

Mbit/s	Megabits Per Second
MD5	Message-Digest 5
MHz	Megahertz
MiB	Mebibyte (2^{20} Bytes)
MIKEY	Multimedia Internet KEYing
m/s	Messages Per Second
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OCF	OpenBSD Cryptographic Framework
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PCI	Peripheral Component Interconnect
PowerPC	Performance Optimization With Enhanced RISC–Performance Computing
RISC	Reduced Instruction Set Computer
RTP	Real-Time Transport Protocol
RFC	Request For Comments
SDRAM	Synchronous DRAM
SEC	Security Engine
SHA	Secure Hash Algorithm
SoC	System on a Chip
SRTP	Secure Real-Time Transport Protocol
SSH	Secure Shell
SSL	Secure Socket Layer
SW	Software
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	Universal Datagram Protocol
USB	Universal Serial Bus
VoIP	Voice over IP
VPN	Virtual Private Network
XCDF	eXtensible Crypto Driver Framework (Basis of Freescale's SEC drivers)
XFRM	eXtensible FRaMework (Basis of Linux IPsec Implementation)

1 Introduction

Securing data packets transmitted through the Internet is gaining on importance but increases the requirements on a network device. What impact do crypto operations have and how a crypto accelerator influences the device's performance is introduced in this chapter. Furthermore, the actual problem of this master thesis is defined and the strategy how it will be solved is presented. Also, some basic terms and concepts are explained and the following chapters are briefly described.

1.1 Motivation

"The *Internet* is a single, interconnected, worldwide system of commercial, government, educational, and other computer networks"[Shi00] that share a suite of specified protocols, i.e. a set of rules, which enable data exchange among devices connected to the networks. Nowadays the Internet represents an essential part of a work and leisure activity for many people worldwide and its popularity together with its importance for the modern society is still growing. With the raising number of Internet users and Internet services also the amount of data transmitted in forms of packets across the networks increases. This data packets do not seldom contain sensitive, private information which no other except of the sender and receiver should know of. However, such data packets must mostly "travel" through multiple external networks out of the control of the sender or receiver and the protocols, the Internet is based on, enable to intercept the data to anybody who has an access to the network through which the packet is transmitted. So, in an open packet-based Internet nobody can be sure that only the intended addressee reads the data packets. Fortunately, security services exist that ensure that no matter who accesses the packets, only the sender and the planned receiver can read and understand their content. Moreover, just to mention some more services securing data transport, the receiver can authenticate the originator of the packets and can also verify whether the packets were not modified during the transmission.

Such security services are offered by security protocols which define a set of rules for protecting data transmissions between systems. The user, or an application employing such protocol, can specify what services should be applied to a communication channel between the sender and the receiver, and the protocol takes care of the implementation. However, since providing security is an additional service, i.e. the communication would work also without it, it places an additional load on the system. Moreover, and this is more crucial to the system's performance, the security services are implemented by security mechanisms which modify content or format, or both, of every data packet before transmission and after reception. These mechanisms, as e.g. the encryption mechanism providing data confidentiality service, can be computationally very intensive

and can place a high additional load on the host processor and bring the system to a critical overloaded state, particularly in case of a low performance processor executing some real-time tasks.

In order to relieve the processor load from the "heavy" security mechanisms, special hardware units were developed. These application specific integrated circuits are mostly called *security engines* or *cryptographic accelerators*. The latter name emphasizes that the mechanisms which are off-loaded from the host processor are mostly the cryptographic algorithms and that the special purpose hardware speeds up their execution. After integrating such accelerator into a system, not only that the general purpose processor is free of the crypto algorithms it can also perform more efficient, because the data packets are cryptographically modified faster as if the processor would have to do it itself. Therefore, it could be said that a security engine increases the system-level performance. However, this is not generally true, since the advantage of using a crypto engine over the software implemented algorithms does not depend solely on the performance of an algorithm accelerator. The gain of a security accelerator which manifests itself on the system level is influenced, except by the accelerator architecture, also by the crypto device driver overhead, the cryptographic interface limitations and by the performance of the employed network protocol stack and of the application which communicates with its peer through the secured channel. As also stated in the scientific paper [fre08]: "There can be vast differences between a device's theoretical cryptographic performance and its performance in a given application."

Moreover, the extra instructions a device driver needs to execute in order to create requests for the crypto engine and then handle its response, likewise the steps the crypto unit must go through in order to configure itself for the required task and to notify afterwards the driver that it is done with the operation, can cause in certain situations, such as small data packet sizes, even the system's performance degradation, i.e. applying the security mechanism on a data packet in software on a general purpose processor would be more efficient. The system's performance degradation is even stronger if the system's cryptographic interface does not support high-level accelerator features, such as highly asynchronous operations.

Since the advantage of offloading and accelerating cryptography by security engines depends strongly also from the employed software and its overheads, it is difficult to estimate an accurate system-level performance just from a raw performance of the algorithm accelerators. Therefore, the best method is to measure the device's performance directly in an environment in which it should be used and compare it to a pure software solution executed by a general purpose processor. However, since such performance evaluations are sometimes desired before the whole software environment, or at least the application employing the security engine, is developed, a much simple test environment related to the aimed one must be used.

It is the main task of this thesis to analyze the impact of a hardware cryptographic accelerator on a system-level performance of a network device while the security protocol and a multimedia application, which should use the accelerator to speed up some security services, was not yet implemented.

1.2 Problem Statement

Cryptographic accelerators can be in form of crypto cards using peripheral buses such as PCI, or in form of security co-processors offloading general-purpose or network processors, or can be part of security-enabled processors which contain a processing core and the security accelerator in

one System on a Chip (SoC) package. These accelerators are highly asynchronous and thus work independently from the processor which can switch to other tasks until the accelerators are done with the job. The security engine evaluated in this thesis is integrated in the *Freescale* processor which is built on the SoC platform. This processor was designed for networking, communication and pervasive computing applications.

Linux with its standard kernel distribution will be used as an operating system. No extra patches or additional drivers will be applied to the kernel in order to see the actual state of a Linux support for cryptographic operations in general and particularly also for the *Freescale*'s Security Engine (SEC). On this place it can be already said that the Linux kernel, of version 2.6.27 and higher, is distributed with a SEC driver which is called *Talitos*.

The focus was to analyze the advantage a crypto accelerator would bring to a Secure Real-Time Transport Protocol (SRTP) implementation. This protocol was designed for multimedia streams and enables high throughput with low packet expansion. It provides ideal mechanisms to secure a voice communication on the Internet and is therefore particularly interesting for the *Frequentis* company which supports this thesis and is world-wide active in communication and information systems for safety-critical applications. A brief overview of commonly used security protocols and the main reasons why SRTP is recently best suited for real-time multimedia applications is given in Appendix E.

However, for the measurements, instead of employing a full implementation of an SRTP, an application should be developed which would have network and cryptographic properties similar to the SRTP. The application should run in the Linux user space, exchange network data packets with an Universal Datagram Protocol (UDP) through a socket interface and cryptographically transform the packets with the default SRTP crypto algorithms implemented either in software or in the hardware crypto accelerator.

Unfortunately, the Linux crypto interface, the only interface through which security operations offered by *Talitos* can be accessed, does not have a user space counterpart in the actual Linux kernels. Furthermore, although the *Freescale*'s security engine supports the default SRTP crypto algorithm, as defined by the RFC [BMN+04], the *Talitos* driver does not. The crypto driver was developed to accelerate some of the security operations of Internet Protocol Security (IPsec), a suite of protocols which establish secure virtual private networks. The crypto algorithms offered by *Talitos* are, according to the RFC [Man07], mandatory-to-implement or optional for IPsec, but the SRTP protocol specification does not mention them. These algorithms are similar to the SRTP's default one but are less efficient, particularly in the hardware.

Because of the above mentioned reasons, in this thesis an application in the Linux kernel space, a kernel module called *KMedia*, will be developed. It will have only the network functionality similar to an SRTP implementation as it will exchange messages with the UDP protocol through a socket interface. The crypto functionality will be more similar to an IPsec implementation, since the messages will have a format similar to the IPsec messages and will be encrypted in the kernel space with the crypto algorithms meant for an IPsec use. No online mechanism will be implemented for a secret key exchange but the kernel module will enable a manual key insertion through a user space interface.

Furthermore, a testing environment with all needed benchmarking tools will be built in which the *KMedia*'s functionality will be verified and its cryptographic performance measured. The performance of the crypto accelerator and also of the software implemented cryptographic algorithms executed by the Central Processing Unit (CPU) will be measured under various message sizes and

at diverse data rates. The obtained results will show the impact of the crypto transformations on the processor load and reflect the performance advantage of the security engine over software implemented crypto algorithms.

From the specified task the following questions arise on which this thesis will give an answer:

- Which type of crypto algorithms is offered by *Talitos*?
- Which type of crypto algorithms is used by SRTP?
- What is the advantage of using a cryptographic accelerator for off-loading security operations from the general purpose processor?
- How many times more efficient is a network device, based on the *Freescale* SoC processor, when it utilizes the Security Engine (SEC) for cryptographic operations compared to when it performs the operations on the processor core?

1.3 Basic Terminology

The scope of this master thesis is networking and cryptography in the kernel of the Linux Operating System (OS). Therefore, the terminology used throughout this thesis has its roots in a networking and cryptographic literature and a kernel source code. Some terms that are used extensively are explained in this section.

A data received through a network interface of a device has to pass different layers of a network stack. These layers, from the bottom up, are called *link*, *internet*, *transport* and *application layer* [Bra89]. The internet layer is also sometimes called as a *network* layer[1]. This protocol architecture is called "TCP/IP network stack" with the Transmission Control Protocol (TCP) denoting the widely used protocol at the transport layer and the Internet Protocol (IP) at the internet layer. The Table 1.1 lists which protocol was employed on which layer in the context of this master thesis and also how the individual data packages at each layer are called according to the book [Ben05].

TCP/IP Network Stack Layer Name	Protocol	Data Unit Name
Application Layer	(*KMedia*)	(*KMedia*) Message
Transport Layer	UDP	Segment
Internet/Network Layer	IPv4	Datagram/Packet
Link Layer	Ethernet	Frame

Table 1.1: Network Stack with Protocols this Thesis is Based on

KMedia is a kernel module developed during this thesis, which cryptographically transforms all network data received from the UDP protocol placed at the transport layer. Since the data blocks on the application layer are called *messages* and *KMedia* is a representative of that layer, all data units that *KMedia* receives, or transmits, are called "messages", "*KMedia* messages" or

[1] According to the Open Systems Interconnection (OSI) reference model [Sta04, Section 2.3].

"UDP messages". To exchange network data with the UDP protocol utilizes *KMedia* the *socket* mechanism, particularly the Linux network socket interface, called simply as "socket" or "UDP socket" in this work.

Since the thesis speaks more about the behavior of the kernel and applications than about some network abstractions, the data lengths are always expressed in units of *bytes* instead of *octets* which is a common term in the network literature. The both expressions describe data quantities of eight bits.

A network cryptographic application *overhead* are instructions a processor must execute in order to prepare the received data for, or adjust them after, a crypto transformation. This includes, e.g., determining the right sort of a crypto processing and performig an application specific formatting of the data. An application overhead exists independently from the employed crypto driver.

A brief definition of several other key terms employed in this thesis follows.

Protocol A *communication protocol* defines rules which determine the format and transmission of data. A *security protocol* defines rules how security-related functions should be applied and what cryptographic methods used.

Security Engine (SEC) A cryptographic accelerator integrated in the *Freescale*'s Processors.

Request For Comments (RFC) A memorandum describing internet systems and standards published by the Internet Engineering Task Force (IETF) organization.

Internet Protocol Security (IPsec) A suite of protocols providing security services for network connections on the network layer of the TCP/IP network stack.

Encryption "A security mechanism used to transform data from an intelligible form (*plaintext*) into an unintelligible form (*ciphertext*). The inverse transformation process is designated "*decryption*". Oftimes the term "*encryption*" is used to generically refer to both processes." [KS05].

Authentication Refers informally to the combination of integrity and data origin authentication [KS05]. An *Integrity* service detects modification of a message and a *Data Origin Authentication* is a "security service that verifies the identity of the claimed source of data." [KS05].

Host A computer connected to data network which hosts client and/or server software.

Producer A host that creates a message and sends it to another host.

Consumer A host that is an end user of the message (does not retransmit it further).

Cryptographic Server A server software offering cryptographic services to other hosts on the network and also to its own host. *KMedia* is such a cryptographic server.

Client A host requiring service, particularly message encryption or decryption with authentication, from a cryptographic server *KMedia*.

Transformation A modification of a message format and/or content, caused mostly by a cryptographic operation as, e.g., by an encryption.

Talitos Performance Denotes the performance of the Linux SEC driver, called *Talitos*, together with the performance of the SEC and its hardware implemented crypto algorithms.

Software (SW) Crypto Driver Performance Stands for the performance of the SW driver and its software implemented crypto algorithms which are executed by a general purpose processor.

Padding A process of appending some extra bytes to a message in order to align it to a size required by a cryptographic operation.

The type of the crypto algorithms, this thesis works with, integrates encryption with authentication in a specific order. Therefore, if not otherwise explicitly stated, a data *encryption* will correspond to an "encryption and the successive authentication" and *decryption* will be a short form for an "authentication and the following decryption".

Additionally, to save some place in the tables and in the figures following abbreviations were used: "KM" means *KMedia*, "TP" Throughput (the maximum data rate), "Msg" Message, "enc" Encryption, "dec" Decryption and "Tal" stands for *Talitos*.

1.4 Structure of the Thesis

The thesis is structured as follows. In Chapter 2 the cryptographic driver *Talitos* together with the IPsec, for which the driver was developed, is introduced. Moreover, the cryptographic algorithms used for a network data encryption by the IPsec and SRTP protocol will be explained and compared to the ones which the *Talitos* driver offers. Afterwards, the chapter presents some work and results reported already by other diverse scientific publications dealing with the evaluation of a possible gain a crypto accelerator can bring to a communication system. At the end, the chapter discusses the main idea of the thesis and the difference to the already performed research in this field. The thesis follows with Chapter 3 in which a complete design specification of the kernel module developed for measuring the crypto performance of the *Freescale*'s security engine with the help of the *Talitos* driver is presented. In the next Chapter 4 the obstacles which were experienced during the module's development and implementation are discussed together with their solutions. The whole testing environment is described in detail in Chapter 5. The chapter involves benchmarking tools, the parameters of the tested- and testing-device and also the definitions of the tested performance characteristics along with the measurement techniques used for the module's performance evaluation. All results obtained from the performance measurements are presented and analyzed in Chapter 6. The main part of the thesis ends up with Chapter 7 which summarizes the master thesis' key results and presents some ideas for further work.

Additionally, the appendices describe many parts of the thesis much more detailed as the main part. Except others, the architecture and the usage of the benchmark tools is explained together with the methods how the test messages were created, in Appendix A and B. Also, in Appendix D, features of all versions of the security engines, as offered by the *Freescale* company, are listed and compared among themselves. Appendix H mentions were the source codes of the *KMedia* kernel module and of the testing tools can be found and Appendix J lists all values measured during performance tests. At the very end, the scientific papers, literature and websites are listed which are referenced within thesis and provided an invaluable source of information.

All source code and documents created during this master thesis are published on the website [1].

2 Methodology

Since in this master thesis the performance of a cryptographic accelerator is measured, this section explains exactly what cryptographic algorithms will be tested on the accelerator and how an application providing confidentiality and authentication on network data can make use of them. First, the available drivers for the Security Engine (SEC) of a *Freescale* processor are briefly analyzed and it is mentioned how the *Talitos* driver, the application developed in the scope of this thesis uses in order to access the crypto algorithms implemented in the SEC, was discovered. Since *Talitos* offers its functionality natively to IPsec, also an overview of this set of security protocols is given. Afterwards, the so-called "authenticated encryption algorithms" used by the IPsec and SRTP network security protocols are analyzed and compared to the ones which can be accessed through the *Talitos* driver. The chapter continues with a presentation of some work and results reported already by other diverse scientific publications dealing with the evaluation of a possible gain a crypto accelerator can bring to a communication system. At the end, the chapter discusses the main idea of the thesis and the difference to the already performed research in this field.

2.1 From Freescale's *t23x* Driver to *Talitos*

A goal preceding this master thesis was to develop a Linux driver for a cryptographic accelerator called "SEC" and implemented in some *Freescale*'s SoC processors. The driver should be designed in such a manner that it would have a perspective of being added into the kernel source tree. Additionally, the driver should be also used to evaluate the SEC engine on performance. During the preliminary work, when information on *Freescale*'s SECs was gathered, it was found out not only that *Freescale* offers a proprietary driver package, what was expected, but also that there already is a similar driver in the Linux kernel.

The *Freescale*'s driver package is called "*t23x*" and can be freely downloaded from the website [10]. According to the user's guide[1] delivered with the package, the driver's primary purpose is "to serve as an example implementation of a driver that can be used either in whole or in part by application developers". It is built upon a framework, the "eXtensible Crypto Driver Framework (XCDF)", which allows to share the SEC's core resources to multiple system components, be it user-mode applications, kernel modules or protocol stacks. The package underlies *Freescale*'s licencing and although the "Licence Agreement" which must be accepted before a download states that it can

[1] version 1.0.0, January 2008

be redistributed in an object form (machine-readable) only, the copyright notice in the source code allows redistribution also in source.

The *t23x* user's guide mentions that *Freescale* uses internally for the SEC crypto accelerators another term too which "appeared also in Linux projects involving the SEC core". This name is "*Talitos*". Searching with *Google* [15] revealed that Linux kernels from the version 2.6.27 contain a crypto driver of the same name and that it is a driver for the *Freescale*'s integrated security engine. The driver registers its algorithms with the "*Linux Crypto API*" so they can be accessed by any system component through the same interface. However, up to the kernel 2.6.33 they are primarily designed for use with the Linux implementation of *IPsec*. This restriction concerns the supported crypto algorithms and also the format of the data intended for crypto transformations. *Talitos* can be redistributed and/or modified under the terms of the "GNU General Public License" version two, or later, published by the "Free Software Foundation" [9].

Since with *Talitos*, being the SEC driver already distributed within the Linux kernel source, the planned main goal of thesis diminished in value, it was changed to the one explained already in the Introduction chapter. A kernel module which became the name *KMedia* will utilize the encryption algorithms supported by *Talitos* in order to cryptographically transform a network traffic, the UDP messages, with the *Freescale*'s security engine.

2.2 IPsec

After the *Talitos* driver was introduced, the IPsec suite of protocols will be described for which the crypto driver was initially developed.

There are many documents on IPsec as the website [31] tries to summarize, but reading the official RFC [KS05, Section 3.], describing the fundamental architecture, and an illustrated guide on the website [35] gives enough information in order to understand the basic internal components of IPsec[2]. Important for this thesis is that IPsec, an extension to the IP layer which provides security services for network connections, is composed of the following three main protocols: Internet Key Exchange (IKE), Authentication Header (AH) and Encapsulating Security Payload (ESP).

The IKE protocol establishes security associations and enables a secret key exchange. Since the focus of this thesis lies on the performance of cryptographic algorithms during encryption of network data and not on the performance of the secret key exchange mechanisms, the IKE protocol was not further analyzed.

The AH and ESP can be used independently or together in two possible modes: *tunnel* or *transport*. The AH ensures integrity of a secured IP packet and authenticity of a node which transmitted the packet. The protocol guards optionally also against replay-attacks. Equal properties are provided also by the ESP protocol[3] but additionally also with confidentiality (encryption) which makes some parts of the IP packet, particularly the payload, illegible for an attacker. The *transport* mode is applied for securing a host-to-host communication and the *tunnel* mode is used when the hosts communicate through security gateways in which case only the communication

[2]The articles as [FS03] from the year 2003 and [PY05] from 2005 describe IPsec as a very complicated suite of protocols. The first one states: "Even though the protocol is a disappointment–our primary complaint is with its complexity–it is the best IP security protocol available at the moment." and the second demonstrates some successfully performed attacks against the Linux implementation of IPsec.

[3]Except that ESP does not authenticate the IP header as AH does.

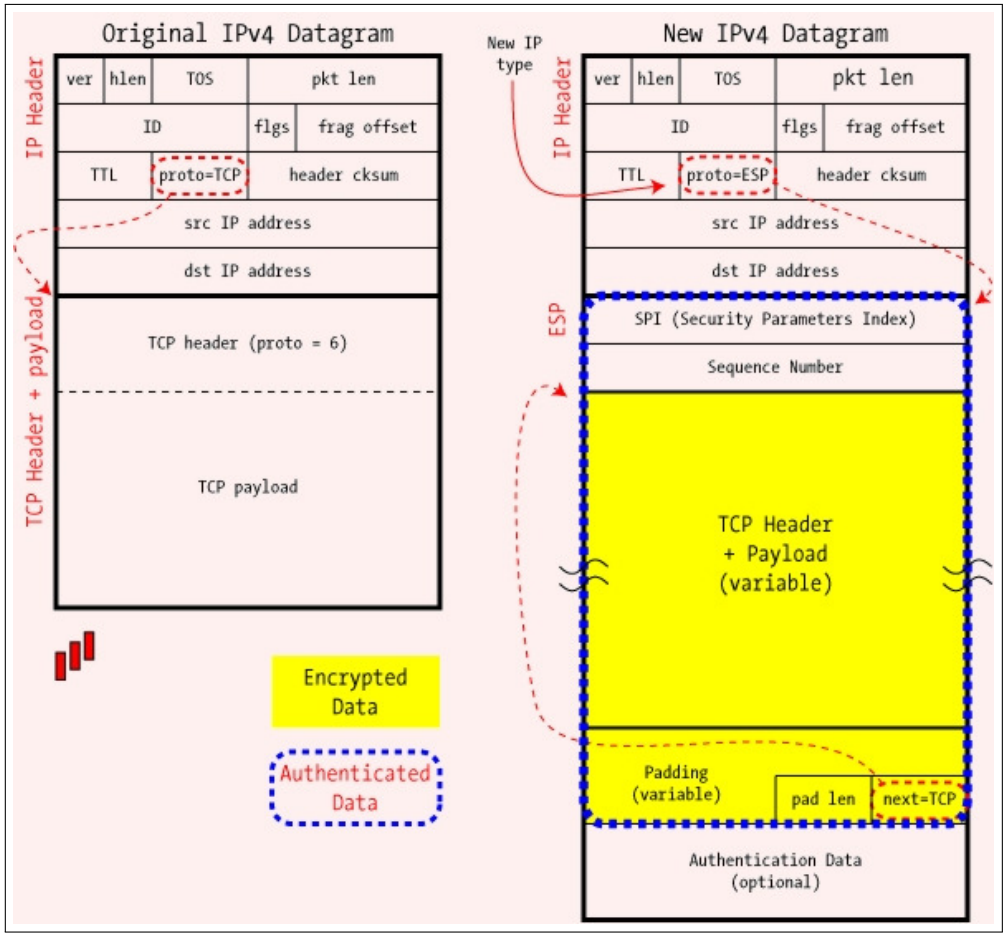

Figure 2.1: IPsec in ESP Transport Mode; Figure from the Website [35]

between the gateways is secured. The main difference between the two modes when handled by the ESP protocol is that in the *transport* mode solely the UDP/TCP segment is encrypted, while in the *tunnel* mode the whole IP packet is encapsulated inside the encrypted shell and a new IP header is built over it. The Figure 2.1 illustrates an IP packet before and after an ESP transport mode transformation. The formats of these IP packets before and after an ESP transformation are particularly important for the design of the messages which the *KMedia* module will have to process.

To summarize, the AH performs only an authentication and the ESP an authentication together with an encryption of an every received network data packet. Since *KMedia* should provide a confidentiality security service, the cryptographic algorithms used by ESP will be described in the next section.

2.3 Authenticated Encryption

The crypto algorithms employed in the Encapsulating Security Payload (ESP) protocol of the IPsec suite belong to the group of symmetric and randomized Authenticated Encryption with Associated Data (AEAD) algorithms.

An "*Authenticated Encryption*" algorithm combines, as its name says, encryption with authentication. The encryption transforms a message from plaintext into ciphertext and ensures *confidentiality* of the message. The authentication consists of computing a data signature over the message which enables on the receiver side to determine whether the message was not illegally modified during the transmission, a security service known as an *integrity check*, and whether the message comes from someone with a proper authentication key, a *data origin authentication* service. Note, that the combination of the *integrity check* and of the *data origin authentication* is defined simply as an *authentication* security service. An AEAD algorithm adds the feature that some *associated data*, also called "*additional authenticated data*", can be checked for integrity and authenticity without being encrypted.

These crypto algorithms have a *non-deterministic property*, i.e. every time the same block of plaintext is encrypted the resulting ciphertext is different. Therefore, they are also termed to be "*randomized*". Naturally, the decryption process is deterministic otherwise it would be not possible to decrypt the ciphertext.

If two parties want to exchange messages through a communication channel secured by these algorithms they have to share the same copy of a bit-string, which is called "secret key" or just as a "key". This is the property of algorithms based on the *symmetric-key model* to which also the AEAD algorithms belong. There are couple of methods how these keys could be generated and distributed to the communicating parties but they will be not described here.

Three approaches exist how to achieve the two security services, confidentiality and authentication, simultaneously, or, in other words, how an authenticated encryption algorithm could be built. They are called "Generic Composition", "Single-Pass Combined Mode" and "Two-Pass Combined Mode" and will be explained in the subsequent sections. All the presented modes are provable secure. However, first the symmetric encryption algorithms and hash functions will be described which are essential building blocks for the modes.

2.3.1 Block Ciphers and Authentication Codes

The authenticated encryption algorithms described in this thesis provide data confidentiality with symmetric encryption ciphers that work on fixed-length groups of bits, i.e. data blocks, and are therefore called "block ciphers". A data string larger than a block size is transformed as a sequence of blocks. If these algorithms are used for an encryption, they turn plaintext into ciphertext and, vice-versa, if used for a decryption they turn ciphertext into plaintext. For a successful decryption the same secret key as used for encryption must be used. The basic block ciphers will be now described according to the information from "Encyclopedia of Cryptography and Security" [Til05].

AES The "Advanced Encryption Standard" block cipher works on 128 bits long data blocks and supports key lengths of 128, 192 and 256 bits. It exploits parallelism, achieves a very good performance in software and also in hardware and allows a fast key setup. The cipher was chosen by the United States' National Institute of Standards and Technology (NIST) as an encryption standard on October 2000.

DES3 or Triple-DES The "Triple-Data Encryption Standard" block cipher was approved by NIST in 1999, designed to enhance the original single DES, which was used as a standard since the year 1977. The single DES employs only a 56-bit secret key and does not provide an

adequate security anymore. The Triple-DES is based on a triple execution of the single DES, a data block is first Encrypted, then Decrypted and finally again Encrypted (DES3_EDE). For each transformation a different, independently generated, key should be used so that the secret key for the DES3 is 168 bits[4] long. The Triple-DES cipher operates, as the single DES, on 64-bit data blocks, is inherently sequential, rather slow and less secure than the AES.

The authentication security service is provided by *Message Authentication Codes (MACs)* which use for their operation a secret key shared between a sender and a recipient. A sender uses its secret key and the MAC algorithm to compute a signature over a message and appends it afterwards to the message. After a reception, the receiver uses its own key and the MAC to compute the signature over the received message and compares its output with the signature that was appended to the message. If the signatures equal, the authentication was successful. This means that the data origin authentication with MAC is based on the assumption that only the trusted sender uses the same MAC-key as the receiver of the message. The MACs cannot denote the origin of a message in an irrefutable way, the sender authenticates a message the same way and with the same key as the recipient, but these algorithms are up to three times of magnitude faster and produce outputs which are only 4–16 bytes long compared to the asymmetric cryptographic techniques[5] creating 40–128 bytes long digital signatures [Til05].

Some dedicated MAC algorithms exist, but they are mostly derived either from block ciphers as the AES and DES3, or from *cryptographic hash functions*, i.e., functions which "take input strings of arbitrary (or very large) length and map these to short fixed length output strings" [Til05]. These cryptographic hash functions have another requirements as the "non-cryptographic" hash functions used, e.g., in hash tables or caches for allocating the storage place of a certain data record. In hash tables it is possible, sometimes even desired, that input strings that differ by a little produce the same, or nearly the same, hash output. This feature would be considered as a fatal flaw in a cryptographic hash function. In the following paragraph some cryptographic hash functions are described.

MD5 "Message-Digest 5" operates on 512-bit message blocks divided into 32-bit words and produces a 128 bits long message digest. After padding, a message is processed block by block by a compression function which allows for parallelism. The MD5 arose after some security issues of its predecessor, MD4, became apparent. However, it was shown that MD5 is also not collision resistant, i.e. different messages can lead to the same MD5 hash output. The work of Marc Stevens et. all [SLW07] proves that at least one practical attack scenario can exploit the collision issue in practice. Correspondingly, the MD5 is not approved for use by the United States' federal agency NIST.

SHA "Secure Hash Algorithm" is available in two versions: SHA1 and SHA2. The SHA1 consists of a crypto hash function which produces a 160 bits long output. To the SHA2 version family belong: SHA224 with a 224 bits long message digest, SHA256 with 256 bits, SHA384 with 384 bits and SHA512 with 512 bits respectively. They all use iteratively a compression function, the input to which consists of a chaining variable and a 512- or 1024-bit message block. Internally, the compression function operates on 32 or 64-bit words. The chaining variable is updated with every iteration and contains the message digest at the end

[4] 3x56 bits

[5] For example, the RSA (Rivest-Shamir-Adleman) digital signature scheme.

of the hashing process. Since the compression function consists of some, possibly parallel, rounds, all SHA hash functions allow for parallelism [BGV97]. Note, that although all of these SHA algorithms are certified by the NIST [oST08], weaknesses have become apparent in SHA1 and therefore the NIST "encourages a rapid adoption of the SHA-2 hash functions for digital signatures, and, in any event, Federal agencies must stop relying on digital signatures that are generated using SHA-1 by the end of 2010."[27]. A hash competition process has been already started by the NIST to develop a new hash function standard, the SHA3, by the year 2012.

The cryptographic hash functions listed here do not use a secret key and provide only manipulation detection service (integrity check). They can prove if a message was modified during transmission but cannot authenticate the sender. However, they can be used in an operation mode which requires a key and thus adds the data origin authentication service to the integrity check service. A cryptographic hash function executing in such a mode becomes a MAC function, as explained above. This mode, among some others, will be explained in the next section.

2.3.2 Generic Composition

The most straightforward and traditional approach to compose an authenticated encryption algorithm is provided by the *Generic Composition* scheme. A standard symmetric encryption cipher is employed for message encryption and decryption and a standard MAC algorithm for building the message signature. There are more composition methods, namely *Encrypt-and-MAC*, *MAC-then-Encrypt* and *Encrypt-then-MAC*, all analyzed on security in the paper [BN08], but mostly, because of better security qualities determined also by the paper [CK01], the "*encrypt-then-MAC*" method is used. First the plaintext data are encrypted with an encryption algorithm and then the resulting ciphertext is authenticated together with the associated data by a MAC algorithm.

Within this scheme, operation modes are used that define, on the one hand, how strings of arbitrary lengths should be transformed by block ciphers which encrypt only fixed, n-bits long data blocks, and on the other hand, how the Message Authentication Codes (MACs) should be created from the cryptographic or non-cryptographic hash functions and block ciphers. In the subsequent paragraphs some of these modes are described according to the articles [Bla05, BMN+04] and the website [37].

CBC The "Cipher Block Chaining" mode has little overhead. This mode transforms a message block-by-block with a block cipher and is therefore inherently serial.

CTR or CM The "Counter" mode, also known as an Integer Counter Mode (ICM), has little overhead and allows for parallelism, i.e. more blocks can be transformed at the same time. Moreover, the mode does not require any plaintext padding as the CBC mode.

OFB The "Output Feedback" mode does not need any padding but is serial. It needs only the encryption operation of a block cipher.

f8-mode The "f8-mode" is an elaborated version of the OFB. This mode is used to encrypt data in the 3G mobile networks of the Universal Mobile Telecommunications System (UMTS).

The next modes help to construct and accelerate MACing:

CBC-MAC "Cipher Block Chaining Message Authentication Code" is a cipher based message authentication code. It is quite simple and just as fast as an encryption in the CBC mode and inherently sequential.

HMAC or KHMAC "Keyed-Hash Message Authentication Code" uses a cryptographic hash function in combination with a secret key to process a message. It is faster than the CBC-MAC and, moreover, the fastest software MAC in a common use today.

Wegman-Carter This mode involves using a non-cryptographic hash function, which is chosen randomly from a set of hash functions that share a common domain and range, to process a message and then a cryptographic function to process the hash output. The fastest known MACs are based on this approach.

Many notations are used to refer to the result of an authentication operation: data signature, message signature, Integrity Check Value (ICV), hash value, message digest, cryptographic checksum, HMAC output, authentication code or simply just MAC.

As an example of an AEAD algorithm composed according to the *generic composition* scheme, the algorithm "HMAC-SHA1-CBC-AES" [McG09] can be considered. Its name depicts that the SHA1 in the HMAC mode is used for an authentication and the cipher AES in the CBC mode is responsible for an encryption transformation. Except of the above mentioned operation modes, some more exist, for both authentication and encryption operations, a list of the approved ones can be found on the website of the NIST [30].

Since, in order to provide an authenticated encryption according to the *generic composition* scheme, two different algorithms with two different secret keys need to make two separate passes over a message, a significant effort was made to find an operation mode which would need just one pass over the message. These approaches are called "*Single-Pass Combined Modes*" and will be briefly described in the next section.

2.3.3 Combined Mode

"Under many circumstances, combined mode algorithms provide significant throughput and efficiency advantages." [Man07]. This subsection describes some combined modes according to the articles [Bla05, MV04].

The following examples of *Single-Pass Combined Modes* achieve performance comparable to that of an encryption alone. They do not work with the associated data per se, i.e. they cannot be used to build AEAD algorithms. However, some methods have been already proposed how this feature could be brought also to this type of modes.

IAPM "Integrity-Aware Parallelizable Mode" is the first provably secure single pass mode. Uses two keys and its optimized version is nearly so efficient as the encryption in generic composition approach alone.

OCB "Offset CodeBook" is based on the IAPM with a long list of improvements. It requires only one secret key and does not need padding. The OCB is about 6.4% slower than an encryption in the CBC mode without exploiting the parallelism it offers.

These modes are highly efficient, but they are covered by patents and that is maybe the reason why they are not in a widespread use. However, the complications associated with the patents motivated some researchers to work on another efficient authenticated encryption algorithms that would be free of intellectual property. Although they need two-passes, some of them, particularly the GCM mode, can provide even higher thoughputs than the single-pass combined modes, both in hardware and software. The following modes belong to the class of *Two-Pass Combined Modes* which are patent-free:

CCM "CBC-MAC with Counter Mode" is a combination of CBC-MAC for authentication and CTR for encryption. Compared to the *generic composition* scheme it offers some advantages, in particular, it needs only one secret key. The mode is considered by NIST as a standard and is mandatory in the IEEE 802.11 wireless standard with the AES used as the block cipher (CCM-AES).

EAX "Encrypt-then-Authenticate-then-(X)Translate" is an improved CCM. It uses the CTR and a variant of the CBC-MAC and is simpler to specify and implement.

CWC The "Wegman-Carter MAC with CTR" mode uses a common secret key and is, in comparison to CCM and EAX, fully parallelizable. This mode is very fast in hardware.

GCM "Galois/Counter Mode" [MV04] can be used, except for an authenticated encryption, also for an authentication only. It needs just one secret key and attains a very high performance on short packets. This mode reaches for 40 byte messages up to 15 times higher throughput in software compared to the CBC-HMAC generic composition mode. Its authors claim that with this mode speeds of 10 Gigabits per second and more can be achieved in hardware. GCM is a preferred mode in the IEEE 802.1AE standard, known as MACsec.

In the article [MV04] some of these just described modes were measured on performance. The results are provided also by Appendix F on the page 136.

2.3.4 IPsec, SRTP and Talitos Authenticated Encryption Algorithms

After the authenticated encryption algorithms together with some of their operation modes were briefly described, the default algorithms used by the IPsec and SRTP security protocols will be listed and compared. In the scope of this thesis only the algorithms offered by the *Talitos* crypto driver, which are mandatory-to-implement or optional in IPsec, will be measured on performance. Therefore, this section should help to understand what can be expected from the default SRTP algorithms compared to the ones offered recently by *Talitos*.

The mandatory authenticated encryption algorithms for IPsec must be composed, according to the RFC [Man07], from the HMAC-SHA1 authentication and the CBC-AES or CBC-DES3 encryption algorithm. It is recommended, but not required, that the IPsec implementations support also the CTR-AES cipher for encryption and XCBC-MAC-AES, which is an extended CBC-MAC operation mode of the AES cipher, for authentication. Optionally, also the HMAC-MD5 authentication algorithm can be offered. Apparently, these algorithms work only in the *generic composition* scheme. The *combined mode* algorithms were not suggested for use by the RFC, but "CCM-AES and GCM-AES are of interest", so it is likely that they will be recommended

in the near future. Beyond these algorithms also some other crypto algorithms can be added to IPsec by the developers.

For the SRTP protocol the HMAC-SHA1-CTR-AES generic composed algorithm is defined as mandatory and default-to-use in the RFC [BMN+04]. Optionally, the AES can execute in f8-mode instead of in the CTR mode. No other crypto algorithms were suggested by the specification. Of course, as also in the case of the IPsec, additional standard and proprietary algorithms beyond those can be implemented.

As for the *Talitos* driver, the following Table 2.1 lists all its supported AEAD algorithms in the Linux kernel 2.6.29 which was used for this thesis. The names are a combination of available symmetric block ciphers operating in the Cipher Block Chaining (CBC) mode and cryptographic hash functions executing in the HMAC mode.

HMAC-SHA1-CBC-AES
HMAC-SHA1-CBC-DES3
HMAC-SHA256-CBC-AES
HMAC-SHA256-CBC-DES3
HMAC-MD5-CBC-AES
HMAC-MD5-CBC-DES3

Table 2.1: AEAD Algorithms Offered by *Talitos* in Linux kernel 2.6.29

From the above description follows, that IPsec could be also used with the default SRTP algorithm HMAC-SHA1-CTR-AES. However, this one is not supported recently by *Talitos*. The only important difference between the default SRTP and all other algorithms in *Talitos* is the operation mode of the symmetric block cipher. While *Talitos* offers only CBC, the SRTP requires CTR. As it was already described in Section 2.3.2, CBC is inherently serial while CTR is fully parallelizable. Therefore, it is expected that CTR would achieve higher efficiency particularly in hardware.

Unfortunately, no documents were found on the Internet that would directly compare the performance of the CTR and CBC mode in hardware or in software, except of the one website [2]. The site compares various operation modes of the AES cipher implemented in the *Crypto++ Library*, which is described on the same website. The measurements were executed on an quad-core processor and therefore also the parallelizable nature of the CTR mode could take effect. The results, placed in Table F.3 on the page 137, show for the CTR mode up to 1,34 times higher throughput than for the CBC mode.

Apparently, on a multi-core processor the CTR mode is more efficient than the CBC mode. However, since no benchmarks were found, that would compare the hardware performance with the software performance of the CTR mode, no direct statements can be said on the possible gain a cryptographic accelerator executing a cipher in this operation mode would bring to a system. On the other hand, in an environment with just one processor core the CTR cannot execute fully parallel. Therefore, it can be expected that the gap between the CTR performance in hardware, which enables concurrency, and software, without concurrency, will be higher than as it will be measured in this thesis with the CBC mode which executes in both cases serial.

2.3.5 Cryptographic Algorithms' Input Data

The driver *Talitos* supported, at the time of working on this thesis, only AEAD crypto algorithms composed of block ciphers running in the CBC mode and of hash functions executing in the HMAC mode. Therefore, only this type of authenticated encryption algorithms will be further described and used throughout the whole thesis. This section looks closer on the input data that must be passed to these algorithms. An extra subsection is devoted also to the secret key usage. It is possible that the input parameters described in this section relate also to other operation modes of the authenticated encryption algorithms. However, the emphasis lies solely on the AEAD algorithms as offered by *Talitos* in the Linux kernel 2.6.29.

2.3.5.1 Encryption

In order to use AEAD algorithms for an *encryption*, the following input data must be provided:

- AEAD Algorithm Name
- Secret Key
- Plaintext
- Associated Data
- Initialization Vector (IV) (optional)
- Nonce for IV (optional)

Every algorithm that should be employed for an encryption transformation is identified by its *AEAD Algorithm Name* as can be seen in Table 2.1. A cipher key and a key for an authentication will be generated from the *Secret Key* by the available crypto driver, either *Talitos* or software driver. *Plaintext* represents data that should be encrypted and then authenticated together with the *Associated Data*.

The *IV* is used as cryptographic synchronization data and can be specified optionally. It has the same bit length as the data block of a block cipher and need not be secret. In order to preserve the non-deterministic property of the crypto algorithm the IV should be random and different for each data packet chosen for an encryption. If one IV is used for two identical data packets, then the resulting ciphertext will equal, which could be helpful for debugging or testing purpose but should be otherwise avoided. The preferred way is to let the crypto device generate the IV at the transformation of each data packet.The process of an IV generation requires a *Nonce*, a one-time-used number which has the value of a sequence number[6] in the Linux implementation of IPsec.

Output of the AEAD encryption operation is a ciphertext with a message signature (ICV) appended to it and a generated IV, in case it was requested. The ICV transmitted together with the transformed message is not in its full length. According to the RFC [KBC97], the truncation of an HMAC operation output is a recommended well-known practice with message authentication codes although "The results in this area are not absolute as for the overall security advantages of

[6]Sequence numbers increase by one for each network data packet and are primarily used for a replay protection.

truncation." [KBC97]. The truncation has some advantages, e.g. an attacker has less information on the hash result, and disadvantages, e.g. the attacker has fewer bits to predict. The truncated output should be not less than half the length of the hash output and not less than 80 bits.

2.3.5.2 Decryption

An AEAD *decryption* transformation needs the following inputs:

- AEAD Algorithm Name
- Secret Key
- Ciphertext with Appended ICV
- Associated Data
- Initialization Vector (IV)

The inputs: *AEAD Algorithm Name*, *Secret Key*, *Associated Data* and *IV* denote the same as the equivalent inputs for an encryption transformation. These inputs must have the same values as used for the encryption otherwise the decryption operation will fail. If the IV was generated, then it must contain the generated value. The *Ciphertext with Appended ICV* carries the encrypted data and the message authentication code which is used for an integrity and authenticity check. At the beginning of a decryption transformation a new ICV is calculated by applying the HMAC with its authentication key, generated from the Secret Key, on the Associated Data and the Ciphertext. If the leftmost bits of the computed hash output match the received truncated ICV then the ciphertext is decrypted using the, from the Secret Key generated, encryption key and the IV. In case they are not equal, an error code is returned and no data will be decrypted [Ken05].

However, the Freescale's SEC 2.0 engine employed in this thesis decrypts a ciphertext even though it is not known whether the computed ICV matches the received one. First after the decryption, the *Talitos* driver checks both ICVs and decides whether the message can be accepted or should be discarded. Later versions of the crypto device, SEC 2.1 and higher, should be already able to make the check in the hardware.

2.3.5.3 Secret Keys

The secret keys should be changed regularly. The more bytes are processed with one fixed key value, the more the security of a crypto algorithm in a block cipher mode of operation degrades. This is due to the birthday paradox. The more data blocks are processed the higher is the chance that there will be at least two data blocks that are equal. This could help an attacker to decrypt the messages. In case of CBC-AES, the key should not be used to protect more than 2^{64} bytes of data according to [McG09, Page 10]. For the CBC-DES3 algorithm the key is limited to at most 2^{32} block encryptions according to the paper [FS03, Page 13].

The sizes of the individual keys used for respective algorithms, supported by *Talitos*, are summarized in Table 2.2. An AEAD secret key length is the sum of the encryption key size and the authentication key size. Although the block cipher AES offers three sizes of key lengths, "a key size of 128 bits is considered secure for the foreseeable future" [FGK03].

Encryption Key Sizes:	
	AES: 128, 192, 256 bits
	DES3_EDE: 192 bits (3 keys, each 64 bits long)
Authentication Key Sizes:	
	SHA1: 160 bits
	SHA256: 256 bits
	MD5: 128 bits

Table 2.2: Encryption and Authentication Key Sizes

The DES3 algorithm uses 3 keys, each having 64 bits. However, the real length of every key is actually only 56 bits, as it was already noted in Section 2.3.1, the remaining 8 bits should be used for an error detection with a parity check mechanism [KMS95].

The HMAC functions used for an authentication could have keys of various sizes, but according to the RFC [KBC97], it is strongly recommended to use keys at least as long as their hash output. On the other side, if they are longer they do not significantly increase the function strength.

2.3.5.4 Input Data Format

The *Freescale*'s Security Engine SEC 2.0, described in Appendix D.2.1, allows to place the input and output data for a crypto operation in different memory locations. However, *Talitos* from the Linux kernel 2.6.29 requires that in one contiguous block of memory, the *Associated Data* immediately precede the *Initialization Vector (IV)* which is directly followed by the data to be ciphered and authenticated. This implies also, that the data designated only for an authentication cannot use the scatter/gather feature of the *Freescale*'s Security Engine and must lie in one contiguous memory block. On the other hand, the data to be ciphered and authenticated can use the scatter/gather capability and thus can be stored in several separate segments of memory, under the condition that the first block of the data parcel is placed immediately after the IV. The following Figure 2.2 visualizes the described data structure.

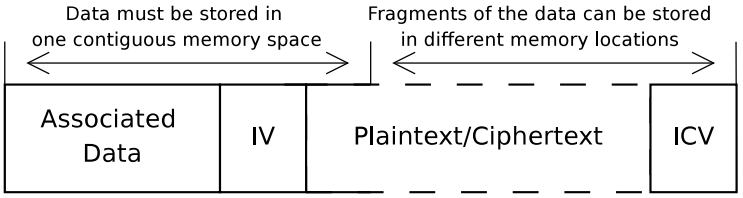

Figure 2.2: Input Data Format

The format is the same as for an encryption so for a decryption transformation, just the content of the data fields changes. Notably, during encryption and authentication, the crypto algorithm will write the appropriate values into the IV and ICV fields and will change the plaintext into ciphertext. On the other hand, during decryption, values from IV and ICV fields will be read out and used for the authenticity check and a subsequent decryption of the ciphertext.

2.4 Related Work

In the previous sections the cryptographic algorithms and security protocols were introduced which will be considered throughout this thesis. Since the goal is to evaluate the advantage a security engine can bring to a system which secures network traffic, this section looks at the research performed already in this field, i.e. what system speedups were observed when cryptographic operations were offloaded from a general purpose processor to a dedicated crypto accelerator.

Unfortunately, no reports were found that would inspect cryptographic accelerator performance and its advantage over software encryption in the context of SRTP. This does not hold for IPsec which is popularly analyzed on performance under diverse crypto accelerators and optimizations. Therefore, also the IPsec related work is presented in this chapter. On one side, the IPsec and SRTP performance cannot be directly compared, because, except of architectural diversities, SRTP is implemented usually in the Linux user space and communicates through a socket with the UDP protocol in contrast to IPsec which is directly implemented at the network layer of a TCP/IP network stack in the Linux kernel space. On the other side, these two protocols perform similar security services, as confidentiality and authentication, using the same type of cryptographic algorithms and therefore also the results obtained with IPsec help to get some main characteristics of a crypto accelerator which should not be all too different when obtained with the SRTP protocol.

The following work [LP06] describes briefly cryptographic hardware, from crypto cards, mostly in a PCI form, through security co-processors offloading general-purpose or network processors over to security-enabled security processors which contain a processing core and a security accelerator in one SoC package. Afterwards, the authors note that "The proximity of the device to the host processor core is usually a good indicator of its affect on cryptographic latency." The Linux port of the described "OpenBSD Cryptographic Framework (OCF)"[7], the "Linux-OCF"[34], is for consideration since to its main features belong, except asynchronous service, also session caching and load balancing to maximize the throughput on multiple crypto devices. Moreover, the crypto interface enables an access to the native Linux Crypto API from the user space. In the context of the crypto performance it is stated that the overhead associated with the processing of small packets in the hardware may overcome the benefit. As an example the authors mention that encrypting packets which are shorter than 32 bytes on the accelerator of the *Freescale*'s MPC8555 processor takes more time than encrypting them with the processor core.

The work [AVJ05] analyzes the IPsec performance at different configurations without using a hardware accelerator. The results were obtained by sending a file which was one Gigabyte long in datagrams of a 1500 byte size between two Personal Computer (PC)s. In case the computers communicated through 100 Mbit/s network interfaces the CPU exposed to be the factor limiting a higher IPsec performance. Compared to the state when no security was provided, enabling only an authentication service caused a 5.6% fall in the maximal achieved data rate (throughput), in an encryption-only case this decrease raised to 17% and, finally, when the data in the transmission was both encrypted and authenticated the throughput was 40% lower. For the authentication the HMAC-SHA1 algorithm with a 160-bit secret key and for the encryption the AES with a 128-bit key was utilized. The authors do not mention in what operation mode the AES cipher was running, so it can be only suggested that it was the CBC mode as it is the default one in IPsec. When the authentication algorithm was employed with a stronger secret key (256-bits) the throughput at the authenticated encryption dropped by 61% with respect to a non-IPsec

[7] According to the [fre08], "the best open source crypto API".

configuration. In all cases, the processor load was about 100%. On the other side, when the network bandwidth was bounded to 10 Mbit/s the CPU was no longer the limiting factor and with all IPsec configurations a throughput close to the link limit was reached. The processor load was twice so high during an authenticated encryption than during a non-IPsec state.

Contrary to the above presented results, the authors of the work [AWK09] observed, that "authentication is more expensive than encryption in terms of processing time". This means, an authentication-only configuration achieves lower throughputs than an encryption-only state. However, the results were gathered from execution time measurements of the SRTP protocol implemented on a bare PC softphone with no operating system. During the tests the phone had to cryptographically transform 10.000 voice packets bidirectionally. Each packet was 160 bytes long and was transformed with the CTR-AES, using a 128-bit secret key, encryption and HMAC-SHA1, using a 160-bit key, authentication algorithm. Furthermore, with respect to the CTR-AES with a 128-bit key, increased the processing time by 10% when the block cipher used a 192-bit key and by 20% when a 256-bit long key was employed. Overall, the authors measured that the SRTP processing adds less than 1 ms to the RTP processing and has no observable effect on the voice quality. Note, although encryption-only configurations were tested, they should never be used in practice since they are very vulnerable [PY05].

The authors of the scientific paper [TRC08] use a mobile application processor SoC with a programmable security engine to evaluate the efficiency of crypto offloading with two different IPsec implementations. One executes in the Linux user-space and the other in the kernel-space. In a 100 Mbit/s network they use the HMAC-SHA1 as an authentication algorithm and the CBC-DES3 as an encryption algorithm for securing data packets generated by the *ttcp* tool[25]. Their results confirm the results of the above presented work [AVJ05] that the encryption transformation is "heavier", i.e. requires more resources, than the authentication. Furthermore, "The percentage of processing time spent by the host processor in crypto operations increases with the packet size". Packets with 64 bytes spent 47%, with 256 bytes 66%, with 1 KiB 84% and with 16 KiB already 96% of the total processing time with the authenticated encryption. The remaining time is consumed by the non-cryptographic operations executed by IPsec, crypto API, TCP/IP network stack, OS and by the application using the IPsec security services. The experimental results reveal also that, in case no crypto accelerator is employed, the throughputs achieved at various packet sizes by IPsec implemented in the Linux user space correspond to about 70% of the throughputs achieved by the kernel space implemented IPsec. The benefits of a kernel space implementation, which does not need a socket or user-kernel context switches, become evident when the security engine is employed. For the packets of size 64–16384 bytes the speedups achieved from crypto offloading are 0.95–4.08 times for the user-space IPsec, and 1.53–10.63 times for the kernel-space IPsec.

The authors of the article [MIK02] measured with the *ttcp* tool throughputs over IPsec on a Gigabit network for the AES and DES3 encryption and HMAC-SHA1 authentication algorithms. Unfortunately, the operation mode of the block ciphers was not defined and only the DES3 cipher was offloaded to the tested crypto accelerator card. The results show that hardware encryption quadruplicates the achieved throughput compared to no offloading, but still, also at 8 KiB packets, the measured maximal data rate was 3–4 times lower than without encryption. Although the AES algorithm was not tested on the crypto card, the results show that in software is the algorithm slightly more than two times so efficient as the DES3 cipher. Furthermore, the IPsec performance at transmitting data files was compared with some other protocols securing data transfer, such as Secure Socket Layer (SSL) and Secure Shell (SSH). In all cases the IPsec proves to be more efficient, especially when used with the cryptographic accelerator.

The measurements in the presented papers have shown that cryptographic operations have a serious impact on the achievable throughput and the use of hardware accelerators can improve the system's performance significantly, especially in case of large packets. However, for small-sized packets, the non-crypto overhead associated with the offloading of a crypto algorithm processing from the general purpose processor, e.g. creating an encryption request for the crypto accelerator, the accelerator configuration according to the request and the interrupt handling, can diminish the crypto accelerator benefits and can even worsen the system's performance.

2.5 Concept

The work presented in the previous section has shown results mainly from the IPsec performance evaluations and the one case when an SRTP protocol was tested did not include cryptographic accelerator measurements. Interestingly, many authors did not really pay attention what crypto algorithms they are testing and that there could be differences in performance between the operation modes in which the block ciphers run. It is assumed that only the default algorithms, as offered by an IPsec implementation, were considered which were all running in the same operation mode and therefore the authors did not mention it in their work.

Unfortunately, not much related work has been found that would assess the advantage of crypto accelerators for securing network data. Only the two presented papers [MIK02] and [TRC08] employ the accelerators in their measurements. The lastly mentioned paper comes closest to the desired performance evaluation. Its authors implemented IPsec in the Linux user space and therefore also the socket mechanism must have been used in order to exchange messages with the TCP/IP network stack. A user space execution together with the socket mechanism adds software overheads which cannot be marginalized, particularly when a network application which needs an access to the crypto hardware is concerned. A user space SRTP implementation using the *Freescale*'s SEC would be exactly this type of application.

An important difference this thesis addresses, opposed to the presented work of the other researchers, is that the Linux kernel in its standard distribution and the hardware, a processor development board from *Freescale*, in its default configuration is used. No optimizations in hardware, nor in software are performed.

An other deviation from the related work is the application employed for the measurements that emphasizes the performance of all elements integrated in the processing path, from the network interface and processor core over to memory subsystems and the cryptographic accelerator. While mostly IPsec is engaged by the vendors of cryptographic hardware and the scientific community, in the scope of this thesis a special application, called *KMedia*, just for this purpose is developed. The application's aim is to secure network data with cryptographic algorithms the simplest possible way. Therefore, also no secret key management is supported and the keys must be inserted manually.

Since *KMedia* is a minimal security application it adds just very little overhead to the crypto processing which must be performed also by the security protocols as IPsec or SRTP if they use the same cryptographic interface, crypto driver and a particular algorithm implementation. In other words, with *KMedia* the emphasis is placed solely on the basic cryptographic services, as they are offered by the operating system and the underlying hardware, and no concerns must be taken with respect to the efficiency of a security protocol implementation.

KMedia is an application executing in the Linux kernel space, therefore also called a "kernel module", which has the simplest possible kernel thread and exchanges messages with the UDP protocol through a socket interface, as the SRTP protocol also does. Therefore, it is assumed, that the test results that will be obtained from *KMedia* performance measurements will present the best possible performance a kernel space implemented SRTP protocol can achieve on the given device.

Naturally, since SRTP is implemented usually in the user space, a user space *KMedia* would be more desired. Unfortunately, as it was already explained in the Introduction section, the Linux cryptographic interface, needed particularly to access the *Talitos* crypto driver, does not have a user space support nowadays and therefore the *KMedia* could be developed only as a kernel module. Note, that the Linux-OCF could be used in order to allow access to the Linux cryptographic interface also from the user space. However, Linux-OCF is not a standard component of a Linux kernel distribution and since it was desired to perform all measurements on the kernel without applying any patches or other additional functionalities, the Linux-OCF was not used.

So, the idea of the *KMedia* module is to get some values that would characterize the best possible cryptographic performance the SRTP protocol implemented in the Linux kernel on the given device with a *Freescale* processor could achieve. Moreover, as explained in the Introduction, two situations will be considered. The first one, when all the cryptographic operations must be performed by the general purpose processor core, and the second one, when the security engine of the processor will off-load the processor core from the cryptographic operations. The measured performance values will be compared and the resulting advantage of the crypto accelerator from that comparison calculated.

The next chapter specifies the *KMedia* module and the further chapters will describe in detail the testing environment together with the employed performance measurement techniques and, finally, also the obtained results will be presented and analyzed.

3 *KMedia* Specification

In the world where important and sensitive data are transmitted through an unreliable open medium such as a wireless network, or through cables of several Internet providers, one cannot be sure if the intended addressee is the only one recipient and whether the data was not modified on its way. Mechanisms are needed to protect the data transmitted through the unreliable networks so that no matter who accesses the content of the data only the receivers defined by the sender can understand it. These mechanisms are provided by a security service called "data confidentiality". It should be also possible to verify that the data is in its original state, unmodified by any possible attacker, as it is provided by the "integrity check" security service. Moreover, there must be a way how to verify that the sender of the data is trusted. This can be performed by the "data origin authentication" security service. The combination of the last two services, the "integrity check" together with the "data origin authentication", is called simply as an "authentication" service.

Kernel Module for Encryption and Decryption Inclusive Authentication (KMedia), described in this specification, provides the presented security services with cryptographic algorithms which were offered by the crypto driver *Talitos* in the Linux kernel 2.6.29. This crypto driver is available in the Linux kernel distributions since version 2.6.27 and enables to execute crypto operations on a cryptographic accelerator integrated in some *Freescale* processors. *KMedia* employs these crypto algorithms to secure network data transmitted by means of the Universal Datagram Protocol (UDP).

Because *KMedia* uses the Linux crypto interface, known as "Scatterlist Cryptographic Application Programming Interface (API)", in order to transform data with the algorithms offered by the *Talitos* driver, in case the driver should be not present in the system, any other crypto driver which implements the needed cryptographic algorithms could be chosen by the interface. *KMedia* supports, except *Talitos*, also the so-called "generic" or "software" driver which controls software implemented cryptographic algorithms in the Linux kernel. For any other cryptographic driver the *KMedia* module was not tested. Appendix G specifies the *KMedia* system requirements and describes the development environment of the kernel module. In Appendix H.1 the source code of the module can be found.

The *KMedia* specification begins with a high-level overview of the module's basic components and their objectives. Afterwards the module's functionalities are described in detail. The format of the messages required for a proper *KMedia* transformation is defined, the usage of the cryptographic operations is explained and the possibilities for an user interaction, the module's configuration and monitoring services, are presented. Thereafter, the module's work flow is explained. The

KMedia specification concludes with a brief description of the strategy followed during module's development and proposes features that could be added to the module in the future in order to enhance its security services and user comfort.

3.1 Module Architecture

The main task of *KMedia* consists from the following three successive steps executed in an infinite loop:

- Receive a message from a UDP socket.
- Transform the message with a crypto algorithm.
- Transmit the message through a UDP socket.

Also, in case an unexpected event during that mentioned steps occurs, e.g. a crypto transformation of the received message fails, *KMedia* reports it by printing a message to the console. And finally, it is possible to change some parameters of the module at load time or even at run time so that the module does not have to be recompiled each time some parameters, as e.g. the port number on which the module should listen for incoming messages, change.

Four programming constructs are needed in order to provide the mentioned functionality to *KMedia*. A *kernel thread* for executing the infinite loop, a *socket API* for receiving and transmitting the messages through a UDP socket, a *KMedia Crypto API* for a cryptographic transformation of the received messages and a *Configuration and Monitoring API* for a user space interoperability with the kernel module. The interfaces are depicted in Figure 3.1 and will be now shortly described.

The *Socket API* enables network functionality of the module and is already implemented in the `net/socket.c` source code file of the Linux kernel distribution. The functions of this API create and configure one UDP socket through which the *KMedia* module exchanges messages with the UDP protocol.

A cryptographic functionality of the *KMedia* module is responsible for network data encryption and decryption. It is based on the *Crypto API for Linux* and the crypto driver *Talitos*. The crypto API is called also as a "Linux Crypto API" or a "Scatterlist Cryptographic API" and it enables an in-place encryption, which means that the produced ciphertext can be written on the same place were the plaintext was. An extra crypto API, the *KMedia Crypto API*, was developed to make the usage of the applied crypto algorithms more transparent than as it is done by the *Linux Crypto API*.

Although the primary purpose of the *KMedia* module is to employ crypto algorithms offered by the *Talitos* driver and with it connected use of the security engine of a *Freescale* processor, the fact that the *Linux Crypto API* is employed allows implicit also to use the software implementation of the cryptographic algorithms in the Linux kernel. From this follows, if there is no *Talitos* driver in the kernel, the software crypto driver, if available, will replace it.

The *Configuration and Monitoring API* allows configuration and monitoring of the module from user space. A user can set module's parameters at module's initialization time through a *kernel*

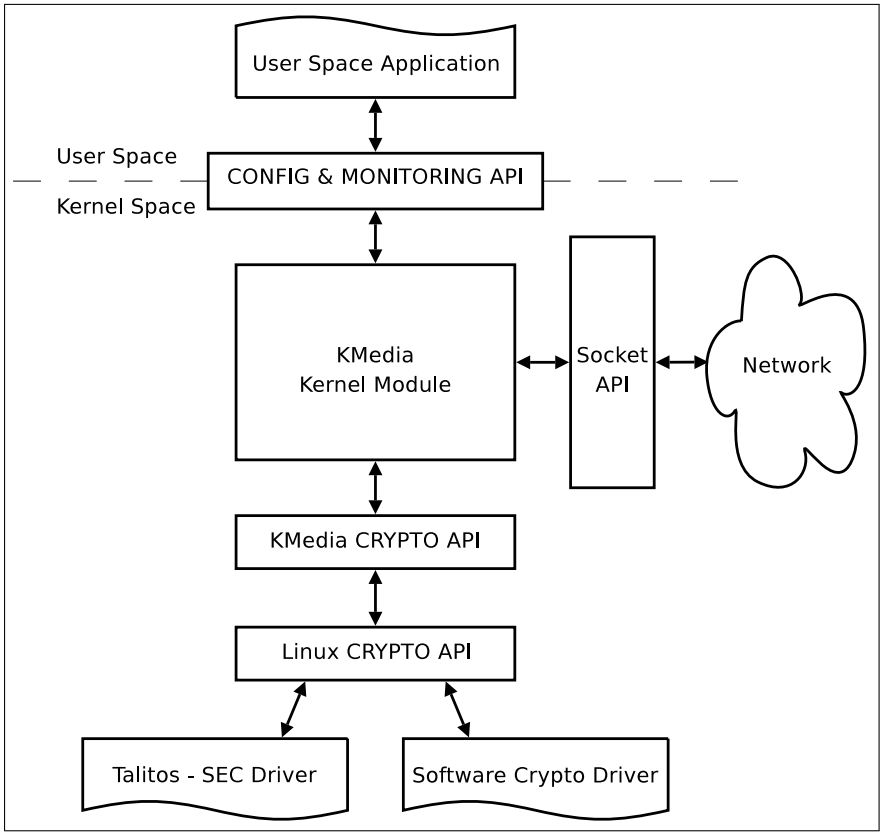

Figure 3.1: *KMedia* APIs (Arrows Show Data Flow)

module interface, or at run time through the *sysfs* interface. The last interface enables also to read out the parameter values. Additionally, the module reports occurred events, such as error states or successful changes of parameters, utilizing the *printk* function.

The following sections describe in detail how the properties provided by these interfaces are implemented.

3.2 Network Functionality

KMedia is designed to provide security services on messages received through a UDP socket. These services can be used to secure a communication channel between a pair of security gateways connecting two sub-networks, between a host and a security gateway, or between two hosts.

From the view of *KMedia* there is no difference if messages are coming from, or going to, applications that run on any host in the network or if the applications lie on the same host as the *KMedia* module. *KMedia* simply listens on a specified port number for incoming messages and sends them after a transformation to destination addresses it is directed to.

Messages going to and from *KMedia*, called "*KMedia* messages", can have one of the two following main states: *plaintext* and *ciphertext*. A *plaintext message* carries unencrypted data which *KMedia* must encrypt and authenticate, and a *ciphertext message* carries encrypted data which is first checked for authenticity and then decrypted. There is also a third state, called *forward*. A

forward message is simply sent to a defined destination address without any transformation and can be used for a testing or debugging purpose.

The different states of a message have one in common, the *KMedia* header, which must be at the beginning of every received message and is never encrypted. It informs the module how the message should be transformed.

In every plaintext message the header is followed by an *Address Field* containing addresses of hosts which the message must pass in order to come to its final destination. According to the header and the *Address Field* the *KMedia* module is able to determine the next destination for a transformed message. The *Address Field* forms together with the *KMedia* header an *Extended Header*. If a message is in a ciphertext state, the *Address Field* is part of the encrypted payload and must be decrypted first before *KMedia* can find the correct next destination.

The following sections describe the *KMedia* header, the *Address Field* and the formats of messages in the plaintext, forward and ciphertext state.

3.2.1 *KMedia* Header Format

Every message that should be transformed by *KMedia* must be prefixed with a *KMedia* header. Its format is illustrated by Table 3.1 and its fields description follows.

⟵ 32 Bits ⟶			
0	4	8	16
Message State	Address ID	Crypto Algorithm ID	Message Length

Table 3.1: *KMedia* Header Format

Message State (4 bits) indicates to *KMedia* how the message should be processed. Three states are possible, "*plaintext*", "*ciphertext*" and "*forward*". An incoming message in *plaintext state* will be encrypted and authenticated, and its state will be set to *ciphertext* and a message in the *ciphertext state* will be decrypted after a successful authenticity check and its state will be set to *plaintext*. The last state, *forward*, causes that the message will be just forwarded to the destination address without any modification. This state could be used for a testing purpose.

In the header is each state represented with an Identification (ID) number. *Plaintext* has the ID 1, the *ciphertext*'s ID is 2 and *forward* is defined by the ID 3.

Address ID (4 bits) informs *KMedia* about the position of the destination address in the *KMedia Address Field* which is described in Section 3.2.2. Values 0, 1, 2, 3 can be used. A message is sent to the destination address after a successful transformation.

Crypto Algorithm ID (1 byte) defines the algorithm for an encryption or decryption transformation. If the *Crypto Algorithm ID* is zero, a default algorithm will be used. Each crypto algorithm gets its unique ID during *KMedia* initialization. This ID can be looked up, and also the default algorithm can be set, through the *sysfs* interface of the module,

which is described in Section 3.4 on the page 37. If a message is transformed by a default algorithm, then the ID of the crypto algorithm which was used as the default one is written into the header.

Message Length (2 bytes) refers to the *KMedia* message size in bytes. The length must be given in network byte order and therefore it is advised to use the function `htons()` to get the right format. This size should ensure that a received message did not change its size on the way from the source to the destination. (E.g., consider the following scenario that would reduce the message size: A message is sent from a source A to a destination C through a host B, which does nothing but forwards the message, i.e. it receives and sends through a UDP socket. If the socket buffer of the host B is smaller than the message size, the message will be truncated. In case the host B will not detect the truncation, it will forward such shortened message.)

With this information in the header, *KMedia* knows how to transform the received message, but the network address to which such transformed message should be sent is still unknown. The *Address Field* provides the necessary information to *KMedia* and is described in the next section.

3.2.2 *KMedia* Address Field

The *KMedia Address Field* contains addresses of hosts which produce, transform and consume a *KMedia* message. This field must be built by the producer of the message, the source host, and is used mainly by the transformation hosts, the crypto servers. The consumer of the message, the destination host, can use this field to send back a reply to the producer. Every address in the field consists of a host Internet Protocol (IP) address and an application port number. The *Address ID* from the *KMedia* header identifies the position of the destination address in the *Address Field*. In other words, the *Address Field* is an array of addresses and the *Address ID* is a subscript into the array.

⟵ 32 Bits ⟶	
0	16
Source Host IP Address	
Source Crypto Server IP Address	
Destination Crypto Server IP Address	
Destination Host IP Address	
Source Host Port	Source Crypto Server Port
Destination Crypto Server Port	Destination Host Port

Table 3.2: *KMedia* Address Field Format

KMedia gets the respective address according to the received *Address ID* and increases the *Address ID* thereafter. The next receiver makes the same, gets the *Address ID*, reads out the respective address from the *Address Field* and increases the *Address ID*. If the received *Address ID* is larger than the highest possible index in the *Address Field*, then the message reached its final destination, the consumer host.

In the next paragraph, the addresses with their corresponding *Address ID* from the *KMedia Address Field* depicted by Table 3.2 are explained. The address names define the hosts a *KMedia* message must pass on its way from the source to the destination in the following order: *Source Host → Source Crypto Server → Destination Crypto Server → Destination Host*. This message flow is illustrated in Figure 3.2.

Address ID = 0: Source Host IP Address and **Source Host Port** identify the producer (creator) of the message. The information, who sent the message, could be important for the recipient of the message, the *Destination Host*. It may be assumed as an address to which a reply should be sent. This IP address and the port number are optional.

Address ID = 1: Source Crypto Server IP Address with **Source Crypto Server Port** determine for the message the first crypto server on the way to its destination. *KMedia* is such a server, but it could be any system compatible with the *KMedia* message processing. The *Source Crypto Server* address could be important if the consumer of the message would like to send back a reply using the same crypto servers. This IP address and the port number are optional.

Address ID = 2: Destination Crypto Server IP Address and **Destination Crypto Server Port** are an important information for the *Source Crypto Server* because they define the address to which the transformed message should be sent. The *Destination Crypto Server* is represented by a host with the *KMedia* module, but, as with the *Source Crypto Server*, it could be any *KMedia* compatible cryptographic system. This address with the port number are mandatory.

Address ID = 3: Destination Host IP Address and **Destination Host Port** is the last network address in the *Address Field*. This IP address with the port number are used by the *Destination Crypto Server* and define the final destination for the message, the consumer host. These fields are mandatory.

Every IP address in the *Address Field* must be in binary format and network byte order. In order to convert a network host address from the numbers-and-dots notation into the appropriate format, the function `inet_aton()` could be used. Also a port number must be in network byte order and it is recommended to get it into the right format by the function `htons()`.

The reason, why the *Address Field* is not an inherent part of the *KMedia* header, is, that the cryptographic transformation process demands additional information to be added in front of the data that should be encrypted and behind the data that should be only authenticated, as it is illustrated by Figure 2.2. Since the header must not be encrypted, it would be needed, at every message transformation, to copy the header with the addresses to make space for the additional cryptographic data. Also, at decrypting it would be needed again to move the header with the addresses in order to restore the original message format as it was before encryption. In order to avoid such copying, it is better to split the header into the *KMedia Address Field* and a smaller *KMedia* header. This way, the *Address Field* can be encrypted with the original *KMedia* payload and a new header can be created directly in front of the additional cryptographic data.

From this follows, if a message is in the *plaintext state*, the *Address Field* is considered as an extension of the *KMedia* header, the *Address Field* together with the *KMedia* header composes an *Extended Header*. In case the message is in the *ciphertext state*, the *Address Field* becomes a part of the encrypted payload, as it is visualized Figure 3.2.

In case a smaller *Address Field* would be sufficient, e.g. just with the mandatory *Destination Crypto Server* and *Destination Host* address, it could be easily adjusted with the macro `ADDR_FIELD_SIZE` in the *KMedia* source code. Nevertheless, in this specification just the field with the four addresses, as described above, is considered.

3.2.3 *KMedia* Message Format in *Plaintext* and *Forward* State

A *KMedia* message that is in *plaintext* or *forward state* consists of the *KMedia* header, the *Address Field* and a payload, as can be seen on the format of the message illustrated by Table 3.3.

The *KMedia* header and the addresses from the *Address Field* were already discussed in the previous sections. The *Payload Data* part can carry any user data and has a variable size. A message in the *plaintext state* is used for the "*Source Host*-to-*Source Crypto Server*" and for "*Destination Crypto Server*-to-*Destination Host*" communication. Since a *forward state* message does not change its format, it is transmitted also between the crypto servers.

⟵ 32 Bits ⟶			
0	4	8	16
Message State	Address ID	Crypto Algorithm ID	Message Length
Source Host IP Address			
Source Crypto Server IP Address			
Destination Crypto Server IP Address			
Destination Host IP Address			
Source Host Port			Source Crypto Server Port
Destination Crypto Server Port			Destination Host Port
∼ Payload Data (Variable Size) ∼			

Table 3.3: Plaintext- and Forward-State *KMedia* Message Format

3.2.4 *KMedia* Message Format in *Ciphertext* State

A message in ciphertext state, also called *ciphertext message*, carries encrypted and authenticated data along with the message authentication code, the *Integrity Check Value*, and cryptographic synchronization data, the *Initialization Vector*. A *ciphertext message* is used for "*KMedia Crypto Server*-to-*KMedia Crypto Server*" communication and its payload is illustrated by Table 3.4.

The *KMedia* header which immediately precedes this payload was already described in Section 3.2.1. The *padding* bytes are appended by *KMedia* in order to align message's length before a cryptographic transformation. The *Initialization Vector* and *Integrity Check Value* fields are filled with an appropriate value during the encryption and successive authentication operation by a crypto algorithm and are not encrypted.

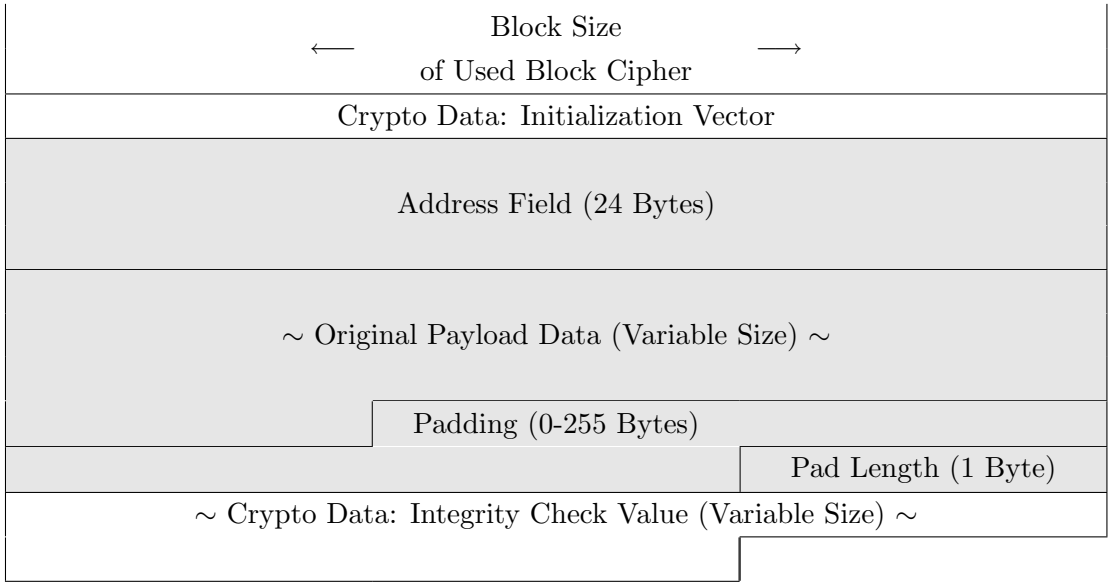

Table 3.4: *KMedia* Ciphertext-State Payload (Fields in Gray Are Encrypted)

A message of the same format is used also intern by *KMedia* for encryption and decryption transformations. Naturally, if a message should be encrypted, all its parts are in plaintext and first after the transformation contains the message encrypted fields like depicted by Table 3.4. This means also, if a message in the *plaintext* state is received for encryption its format must be slightly changed to make place for the *Initialization Vector* between the *KMedia* header and the *Address Field*.

Since the both drivers, software driver and *Talitos*, support "Scatter/Gather" functionality, the *Address Field*, *Original Payload Data* along with the *Padding bytes* and *Integrity Check Value* could be located in diverse memory segments. Nevertheless, *KMedia* obtains the message from the underlying UDP protocol in one contiguous memory block and the same block is then used for transmission. Therefore, all crypto transformation happen in one contiguous memory area.

3.2.4.1 Padding

The format of a *KMedia* message in the *ciphertext state* is affected by the type of cryptographic algorithms which perform the message transformation. *Talitos* offers, and *KMedia* uses, block cipher algorithms operating on fixed-length groups of bits, the blocks. A message larger than a block size is transformed as a sequence of blocks. This implies that the data size to transform must be multiple of the block size of the cipher algorithm chosen for the message transformation.

Pad bytes are used in order to align data to the required length and their number is noted in an one byte long field called *Pad Length* which is always added to the data. The *Padding* and *Pad Length* field conform to the Request For Comments (RFC) [Ken05] specification where it is written: "The padding bytes are initialized with a series of (unsigned, 1-byte) integer values. The first padding byte appended is numbered 1, with subsequent padding bytes making up a monotonically increasing sequence: 1, 2, 3... (This scheme was selected because it offers limited protection against certain forms of "cut and paste attacks in the absence of other integrity measures, if the

receiver checks the padding values upon decryption")". *KMedia* checks the padding after every decryption as required by the RFC.

Also the *Pad Length* is implemented in *KMedia* according to the same RFC [Ken05]: "The Pad Length field indicates the number of pad bytes immediately preceding it. The range of valid values is 0 to 255, where a value of zero indicates that no Padding bytes are present."

3.2.4.2 Initialization Vector

The position of the *Initialization Vector* in the message is affected by the way how *Talitos* handles the input data for transformation. Unfortunately, *Talitos*, as it is in the Linux kernels up to the version 2.6.32, supports only the message format as it is defined and used in the Linux implementation of Internet Protocol Security (IPsec), i.e. the *Initialization Vector* follows the header and immediately precedes the data to be ciphered, as it is illustrated by Figure 2.2.

Also, the *Initialization Vector* is said to be a part of the payload, moreover, "it sometimes is referred to as being part of the ciphertext, although it usually is not encrypted per se" [Ken05]. The RFCs specifying IPsec do not define a separate field for the vector in the payload field. They demand that this is done by the specification of an encryption algorithm that needs the *Initialization Vector* and is used with IPsec, but "typically, the Initialization Vector immediately precedes the ciphertext" [Ken05]. In case of *KMedia* all supported algorithms need the vector and therefore the field for it is explicitly defined in the *KMedia* message format.

3.2.4.3 Encryption/Decryption Workflow

An encryption algorithm takes the *Address Field*, *Original Payload Data*, *Padding* bytes and *Pad Length* as input, encrypts them in-place and writes the generated and used *Initialization Vector* into the appropriate field. The *Integrity Check Value* is computed afterwards over the encrypted data, the *KMedia* header and the *Initialization Vector* and saved into the respective field in the message payload. A packet labeled "IP_packet_B" in Figure 3.2 illustrates which fields are encrypted and which authenticated and Section 3.3 describes in more detail how cryptography in the *KMedia* kernel module works.

A decryption algorithm computes first its own *Integrity Check Value* over the received *KMedia* ciphertext message, except of the *Integrity Check Value* field. Afterwards, the calculated value is compared with the received *Integrity Check Value* and if they equal, the decryption algorithm takes the *Initialization Vector* and decrypts the encrypted fields. It is the role of *KMedia* to strip off the not anymore needed data fields, as the padding bytes, and to transform the message into the *plaintext* format.

3.2.5 Message Size

Exceeding the Maximum Transmission Unit (MTU) that can be passed to the Ethernet protocol on the link layer of the Transmission Control Protocol (TCP)/IP network stack is not an issue for the *KMedia* module, since a UDP socket is used. The underlying Internet layer takes care of the fragmentation and reassembly of the packets that are larger than the maximum packet size of the used network. This way it is assured that a message received though a UDP socket on

the receiver side has the same size as the message on the sender side before it was passed to the socket, provided the buffer supplied to the socket is large enough.

If an incoming message is larger than the reserved buffer, it will be truncated by the socket. This causes that some data will get lost and a crypto server responsible for decryption will not be able to decrypt the message correctly. Therefore, *KMedia* identifies and discards such messages.

The maximal message size, the 4 bytes long *KMedia* header and the 24 bytes long *Address Field* inclusive, a client can send to *KMedia* can be set at the module's load time. Defaultly, this size equals 1368 bytes, whereas the maximum permissible message length that can be specified is 65403 bytes. These values were chosen in order to allow adding extra cryptographic data to the message during encryption and authentication process without exceeding the MTU of an IP datagram on an Ethernet network (1500 bytes) and the maximum IP datagram size (65535 bytes) respectively. The Table 3.5 shows in detail how these sizes were calculated.

Maximal Size of Additional *KMedia* Crypto Data:
32 bytes (Integrity Check Value) + 16 bytes (Initialization Vector) + 16 bytes (Padding Length) = 64 bytes
1368 bytes long *KMedia* Message:
1500 bytes (MTU for Ethernet) - 60 bytes (IP Header) - 8 bytes (UDP Header) - 64 bytes (Additional *KMedia* Crypto Data) = 1368 bytes
65403 bytes long *KMedia* Message:
65535 bytes (Maximum IP Datagram Size) - 60 bytes (IP Header) - 8 bytes (UDP Header) - 64 bytes (Additional *KMedia* Crypto Data) = 65403 bytes

Table 3.5: *KMedia* Message Size Calculation

The computations in Table 3.5 consider the maximal possible sizes of the headers and of the extra crypto fields. The maximal padding length follows from the fact that the data size to encrypt must be multiple of the algorithm's block size, and the maximal block size of an crypto algorithm used in this version of *KMedia* is 16 bytes. In worst-case, the number of bytes to encrypt (the *Address Field* and the *Original Payload Data*) is already a multiple of the block size. Adding the required one byte long field *Pad Length*, causes that up to 15 additional pad bytes must be used to make the data block again aligned. Thus, in the worst-case, the one byte long *Pad Length* and the 15 pad bytes equal to the padding length of 16 bytes.

3.2.6 Use Case Examples

After specifying the format of a message which must be sent to the *KMedia* module in order to use its services and of a message which is transferred between two *KMedia* crypto servers, this section describes some possible scenarios for utilizing the *KMedia* cryptographic services.

Sending a message from a *Host X* to a *Host Y* using the cryptographic server *KMedia* could work as depicted in Figure 3.2. On an example of an IP packet is shown how some header information is used and how it, together with the message, changes on the way from the source to

the destination. Intentionally it is not defined on which host *KMedia* server is placed. It could be any host on the network or the same host as on which the client application runs that produces or consumes the message.

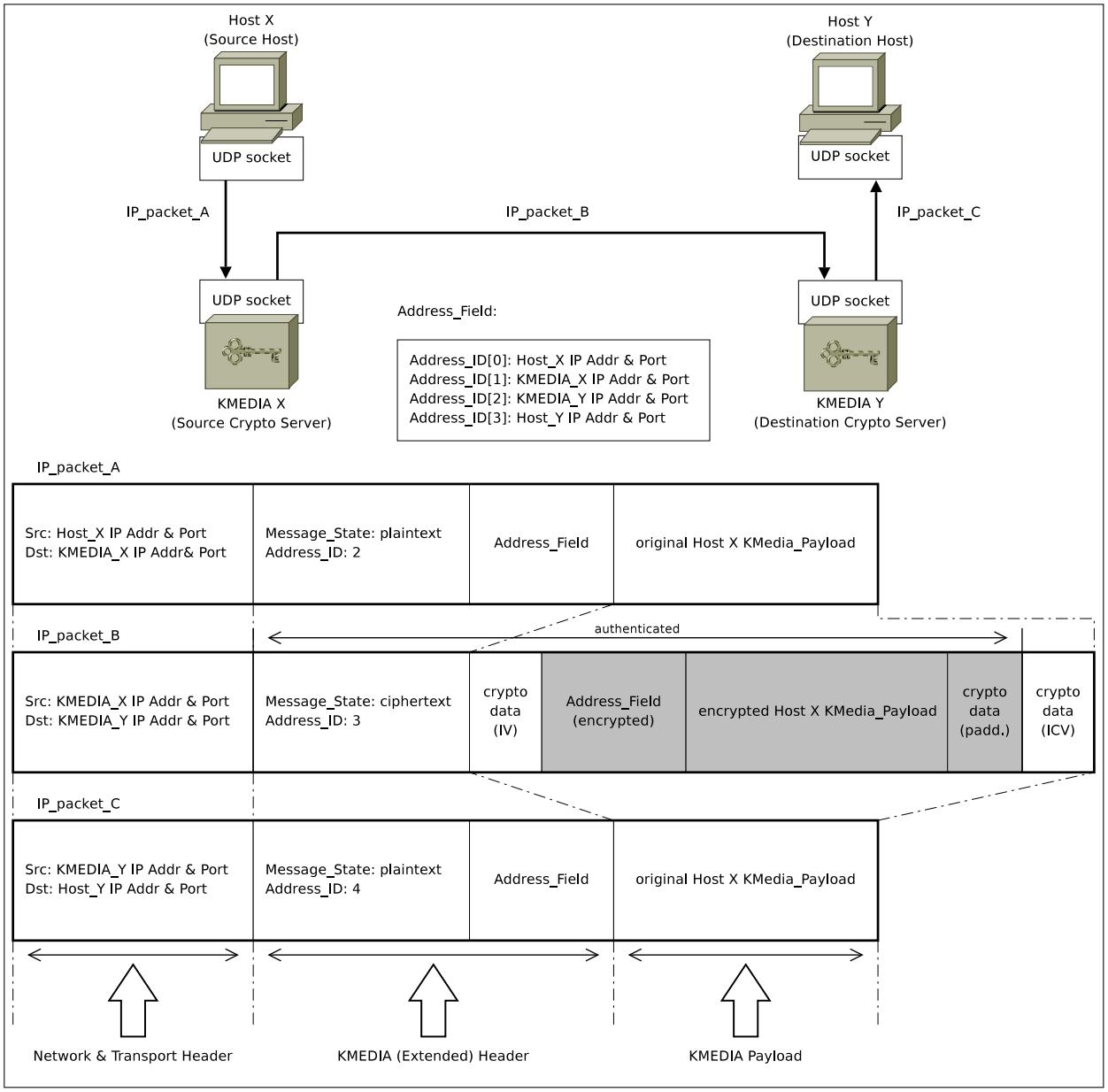

Figure 3.2: *KMedia* Network Topology Example With Illustration of IP Packet Transformations (Fields in Gray Are Encrypted)

First, the *Host X* composes payload data and, before passing the data into to its UDP socket, the host prefixes the data with an *KMedia* Extended Header, which consists of the *KMedia* header and the *Address Field*. Because the *Host X* must fill in the whole *Address Field* it follows that the host must know not only the address of the first crypto server, labeled in the figure as *KMedia X*, but also of the second, *KMedia Y*, and, naturally, also the address of the final destination for the message, the *Host Y*. The *Address ID* is set to the value "two" and the *Message State* to "*plaintext*".

After the message is received by the *KMedia X* server, the server notices the next destination address for the message by looking into the *Address Field* at position two, as defined by the *Address ID*. Thereafter, the *Address ID* is increased to "three" and the message state changed to "*ciphertext*". As a next step, the *Payload Data* together with the *Address Field* is aligned and encrypted. Afterwards, the encrypted data is authenticated together with the new header and an *Initialization Vector*. Such transformed message is transmitted to the next *KMedia* crypto server residing at the address noted earlier.

The *KMedia Y* server decrypts the received message after an successful integrity check, restores the original format of the message as it was before the crypto transformation, changes the message state back to "*plaintext*" and finds the next destination address according to the *Address ID* and the *Address Field*. The *Address ID* is increased to the value "four", which indicates that the next receiver of the message is the last one, the consumer *Host Y*.

After the *Host Y* removes the *KMedia* Extended Header, the data, which the *Host X* wanted to send to the *Host Y*, will remain. If the *Host Y* wishes to reply, it can use the addresses from the *Address Field* in a reverse order.

Another scenario of using the *KMedia* module as a crypto server assumes that some addresses, including the port numbers, in the *Address Field* are equal. For example, the *Source Host* address and the *Destination Crypto Server* address could be the same. This would mean that the *Host X* sends a *plaintext* message to the crypto server and the server sends the transformed message in the *ciphertext* state back to the *Host X*. This *ciphertext* message could be then transmitted directly from the *Host X* to the *Host Y* and the *Host Y* would decrypt it using the crypto server on its side the similar way as the *Host X* did it.

For testing purpose it is also possible that both addresses defining the crypto servers are equal. This means that the crypto server sends the transformed message back to itself: the *Host X* transmits a *plaintext* message to the *KMedia X* which encrypts it, sends it to its own address, decrypts it, and then sends back the original plaintext message to the *Host X*.

KMedia can secure host-to-host communication, where the communication end points are also the cryptographic end points, i.e. if *KMedia* and the application that wants to communicate through a secure channel reside on the same host. Also, *KMedia* can be used as a security gateway and secure communication between two sub-networks. In this case, hosts in one sub-network send *plaintext* messages to the *KMedia* server which transforms them and transmits in *ciphertext* state to its *KMedia* peer acting as a security gateway in another sub-network. Coming there, the messages are decrypted and forwarded to the appropriate local hosts.

As it was already mentioned at the beginning of this section, for *KMedia* there is no difference if messages come from the native host or from another hosts on the network, it just listens for messages on a defined port number and sends to an address defined in the *Address Field* of the message. There are quite many possibilities how *KMedia* could be used and just some of them were mentioned.

3.3 Cryptographic Functionality

Since the primary purpose of the *KMedia* module is to provide confidentiality, integrity and data origin authentication on messages received through a UDP socket by utilizing the crypto device driver *Talitos*, the *KMedia*'s crypto functionality is affected by the way how *Talitos* handles the

input data and what crypto algorithms does it support. The *Talitos* crypto driver was designed to support only the IPsec suite of protocols for securing network connections. Therefore, *KMedia* must use the provided crypto algorithms the same way as the IPsec implementation in the Linux network stack does it.

The crypto algorithms supplied by *Talitos* and used by *KMedia* are listed in Table 2.1 on the page 15. These algorithms are employed in the Encapsulating Security Payload (ESP) mode of the IPsec and belong to the group of symmetric and randomized Authenticated Encryption with Associated Data (AEAD) algorithms built by the "*generic composition*" method. The Section 2.3 introduces this type of algorithms and explains their structure and functionality. In this section, the usage of these algorithms in the context of *KMedia* will be described. At the end also some limitations of the *KMedia Crypto Interface* will be presented.

3.3.1 The Crypto Algorithm Inputs in *KMedia*

As it was already defined in Section 2.3.5, an authenticated encryption algorithm requires following inputs for an encryption: the Algorithm Name, Secret Key, Plaintext, Associated Data, Initialization Vector (IV) and a Nonce for the IV. For decryption, the same Algorithm Name, Secret Key and Associated Data are needed as used for the encryption, but also the IV and Integrity Check Value (ICV) created during the encryption transformation together with the Ciphertext.

The *Plaintext* input in *KMedia* consists from the *Payload Data* of a *KMedia* message in a plaintext state with the *Address Field* and the bytes used for padding. Such plaintext data end up after an encryption and the subsequent keyed-Hash Message Authentication Code (HMAC) authentication with the *KMedia* header, representing the *Associated Data*, as a *Ciphertext with an Appended ICV*, which in turn composes together with a randomly generated IV the payload of a *KMedia* message in a ciphertext state. The "Network Functionality" Section 3.2 describes in detail the format of a *KMedia* message in both states and thus also the structure of the mentioned input data.

In *KMedia* every message can specify in its *KMedia* header what crypto algorithm should be used for its encryption or decryption. Since an *AEAD Algorithm Name* can be up to 64 bytes long[1] and because it is much easier to work with numbers than with names it was decided to assign to every algorithm name a unique, one byte long number, the *Crypto Algorithm ID*. If the ID in the header of an incoming message has a zero value, *KMedia* will use a default algorithm and will write the ID of the used crypto algorithm into the *KMedia* header. The default algorithm can be defined through the *Configuration and Monitoring API* described in Section 3.4.

A *Secret Key* must not be sent in the message and therefore a mechanism is needed to let the both sides of a secure conversation know the key value. *KMedia* does not support any online mechanism for secret key exchange and therefore the only possibility is to define the keys manually through the *Configuration and Monitoring API*.

Every crypto algorithm has its own Secret Key assigned that will be used every time the crypto algorithm is employed. Both sides of a secured channel must have equal keys assigned to the respective crypto algorithm. The keys should be defined immediately after a *KMedia* initialization and should be changed regularly, because with the rising number of messages encrypted with the same key degrades the security strength, as it was described in Section 2.3.5.3.

[1] According to the `CRYPTO_MAX_ALG_NAME` macro used by the algorithm name field in the `struct crypto_alg`. Both constructs are specified in the `include/linux/crypto.h` header file included in a Linux kernel distribution.

The Table 2.2 on the page 18 specifies the sizes of all supported cryptographic functions. However, although for the AES three sizes of secret key could be used [NIS01], only the 128-bit key is supported by default by *KMedia*. It is not possible to change the key length through the *Configuration and Monitoring API*, but, if desired, the size could be changed in the `kmedia.h` header file. Also, the key sizes of the hash functions could be changed in the header file, if desired. *KMedia* accepts any key input of a correct length and format as defined in the *Configuration and Monitoring API* section, but makes no other checks. Therefore, it is up to the user to take care that no keys with a weak nature are inserted.

Finally, a value indicating how many messages were already received by *KMedia* is used as a *Nonce* every time the IV must be generated. The number cannot be used for a replay protection because it is not transmitted together with the message.

3.3.2 *KMedia Crypto API* Constraints

Only the algorithms offered by the *Talitos* driver as listed in Table 2.1 are supported. This is due to the main goal of the crypto interface to provide a transparent access to crypto operations offered by *Talitos*.

In order to transform a message a user formulates a request for a crypto operation, bounds it to a crypto algorithm object and submits it to an appropriate interface function. When the transformation completes, the user is informed through a callback function and can work with the transformed data afterwards. Between submitting the request and being informed on completion of the transformation a certain time can pass. During this time, theoretically, new requests could be defined and bound to the same crypto algorithm and passed to the driver. However, although multiple requests are supported by *Talitos*, *KMedia Crypto API* enables to bind only one request to a crypto algorithm at a definite time. In order to place a new request and bind it to the same crypto algorithm, the user must wait until the previous operation with the algorithm finishes. While this can be seen as a big limitation of the crypto interface, the *KMedia* module has no problem with this approach. *KMedia* was designed to transform only one message at a time, i.e., it continues with the next message first when the previous one was transformed and transmitted, or, in case of a failure, dropped.

The "one request per algorithm at a time" constraint in the *KMedia Crypto API* is a consequence of avoiding the use of memory allocation functions such as the `malloc()` at run time. Each new request requires an extra memory space for its structure and data. The Linux implementation of IPsec allocates new memory space for a request structure every time a datagram should be transformed. In *KMedia Crypto API* each supported crypto algorithm becomes at the initialization time among other things also two request structures with the needed memory space. One is for a crypto transformation when the IV must be generated and the other one if the IV is provided, thus in common situations for encryption and decryption respectively. The requests differ in the number of structure members and the needed memory size. When a transformation must be done, the appropriate request is chosen without the need to allocate a new memory space. In the future there could be a pool with a certain number of request structures so that multiple requests for a transformation with the same crypto algorithm could be issued to the crypto driver at one time point.

This design was chosen in order to achieve better performance and to avoid the possibility of running out of the memory on devices with small-sized memories. The *KMedia* module using

the *KMedia Crypto API* allocates all memory it needs at the initialization time and does not need to call for more memory later at run time. Nonetheless, the truth is, that the crypto drivers accessed by *KMedia* through the crypto interfaces can, and do, allocate and free memory spaces at run time. Therefore, unless the drivers are not rewritten to do all the memory allocation at the initialization time, if possible at all, the use of a `malloc()` and friends cannot be totally avoided.

3.4 Configuration and Monitoring

KMedia can be configured at load time by using module parameters and during execution time through the *sysfs* interface. The *sysfs* offers a convenient way for exchange of information between the kernel and the user space, it allows also to read out some statistical information and module parameters. This section defines which parameters are configurable and which only readable and explains also the different types of messages that *KMedia* prints to the console on certain occasions.

3.4.1 "Kernel Module" Interface

The following module parameters are configurable at load time, first the parameter name is mentioned and then its description:

port_num: The port number on which the UDP socket of the *KMedia* server listens for incoming *KMedia* messages. Any number from 1024 to 65535 is valid.

max_msg_size: The maximal allowed size, in bytes, of a *KMedia* message in a plaintext state which *KMedia* should transform. The size includes the *KMedia* header and *Address Field* length. Larger messages will be dropped. This parameter is used for computing the size of the socket buffer, therefore the larger the value of max_msg_size, the more memory must be allocated for the buffer. The maximal value that can be set is 65403 bytes for reasons already explained in Section 3.2.5.

3.4.2 "sysfs" Interface

The *KMedia sysfs* files which represent *KMedia* parameters and are "readable and writeable" or "only readable" by a user space application are created in the directory with the following absolute name: "`/sys/modules/kmedia/kmedia_crypto`". From now on, all names of files that will be mentioned are relative to this directory.

Every crypto algorithm that could be used with *KMedia* has its own directory filled with files that represent its attributes. The directories are called "`algXXX`" where the "`XXX`" stands for a three digit number representing the algorithm's ID, e.g. `alg001`, `alg101`.

Following parameters can be changed at run time through the *sysfs*:

port_nr: It has the same functionality as the `port_num` from the module parameters.

algs/default_alg_id: The ID number of a crypto algorithm that will be used in case the incoming message did not specify any crypto algorithm for its transformation[2]. The outgoing message, transformed by the default algorithm, will carry this ID in its header.

algs/algXXX/key: A secret key of a supported algorithm. It is a string of hex numbers and must be so long as it is specified in the `algs/algXXX/key_size` file. It contains neither "0x" identifiers nor white spaces.

The next parameters are only readable through the *sysfs* interface:

addr_size: The size in bytes of the *Address Field*.

header_size: The size in bytes of the *KMedia* header.

max_msg_size: The maximal allowed size, in bytes, of a *KMedia* message in a plaintext state (including the size of the *KMedia* Extended Header).

statistics/msgs_rx: The number of received messages.

statistics/msgs_tx: The number of transmitted messages.

algs/algs_complete: A list of all cryptographic algorithms that can be used with *KMedia*, including additional details for each. The output is the same as iteratively reading all files from each algorithm's directory. This way one gets all information about all crypto algorithms at once.

algs/algXXX/ID: A unique ID number of the algorithm residing in the directory. This ID must be used in the *Crypto Algorithm ID* field of the *KMedia* header.

algs/algXXX/IVsize: The size in bytes of the algorithm's *Initialization Vector*.

algs/algXXX/ICVsize: The size in bytes of the algorithm's *Integrity Check Value*.

algs/algXXX/driver: The name of the algorithm's driver. If the name includes the string "talitos" then the *Talitos* driver is used for the message transformation. If it includes the string "generic", then the message transformation is done by the software driver.

algs/algXXX/enc_key_size: The size in bytes of the encryption key.

algs/algXXX/key_size: The size in bytes of the secret key. It is a sum of an encryption key size and an authentication key size.

algs/algXXX/name: The full name of the crypto algorithm whose attributes are listed in the directory.

In case an algorithm is not supported, because e.g. there is no appropriate driver for it, its directory and files will be created anyway, but, except of the name and ID, they will all return a zero value on reading, the driver name will be "null".

The following additional *sysfs* files are visible and readable only if the *KMedia* kernel module is compiled with the macro `TIMESTAMPING` in the `kmedia.h` header file. The macro enables a

[2] *Crypto Algorithm ID* in the *KMedia* header of the message equals zero.

time-measuring functionality of the module, i.e. to measure how much time a received message spends in the module. Since it adds an unnecessary overhead which should be avoided in a normal operation, this functionality should be used only for a testing purpose.

statistics/time_sock2sock_clk_tck: The number of the Central Processing Unit (CPU) clock ticks *KMedia* spent transforming the received messages. For each message it is measured how much ticks it takes from the point when a message was successfully received to the point a message is prepared to be sent, thus the number of ticks it takes from the receive socket to the send socket while the time spent in the sockets waiting for a message and sending it is not included. The parameter `time_sock2sock_clk_tck` shows the sum of the ticks measured for all messages that passed the *KMedia* module since the module initialization.

statistics/time_crypto_clk_tck: The sum of the CPU clock ticks the *KMedia Crypto API* spent transforming the received messages. For each message it is measured how long the pure crypto transformation lasts, thus the number of ticks it takes from the point a request for a crypto transformation to the *KMedia Crypto API* is passed to the point the crypto interface returns the transformed message. The parameter `time_crypto_clk_tck` shows the sum of the ticks measured for all messages that passed *KMedia* since the module insertion time.

statistics/time_num_msgs: The number of messages integrated in the time-stamping measurement. The above presented `time_sock2sock_clk_tck` and `time_crypto_clk_tck` show the sum of the ticks measured for all messages transformed by *KMedia*. In order to get an average value for one message, the values from the two parameters must be divided by the value from the `time_num_msgs`.

statistics/time_clk_tck_per_sec: The number of CPU clock ticks per second. It is measured at *KMedia* initialization and needed in order to convert the values in CPU clock ticks into seconds. If possible, a more precise value termed as "timebase" from the file `/proc/cpuinfo` should be used.

3.4.3 "printk" Interface

Additionally, *KMedia* prints messages to the console by calling the kernel function *printk* in order to report an occurred event. Three types of messages are specified: *KMedia Warning*, *KMedia Info* and *KMedia Drop Message*.

KMedia Warning message is printed in case *KMedia* fails to allocate and initialize data structures during the module initialization or at run-time. The message identifies the occurred problem and is printed with the priority (loglevel) `KERN_WARNING`. Thereafter, the module frees all its already allocated resources and is removed from the kernel.

The *KMedia Info* messages are printed in situations that do not create serious problems to *KMedia* but are worthy to note. Mostly these situations occur when a user wants to change some module's parameters, let it be at module initialization through module parameters or at run time through the *sysfs* interface, but the input has an incorrect value, type or format. The messages describe the occurred problem and are printed with the priority `KERN_INFO`. *KMedia* uses this type of messages also just for an information purpose, e.g. to inform that a value was successfully assigned to the module's parameter.

The last type of a *printk* message, the *KMedia Drop Message*, is used in cases when a message received through a UDP socket will be not sent to its next destination and will be simply discarded. The printed message with `KERN_INFO` priority explains the reason for dropping the message, e.g. the crypto algorithm ID as defined in the message header is incorrect, or the crypto algorithm is not yet supported (no appropriate driver was found).

3.5 *KMedia* Execution

Immediately after the *KMedia* module is inserted into the kernel, it initializes its resources. This includes loading of all supported cryptographic algorithms, allocating the needed memory space for a message reception and transformation, setting up the kernel socket to listen on the right port for incoming UDP packets, creating and starting the kernel thread with an infinite loop that handles the message reception, transformation and transmission, and, finally, building the *sysfs* files for the *KMedia* module attributes that should be visible and some of them also writeable from the Linux user space. In case no port number and/or maximal message size will be set at module's load time through the module parameters, predefined default values will be used. After a successful allocation of all resources, the module prints a message on the console to inform what values the port number and the maximal message size have. Every secret key assigned to a crypto algorithm at the module's load time has a default value that should be changed as soon as possible through the *Configuration and Monitoring API*.

The normal flow of network and crypto operations in the infinite loop which transforms only one message at a time is as follows:

1. *KMedia* waits for an incoming UDP message from the kernel socket and checks periodically whether the port number should be changed or the thread should terminate (in case the module is unloaded).

2. After a successful reception of a message some sanity checks on the message are carried out. It is controlled whether the message has a correct size and whether the fields in the *KMedia* header have correct values.

3. Afterwards, *KMedia* attempts to get the crypto algorithm according to the Crypto Algorithm ID from the message header. If the ID is zero a default algorithm is prepared.

4. If the cipher algorithm was successfully loaded, *KMedia* continues according to the message state.

   (a) In case the message is in *plaintext state*:

      i. It is examined whether there is enough space available in the allocated buffer for a message transformation. The message grows in size during the AEAD encryption process and it could happen that the allocated buffer is not sufficiently large.

      ii. After the message is aligned with padding bytes, the destination address for the transformed message is read out from the *Address Field* and the address ID is incremented. A new header is constructed exactly IV-length bytes in front of the *Address Field* to make space for the IV that must follow the header.

iii. Afterwards, a *KMedia* request for an AEAD encryption operation is built. It must be defined where is the free space for the IV, where are the data for encryption and authentication and where the data only for authentication.

iv. The request is bound to the crypto algorithm prepared in the step 3 and submitted for processing by calling an appropriate *KMedia Crypto API* function[3]. *KMedia* waits afterwards for completion of the transformation and continues with step 5.

(b) Provided that a message in *ciphertext state* was received the following steps are performed:

i. First, a *KMedia* request for an AEAD decryption transformation is built. It is similar to the encryption request. It must be defined where the associated data, IV and ciphertext are placed.

ii. Also, as with the encryption, the request is bound to the, in step 3 prepared, crypto algorithm and submitted for processing by calling an appropriate crypto interface function[4]. *KMedia* waits until the transformation finishes.

iii. After a successful decryption, the padding bytes are checked and stripped off. Subsequently, a new *KMedia* header is built exactly in front of the *Address Field* from which the destination address for the decrypted message is read out. Thereafter the address ID number is incremented and *KMedia* continues with step 5.

(c) If the message state is *forward*, just the next destination address is read out from the *Address Field* and the address ID is incremented.

5. After a successful completion of the transformation, the message is transmitted through the UDP kernel socket to its destination address that was read out from the *Address Field* in one of the previous steps.

6. The infinite loop in the kernel thread continues again with the step 1. *KMedia* is prepared to receive and transform next message.

Every time it is detected that the message intended for a transformation is of a wrong format or size, or has incorrect values in its header[5] or *Address Field*, it will be dropped and the message transformation loop will continue with waiting for a next incoming message. The same happens when an error occurs during the message reception, crypto transformation or transmission. In any case, the reason for dropping the message will be printed on the console.

During the whole transformation process the employed crypto algorithm is locked. This is done in order to prevent changing of a secret key of a crypto algorithm that is currently in use.

3.6 Development Tactic

KMedia module was written in three main stages which are shortly described in this section.

In the first stage the *KMedia Crypto Interface* was developed to provide cryptographic functions to the kernel module. At the end of this stage the *KMedia* module was able to encrypt and decrypt data using the *Talitos* and software crypto driver. This way the data could be transformed by

[3] `kmedia_crypto_ givencrypt_authenc()`

[4] `kmedia_crypto_decrypt_authenc()`

[5] E.g., there is no appropriate driver for the crypto algorithm ID which is defined in the message header.

cryptographic algorithms implemented either in hardware, in the security engine of a *Freescale* processor, or in software, in the Linux kernel distribution.

Afterwards, the network functionality was developed. This second phase included the kernel thread and the UDP kernel socket implementation. At the end, the module was able to receive and transmit UDP messages from any and to any Internet address.

In the third phase the two features from the previous two stages were merged. As a result, the messages passing the module could be accordingly cryptographically transformed.

The *Configuration and Monitoring API* evolved during the whole development of the *KMedia* module. Especially the reporting of failed function calls started together with the first stage. Thereafter, with the socket implementation, also the module parameters were deployed and, at the end of the module writing, *sysfs* directories and files for *KMedia* parameters were created.

In order to achieve the best possible performance of *KMedia*, all operations that need to allocate memory space are done at the module initialization time. This means that no `malloc()` and related functions are used by the *KMedia* kernel module or the *KMedia Crypto API* at run-time.

3.7 Coding Style

The Linux kernel coding style was used for writing the code. Functions that need to be visible from outside of the *KMedia Crypto Interface* have names with the "`kmedia_crypto_`" prefix.

3.8 Possible Extensions

KMedia does not support an additional loading of drivers for crypto algorithms, only the drivers available at the module initialization time are used. Therefore, every time one wants to add or remove a crypto driver, she must unload first the *KMedia* module from the kernel, make the wanted changes, e.g. load/unload a crypto driver into/from the kernel, and then initialize the *KMedia* module again by inserting it back into the kernel. The possibility of adding or changing a crypto driver at module's run-time would enhance the comfort with using the module.

Also, as it was already mentioned in this specification, only one request for a message transformation can be submitted at a time to the *KMedia Crypto API*. The next request can be issued first when the previous one was processed. Creating a pool of requests structures that could be passed concurrently to the API could help to advance the performance of *KMedia*, particularly when dealing with bursts of messages.

The third point suitable for improvement was also already mentioned. The secret cryptographic keys used by the crypto algorithms can have various lengths, but in *KMedia* these lengths are hard-coded in the header file `kmedia.h`. Therefore, recompilation is needed every time a key size is changed. One solution could be to enhance the *Configuration and Monitoring Interface* with the possibility of changing the size of a secret key at run time.

And finally, although confidentiality, integrity and data origin authentication are provided, a capture-replay network attack is still possible. An attacker could record a network transaction and replay it later. It is recommended that a further development of *KMedia* employs also a security service against this type of attack.

4 *KMedia* Development: Problems and Solutions

The problems encountered during the development and implementation of the *KMedia* kernel module are described in this chapter. If multiple possible solutions to an occurred problem arose, they are here discussed and the reason for choosing one of them for the implementation is given. It is also mentioned which sources of information were used in order to get a deeper knowledge of the given problematic about the crypto algorithms, kernel sockets, kernel threads and the *sysfs* interface. These sources helped also with the implementation of the programming constructs. First, the cryptographic, then the networking and finally the monitoring functionality together with the kernel thread implementation is discussed. All functions mentioned in this chapter relate to the Linux kernel version 2.6.29 and the file names to its source code root directory.

4.1 Encryption

At the beginning with the deployment of the crypto algorithms it was important to understand what crypto algorithms are offered by the *Talitos* driver and how *KMedia* could make use of them. Since there was no appropriate reference manual addressing *Talitos*, its source code must have been inspected, namely the files `drivers/crypto/talitos.c` and `drivers/crypto/talitos.h`. This way it was determined that the only supported algorithms are the symmetric, random, generic-composition Authenticated Encryption with Associated Data (AEAD) algorithms using the encrypt-then-MAC method, described in Section 2.3. The internet-draft [McG09] helped a lot to get the theoretical knowledge necessary for working with these algorithms, particularly with the HMAC-SHA1-CBC-AES. The article [BN08] describing mainly the security aspects of such algorithms but dealing partly also with the previous and recent work on this area[1] was also very helpful together with the RFCs [KBC97, PA98, MD98] describing the HMAC and CBC operation modes and the CBC-DES algorithm respectively.

The more challenging part was to find out how these AEAD algorithms can be accessed. In the *Talitos* source are some programming constructs' names prefixed with "`ipsec_`" and in the comments is mentioned that the code of the *Linux Crypto API* was employed. Also, when looking into the version control system dedicated to the development of the kernel crypto functionality [20] the first commit of the *Talitos* driver mentions that it is for use with Internet Protocol Security

[1] In Section 1.3 and 1.4 in [BN08].

(IPsec). Up to the recent kernel 2.6.33 this primary purpose was not changed. Therefore, the next step was to get to know IPsec and to find out how its Linux implementation employs the *Talitos* crypto functionality with the help of the *Linux Crypto API*.

A brief overview of IPsec was already given in Section 2.2. From the IPsec suite of security protocols particularly important for this thesis is the Encapsulating Security Payload (ESP), defined in the RFC [Ken05], because it is responsible for data packet encryption and authentication using the AEAD crypto algorithms. Therefore, in order to see how it employs the cryptographic algorithms, it was necessary to find out how this protocol is implemented in the Linux kernel.

Unfortunately, not much documentation was found on Linux implementation of IPsec. A good starting point was the website [19] and the book [Ben05, page 882] but then the quite poorly commented source code must have been inspected. Searching in the source tree for file names or functions with an "esp" string using the Linux cross reference site [23] returned quite many results but also the file name `net/ipv4/esp4.c` which, as it was determined, implements the wanted ESP protocol. The `esp4.c` file was a great help because it defines everything needed to use the AEAD algorithms, from initialization routines to issuing requests for encryption or decryption to Linux Crypto Application Programming Interface (API) and getting the response from the interface when the transformation finishes.

To get some practical experience, the crypto testing module `crypto/tcryp.c` was extended with a function to examine the generic composition AEAD algorithms. An available function "`test_cipher_speed()`" from the file was taken as a template and modified according to the operations in the `esp4.c`. This way the practical experience with the *Linux Crypto API* and the crypto algorithms was gathered which was then directly used in the design and implementation of the *KMedia Crypto API*. The only big deviation of the interface from the `esp4.c` code is, that the memory space for a transformation request structure is not allocated for every single message that must be transformed but only once at the crypto algorithm initialization time, Section 3.3.2 handles this topic in more detail. Note that `tcrypt.c`, as distributed with the kernel source tree, is able to test some AEAD algorithms with the help of the code from the `crypto/testmgr.c` file. Unfortunately, only the combined mode algorithms are supported which execute in the Galois Message Authentication Code (GCM) and Counter with CBC-MAC (CCM) operation mode and do not need that much initialization work as the generic composition algorithms. Additionally, `esp4.c` does also some data alignment operations not performed by the testing module.

Since the implemented *KMedia Crypto API* works with the *Linux Crypto API* it does not take care whether the transformation is performed by the software or hardware-*Talitos* driver. Usually only the algorithm name is specified and the Linux crypto interface chooses an appropriate driver. Since the *Talitos*' algorithms have a higher priority they are preferred over the software implemented algorithms. To perform a transformation with the software driver, either the *Talitos* driver must be removed from the kernel, or the algorithm's driver name must be explicitly specified.

One issue with the implementation of the cryptographic functionality for the *KMedia* module was the placement of the synchronization data, the Initialization Vector (IV), in the message. It was believed that it does not have to conform with the IPsec specification, since the Linux crypto interface does not define where the vector should be and therefore applications, which do not implement IPsec, should be allowed to place it where they want to, for example, at the end of the encrypted and authenticated message. This is true so far the software driver is utilized, but this does not work with *Talitos*. The crypto driver was designed to support *IPsec* and therefore when a kernel module wants to make use of the *Talitos*' crypto resources it must use them the way

the *IPsec* does it. Interestingly, at encryption writes *Talitos* the generated IV into the defined memory space and encrypts a message correctly. Also at decryption the IV is taken from any place in the memory. But if the IV placement does not conform to IPsec it is not taken for a message signature computation and therefore the computed Integrity Check Value (ICV) is wrong. Software implemented algorithms give no restriction on the IV location.

In the IPsec's ESP protocol specification [Ken05, Section 2.3] the IV position is not explicitly defined but it must be carried in the payload field, typically immediately preceding the ciphertext and thus following the header which should be authenticated. This was confirmed also in a personal e-mail with the *Talitos* programmer [21]. A plaintext message received by *KMedia* has the header immediately in front of the data that should be encrypted. When the IV must be between the header and the ciphertext, then the header must be moved to make place for the IV. Since memory moves degrade performance this is not an ideal solution. Another possible way would be to leave a free space for the IV already in a plaintext state message, but this requires that a host which wants to use the *KMedia* security services knows how long the IV is. It would also induce wasting with network resources since the IV field would carry between the host and the *KMedia* device no important data. Therefore, the Linux implementation of IPsec and of the network stack was inspected to get to know how the kernel deals with this requirement.

Mainly the points in the network stack were analyzed in which the packets are prepared for, or adjusted after, a transformation performed by a crypto algorithm utilized by the ESP protocol in the *transport* mode. In both cases, when a packet is to be transmitted and also when it is received, the Internet Protocol (IP) header is moved. Either to make space for the ESP header and the successive IV, or in order to close the gap which arose after removing the header with the vector. A more detailed analysis can be found in Appendix G.3.

After looking at the few aspects of IPsec in the Linux kernel it can be said that the implementation does not disturb the main composition of the kernel's network stack. The security policies which must be applied to an outgoing packet are just an extension to the routing mechanism and if an ingress packet is to be processed by a host's transport layer (TCP or UDP) the routine responsible for calling a transport layer protocol handler, the `ip_local_deliver_finish()`, invokes first the appropriate IPsec protocol handler and then the TCP or UDP handler. It is believed that this strategy lowers the code complexity but it has its cost. The moving of the IP headers in the transport mode at every received or transmitted packet can not contribute to a better system performance.

Notable is the case when a packet is transmitted in the ESP transport mode. As it is explained in Appendix G.3, first the routing and IPsec policies are determined and afterwards the UDP and IP header built. This means, that although it is known before the IP header creation if an ESP header would have to be built is the IP header placed just in front of the UDP. Afterwards, few steps later, the IP header must be moved forward to make space for the ESP header and the IV which must lie between the IP and UDP header.

Since in *KMedia* it was wanted to avoid memory moves so far as possible it was decided to split the *KMedia* header into two parts: to a smaller one *KMedia* header and a *KMedia Address Field*, both objects are described in Section 3.2. This way the addresses are encrypted together with the *KMedia* payload and the header which carries some control data which are changed in any case during each *KMedia* transformation is defined already at an appropriate place, IV length bytes ahead of the Address Field when the message is going to be encrypted and just right in front of the Address Field after a message decryption.

4.2 Networking

The biggest problem with the networking functionality was its design, the subsequent implementation was rather straightforward. Following questions must have been answered: How should the *KMedia* module know to which address a message after a crypto transformation must be sent? How will the module know what to do with the message, should the message be encrypted or decrypted? Must an application requiring cryptographic services send its message explicitly to *KMedia*, or should *KMedia* act as a security gateway in which case the encryption would be transparent for the sending application? In the next sentences it will be spoken also about "frames" on the link layer and "packets" on the internet layer but always in context with the UDP header and the UDP payload they contain in their body.

The first intention was to use the both available network interfaces of the *Freescale*'s processor board. One would be connected to a trusted local network and the second one with the unreliable and untrustworthy Internet. This way it would be clear that messages coming from the local network must be encrypted and then transmitted to the Internet, and vice versa, messages coming from the Internet would be decrypted and forwarded into the local network. However, the information to which address the transformed messages should be sent was still missing. The destination address could be either hardcoded directly into the module or it could be provided through the user space interface as a part of the module's configuration. While this would be a simple solution it was desired to make the module's forwarding capability more flexible, independent from its configuration. The information which way the message wants to go should be read out from the message and not prescribed by the module.

Therefore a solution, inspired by the routing mechanism, was thought up. If clients communicate with servers that are not in their local network they have to send the packets to the routers which forward the packets afterwards to the next routers on the path toward the wanted server. The packets sent to the routers carry an IP address of the destination server in the IP header according to which the routers decide on the path of the packet. If clients would transmit packets to *KMedia* instead of to the router and *KMedia* would be able to handle such packets, then *KMedia* would know the destination to which the packets after encryption should be sent. According to the destination IP in the client's packet IP header *KMedia* in the client's local network would choose *KMedia* in the server's local network which would be responsible for decryption. The *KMedia* on the server side would receive the client's encrypted packet from the *KMedia* on the client's side and transmit it to the server after decryption. This way the server would get the original client's IP packet. The Figure G.1 on the page 141 illustrates this scenario on an example which is described in more detail in Appendix G.4.

While this approach is transparent for the client and the server it is quite complex for an implementation. If *KMedia* should act as an intermediary in front of the router, thus all traffic from hosts on the local network would flow first to *KMedia* and then to the router, *KMedia* would have to deal with every sort of packets transmitted by the hosts. Therefore, every packet would have to be classified somehow whether it is the packet that should be encrypted. Also, since encrypted packets should be sent only between two *KMedia* crypto servers, a *KMedia* residing in the source host subnetwork would have to know according to the packet's destination IP which *KMedia* lies in the destination's host subnetwork, so something like a list of "which *KMedia* operates in a subnetwork of which hosts" would have to be created. Since the classification and the list seemed to add much overhead to the *KMedia* processing it was decided to work on a simpler solution.

If it should be possible to send a transformed packet to a destination defined by the packet, thus not to set the address statically in the *KMedia* module, and the method described above presented a rather complex solution, the remaining possibility was to send the destination address as a part of the UDP payload. Therefore, a message was split into a *KMedia* header and a *KMedia* payload. The header contains, except the destination address, all other relevant information for the payload processing, e.g. whether it should be decrypted or encrypted. The payload carries either the client's data in a plaintext or, after an encryption, in a ciphertext state together with all needed additional cryptographic data. To speedup the crypto transformation process the header was later split into a smaller header and an "Address Field" as it was already described at the end of the previous Section 4.1. Since this approach was implemented all information about the header, Address Field and payload together with an use case example can be found in the *KMedia* specification, particularly in Section 3.2.

This lastly presented solution was easy to implement and enables to *KMedia* to find quickly the right destination address for each transformed message. Since the Address Field is implemented as an array, finding the right destination is only a question of reading out the address' index from the header. Also, *KMedia* can receive all messages through one port of one network interface since the header depicts the state of the received message, e.g. whether it is encrypted and should be decrypted or vice-versa. The negative side, except of larger messages that must be transmitted, is, that quite strong client's cooperation is demanded. A client must fill in the whole Address Field and also the header before sending it to *KMedia* for encryption. Also, when the message is received by the final host, this additional data must be stripped off before the *KMedia* payload can be handled.

A valuable source of information about all sort of sockets were the websites [5, 8] and also the kernel source code. The website [28] mentions briefly why it is needed to distinguish between user sockets and kernel sockets.

4.3 Threading and Monitoring

No remarkable issues were encountered during the implementation of the kernel thread and the user space interface, both defined in the *KMedia* specification. It was just a matter of finding the right kernel function to create and start up the *KMedia* kernel thread. The basic info was obtained from the book [Lov05, The Linux Implementation of Threads] and afterwards the kernel code which defines the threads, as e.g. in the file `kernel/kthread.c`, helped a lot. Moreover, the code was found to be one of the best commented Linux kernel source codes encountered during the *KMedia* implementation.

To create some files for the *sysfs* interface, which is used for the exchange of information between the user- and kernel-space, first the parameters were chosen that should be readable, or also writeable. Thereafter, a *sysfs* file tree was designed from these parameters and finally implemented. All about the *sysfs* is well documented in the book [CRKH05, Chapter 14: The Linux Device Model].

5 Testing Tools, Environment and Methods

The following chapter discusses techniques applied in performance measurements of a *Freescale*'s processor board with a network-oriented System on a Chip (SoC). The role of the *KMedia* kernel module was to enable testing of the device with the primary focus on cryptographic operations.

Because *KMedia* can use two different drivers in order to provide security services on network data received trough a Universal Datagram Protocol (UDP) socket it was particularly important to get performance results for each of the two drivers and to compare them. Nonetheless, the tests do not provide performance assessment of a particular driver they just show how the two drivers influence the performance of the system as a whole.

While one driver offers cryptographic operations through a crypto accelerator integrated in a *Freescale*'s processor the second uses software implemented crypto algorithms available in the Linux kernel. In the first case a special hardware is responsible for the crypto execution and in the second a Central Processing Unit (CPU) itself must take care of the algorithms. The advantage of the second type is that no special hardware is needed. However, since cryptography is very computationally intensive, the possibility of offloading the crypto operations from the CPU could be proved to be very useful.

First, the tools needed for *KMedia* verification and performance measurements will be introduced. Afterwards, the test equipment along with the Device Under Test (DUT) will be described and the message sizes and formats employed in the measurements will be discussed. The final section describes what performance characteristics were measured and how, in what tests' configurations.

5.1 Test Tools

The benchmarking tools described in this section were developed for, or adapted to, the performance measurements and verification of a network device with a cryptographic functionality. Two of the presented tools, *Netperf* and *Socat*, are freely available applications which support a huge list of options and parameters. All other tools are small utilities, not larger than one file, developed in the scope of this thesis. Their source codes are listed in Appendix H.

This section gives just a brief overview of these tools. A more detailed description of each of the tool with command line examples how to compile, configure and use them is given in Appendix A.

5.1.1 Netperf

Netperf is a network performance benchmark with the primary focus on "unidirectional data transfer and request/response performance using either Transmission Control Protocol (TCP) or UDP and the Berkeley Sockets interface" [Jon07]. It works on a client-server model and consists of two executables, `Netperf` and `Netserver`. While the `Netserver` runs as a standalone daemon in the background, the `Netperf` configures and initiates each test.

Basically, in *KMedia* test environment, *Netperf* sends messages of a defined size to *KMedia* at a specified rate, e.g. one message every one millisecond. *KMedia* cryptographically transforms the received messages and transmits them to the *Netserver*, which just counts the received messages. After the test finishes, the *Netperf* gets the number of received messages from the *Netserver* and computes the reached data rate, which is then printed on the terminal. The Figure 5.1 illustrates this.

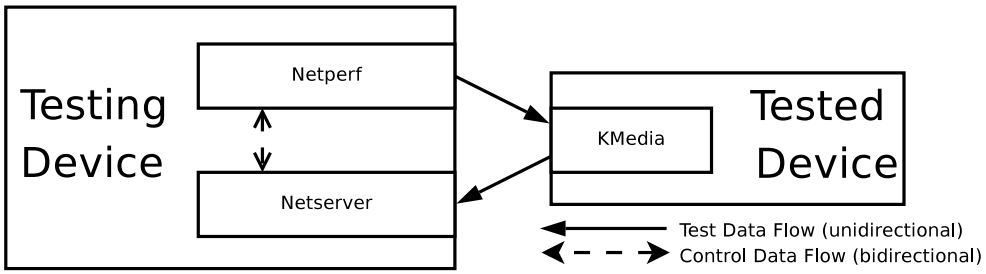

Figure 5.1: Netperf Test Set Up

5.1.2 Socat

In words of the author of the tool, *Socat* is a "*Netcat++*" [14], an enhanced version of the tool which describes itself as a "TCP/IP swiss army knife". Both tools can establish network connections of different types and configurations and transfer data through them, but *Socat* has, compared to the *Netcat*, an extended design with some new implementations. It can be used for many different purposes as it offers a large set of different types of data sinks and sources and also because many address options may be applied to the established data streams [Rie10].

Socat was engaged in receiving and saving UDP messages of sizes over 60000 bytes. Contrary to the *Socat*, uses *Netcat* maximal 8 KiB [18] large reads and writes, what was insufficient for the *KMedia* test purposes.

5.1.3 KMedia Message Checker

The purpose of this tool is, as can be derived from its name, to check the content of a message on correctness. At the beginning, the *KMedia Message Checker* reads in a file which contains the exact copy of an expected UDP message that should be checked. Afterwards, everytime the *KMedia Message Checker* receives a UDP message, it compares it with the data from the read-in file. In case the message does not match the content of the file, it is dropped. If the message is of a correct size and content it is forwarded either to a default network address defined in the source code as a macro, or to an address defined at the command line.

5.1.4 KMedia Header Engine

A *KMedia* header together with an *Address Field*, as described in Section 3.2, is an inherent prefix of every *KMedia* message that should be transmitted to the *KMedia* kernel module. Without the information in this *Extended Header*, how *KMedia* header together with *Address Field* is called, *KMedia* module cannot transform the received UDP message and will drop it. Therefore, all applications which want to use the security services offered by the *KMedia* module must prefix their UDP plaintext messages with the *Extended Header*. In order to prevent adding code to, or re-writing, applications that already use the UDP messages for a data transfer the applications can use the *KMedia Header Engine* tool.

At initialization time, *KMedia Header Engine* creates the *Extended Header* according to the data from a read in configuration file. Afterwards, the tool waits for incoming messages, either from an application or from the *KMedia* module. If a message comes from an application, the tool prepends the *Extended Header* to it and forwards the message to *KMedia*. If a message from *KMedia* is received, the *Extended Header* is stripped off and the message, without the header, forwarded to the application. This way the *KMedia Header Engine* enables a transparent use of the *KMedia* module, since an application does not have to deal with the extra header required by *KMedia*.

5.1.5 KMedia Statistics Reader

This utility's task is to collect and evaluate some statistical information about the *KMedia* module. Namely, the average times of *KMedia* message transformations and the message drop rates.

The tool reads in relevant data offered by the *KMedia* module through its "Configuration and Monitoring Interface", described in Section 3.4, and evaluates them in order to compute the time a message spent in the *KMedia* module or the time which was needed for a cryptographic transformation of the message. Additionally, the tool also computes the number of messages dropped by the *KMedia* module and the number of messages dropped by the UDP socket. The difference is, that in the second case the messages are discarded before they enter the *KMedia* module processing space, thus *KMedia* does not know about their existence. Such situations occur when the system, in which *KMedia* executes, receives on the *KMedia* port more messages as it can handle. Therefore, it signals that the system's resources, such as the processor, message queues, but also *KMedia* itself, are exhausted. This is in contrast to the reason why *KMedia* fires the messages, as it happens mostly because of a wrong message format or content.

All the information provided by the *KMedia Statistics Reader* helps to analyze the behavior of the tested device under various measurement configurations. While the timing data characterize the performance of the *KMedia* module and of the individual cryptographic algorithms applied in the message transformation, the information on dropped messages signals system overloading or an incorrect *KMedia* message composition.

5.1.6 KMedia CPU Top, KMedia CPU Looper

The CPU utilization is one of the important indicators showing how suitable is to place certain network device into a desired environment. Particularly, if the device should have enough resources also for some other tasks except networking, it might be useful to know how loaded is the

processor at diverse data rates. In order to measure the processor utilization, *KMedia CPU Top* and *KMedia CPU Looper* benchmark tools were developed.

As already the name indicates, the design of the *KMedia CPU Top* utility was influenced by the standard Linux program called *Top*. The mechanism how *Top* computes the processor utilization, and what data it uses for that, is implemented also in this tool. However, the implemented CPU load measurement technique does not yield satisfactory results. The *Top*'s measuring mechanism is not suitable for situations when the processor has to deal with high interrupt rates produced by high network data rates. This is confirmed also by the *Netperf* manual [Jon07] and the website [12, Methods of measuring].

Therefore, the *KMedia CPU Looper* was developed in order to use a more appropriate CPU measurement mechanism. The tool is based on the *Netperf* utility which provides, except network performance measurements, also various techniques for the processor load measurement. Basically, the *KMedia CPU Looper* consists of a "looper" process which executes in tight little loops and counts as fast as it can. The processor utilization computation is based on the ratio of the value reckoned out when the system was under a load and a value generated when the system was believed to be in an idle state. This implies that a calibration step, to let the "looper" count on the system when it is idle, must be performed before the actual measurements begin. However, it is sufficient to do it just once at a particular system.

5.2 Test Set Up

A network test environment consists of a *DUT* and a *Network Test Equipment*. While the *DUT* is a "network forwarding device to which stimulus is offered and response measured" [Man98, page #2] a *Network Test Equipment*, called *Tester* in this document, runs a number of standard network tests and trials against the DUT. This way the performance of the device can be measured and it is verified whether the device works as expected.

In order to test a network device and to compare its performance with some other devices a cryptographic benchmark is needed which would emphasize the performance of all elements integrated in the processing path, as e.g. the network interface, CPU, memory subsystems, crypto accelerator. Many vendors use the IPsec packet processing as a benchmark. Since the IPsec offers security services at the internet layer but tests which would reveal the performance at application layer were needed *KMedia* module was designed, developed and used for benchmarking. Therefore, the term "*KMedia*-DUT" will denote the device under test with *KMedia* as the cryptographic benchmark.

The *Tester* is connected with the *KMedia*-DUT as it is suggested by the RFC [BM99] and depicted in Figure 5.2. This concept has the advantage that "the tester can easily determine if all of the transmitted packets were received and verify that the correct packets were received" [BM99, page #3].

Test traffic is transmitted through the *Tester*'s sending port to the DUT's receiving port. At the DUT the network data is accordingly transformed and forwarded back through a sending port to the *Tester*'s receiving port. If the *Tester* sends plaintext data, the DUT must response with ciphertext, and vice-versa. At *KMedia*-DUT the receiving and sending port equals, but on the *Tester*'s side they are different since one application is responsible for the message generation and an other one for counting of the forwarded messages.

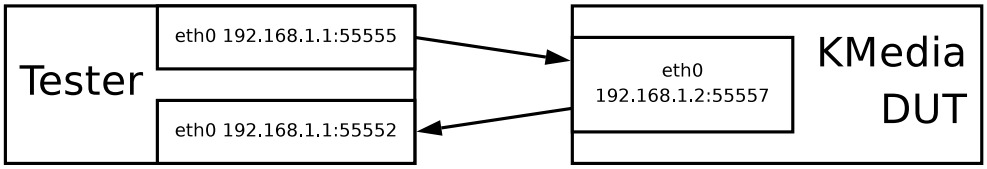

Figure 5.2: *KMedia* Device Under Test Topology (Notation in the boxes: Interface-Name IP-Address: Port-Number)

5.2.1 DUT Set Up

The *Freescale*'s Processor Board, labeled "MPC8349E MDS", in its factory default configuration was used as a Device Under Test (DUT). The state of all switches and jumpers on the board was during all measurements exactly as described in the "MPC8349E MDS Processor Board-Getting Started Guide" [Fre05a].

The embedded communication processor MPC8349E is from the PowerQUICC II Pro series which is the low end of the *Freescale*'s PowerQUICC product line. It is based on the SoC architecture and integrates a PowerPC processor core built on Power Architecture technology. To the advanced functional blocks on the SoC belong DDR memory, Dual Gigabit Ethernet, Dual Peripheral Component Interconnect (PCI), Hi-Speed Universal Serial Bus (USB) Controllers and a cryptographic accelerator termed also as a *Security Engine* (SEC). Figure C.1 on page 118 outlines the major functional units within the SoC and Appendix C lists their features.

Processor Board:	*Freescale*'s MPC8349E MDS
Processor Core:	e300c1 at 528 MHz, Revision 1.1, 32-bit Power Architecture
DRAM:	256 MiB 64-bit DDR1 at 264 MHz data rate
Crypto Accelerator:	Security Engine (SEC) 2.0 at 88 MHz
Ethernet Interface:	Dual Three-speed (10/100/1000 Mbit/s) Ethernet Controllers
Flash Memory:	8 MiB
Bootloader:	U-Boot v2009.03
Operating System (OS):	Linux Kernel 2.6.30
Crypto Benchmark:	KMedia ver. 1.0
Testing Tools:	*KMedia* CPU Looper (738030 calibration maxrate), *KMedia* Statistics Reader

Table 5.1: Device Under Test Configuration

According to the website [11] "The MPC8349E Family's SEC supports DES, 3DES, MD-5, SHA-1, AES, and ARC-4 encryption algorithms, as well as a public key accelerator and an on-chip random number generator." The SEC embedded in the tested processor was in version 2.0. Its features are listed in Appendix D together with characteristics of all other SEC versions provided by *Freescale* at the time of writing this thesis.

Table 5.1 summarizes some major parameters of the tested processor board. A detailed hardware specification of the embedded processor can be found in the document [fre09a] and, in order to see the functional characteristics, the reference manual [Fre05b] should be consulted.

The processor core frequency and the DRAM parameter values together with the Flash Memory size in the table 5.1 correspond to the values determined either by the bootloader or by the Linux OS[1] and can differ slightly from the statements in the official documents, e.g. the "Getting Started Guide" [Fre05a] specifies 533 MHz for processor core frequency. The SEC version can be retrieved from */sys* directory[2] but as its frequency value was not found in the system files it must have been calculated.

According to the *Table 4-35: "Configurable Clock Units"* from the processors' reference manual [Fre05b], the default frequency for the security core equals the clock of the coherent system bus (*csb_clk*) divided by three. With the *csb_clk* being 264 MHz, as reported by the bootloader, the SEC frequency gets the default value $\frac{264\ MHz}{3} = 88\ MHz$. As can be seen, the *csb_clk* is identical to the DDR clock, this is due to the processor board factory setting of the DDR clock mode [Fre05a, Switch 6.6 Configuration, page #4]. As reported by the reference manual, it is possible to switch off the SEC core or to enhance its frequency to $\frac{264\ MHz}{2} = 132\ MHz$. However, "The operating frequency of an algorithm accelerator is not a good indicator of its raw performance, because a higher frequency accelerator is likely to perform fewer internal permutations per clock cycle than the lower frequency accelerator." [fre08]. Nonetheless, all measurements were done just with the default 88 MHz frequency.

The employed Linux kernel in version 2.6.30 was configured for the mpc83xx systems[3] along with support for the *iptables* and, because of a bug in the silicon of the *Freescale*'s SoC, the PCI support must have been switched off. Cross compilation followed by a "gcc 4.3.3"-based cross development toolchain.

The root filesystem was downloaded from the website [3], but since its "Busybox" [4] 0.60.1 utilities were incompatible with the newer kernel, they were replaced by some tiny applications from the "Busybox" in version 1.10.2.

The KMedia's purpose is to classify network data received through a UDP socket, to determine whether the messages require security service and, if so, which cryptographic algorithm should be used. After transformation, each message is sent to an adequate destination read out from the message header. An incoming plaintext message is encrypted and authenticated and an incoming ciphertext message is checked on integrity first and then decrypted. This way the Linux kernel module *KMedia* provides the following security services to the UDP messages: confidentiality, integrity and data origin authentication.

The measurements were done with the KMedia in version 1.0. Its complete design specification can be found in Chapter 3. For some tests, which required to measure the time spent with message crypto transformation, *KMedia* was compiled with the defined macro `TIMESTAMPING` as described in the *Configuration and Monitoring* section of the *KMedia Specification* on the page 37.

5.2.2 Tester Set Up

An ordinary desktop computer with characteristics listed in Table 5.2 served as a UDP message generator, counter and also as a message checker. Thus, its role was not only to send test traffic to the *KMedia*-DUT but also to control whether the device replied with an adequate number of messages of correct format and content.

[1] The parameters were read out from the system files of the */proc* directory.
[2] From file `/sys/bus/of_platform/devices/e0030000.crypto/modalias`.
[3] By executing `make mpc83xx_defconfig`.

Processor:	2 x 2.80 GHz, 32-bit Intel Pentium 4 Architecture
DRAM:	512 MiB 64-bit SDRAM at 400 MHz data rate
Crypto Accelerator:	not available
Ethernet Interface:	10/100 Mbit/s
OS:	Debian GNU/Linux 5.0.3 (lenny), Linux kernel 2.6.32
Crypto Benchmark:	KMedia ver. 1.0
Testing Tools:	Netperf, Socat, *KMedia* Header Engine, *KMedia* Message Checker

Table 5.2: *Tester* Configuration

The employed Linux kernel 2.6.32 was compiled with the gcc 4.3.3 based toolchain with a default kernel configuration and thus with no network performance optimization.

Since *KMedia* uses non-deterministic cryptographic algorithms, meaning that every time the same block of plaintext data is encrypted the ciphertext output is different, a ciphertext message coming from *KMedia*-DUT could not be easily compared with a ciphertext message saved on the *Tester* in order to verify it. Therefore, the incoming ciphertext messages must have been decrypted before verification "on the flow" by a *KMedia* module executing on the *Tester*.

5.2.3 Discussion

A problem with the DUT topology as shown in Figure 5.2 is that the *Tester*'s performance may be insufficient to test advanced networking SoCs with cryptographic acceleration. As it is also stated in an article [13] "this setup is rarely used due to the cost of networking testers that support high IPsec data rates."

A more appropriate scenario is the system under test topology depicted in Figure 5.3 inspired by [KH09, Figure 2]. Two identical DUTs are used to exchange cryptographic data and the *Tester* sends and receives only cleartext traffic.

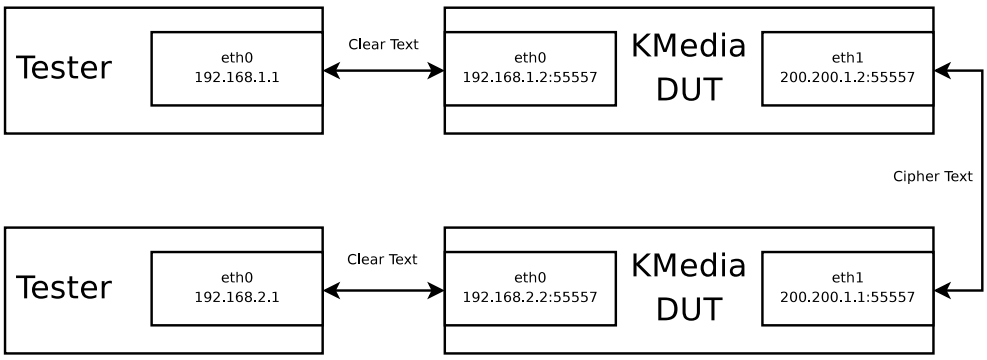

Figure 5.3: System Under Test Topology (Notation in the boxes: Interface-Name IP-Address:Port-Number)

Unfortunately, only one device was available for the planed *KMedia* measurements and therefore this test topology could not be built. On the other hand, although there were some performance

issues with the test equipment in the configuration as listed in Table 5.2, they were not many and were not so serious that it would make the testing impossible.

As no crypto accelerator was available for the *Tester*, *KMedia* could use for all its crypto operations only the software driver with the crypto algorithms implemented in the Linux kernel. This put a high load on the *Tester*'s CPU, particularly when the DUT was transforming the messages with its SEC accelerator. In most cases, the *Tester* was enough powerful to keep up also with the traffic produced by the DUT's accelerator, but, unfortunately, situations occurred in which the content checking of incoming ciphertext messages, which were longer than 2048 bytes, must have been turned off, because the *Tester*'s CPU was unable to process them. Such cases are mentioned in every relevant test evaluation.

5.3 Test Messages

Since *KMedia* and the tools used for network testing operate on the application layer of the TCP/IP network stack, they all work with network data units called "*messages*", explained in Section 1.3. So, although the RFCs describing the methodology for testing network interconnect devices[4] speak in terms of "*frames*" on the link layer, this document will stay at application layer with "*messages*". It implies that the message sizes are used in the resulting tables, graphs and also for computing the network throughput.

It is also important to recall the structure of a *KMedia* message: The content of the first 28 bytes of every plaintext *KMedia* message is the so-called "*KMedia* Extended Header" composed of a *KMedia* header and a *KMedia* Address Field. This Extended Header is followed by the *KMedia* payload with some user data. The ciphertext state *KMedia* message consists of an encrypted *KMedia* payload with a prefixed *KMedia* header. The payload in a ciphertext message is more complicated as it contains the payload and the Address Field from the plaintext state message and some additional cryptographic data. For more details Section 3.2 from the *KMedia Specification* should be consulted. The following subsections describe what messages were used for testing, their sizes and formats.

Note that in order to relieve the *Tester* from the burden of creating the messages "on the flow" during tests, the messages of defined sizes and with correct content were prepared before the measurements. Appendix B describes the methods and tools employed in the message generation.

5.3.1 Message Sizes

To get a full characterization of the device under test not only the minimum and maximum sizes of messages were taken into account but also many in between. The second column of Table I.1, placed in Appendix I, lists the plaintext message sizes used in the *KMedia*-DUT measurements. Additional columns illustrate the impact of a message size on the frame size, *KMedia* payload and the encryption size. The row in gray indicates that the Maximum Transmission Unit (MTU) was reached which corresponds to a 1472 bytes long message, therefore all messages above that size are fragmented into multiple frames.

In the *KMedia*-DUT testing environment the frame size corresponded to the sum of a 14 bytes long Ethernet header, 20 bytes long IP header, 8 bytes long UDP header and of the message size,

[4] E.g. [BM99], [Bra91]

which is a sum of a 4 bytes long *KMedia* header, 24 bytes long *KMedia* Address Field and of the *KMedia* payload.

The listed encryption size does not provide the real encryption size, nor the message size in ciphertext state. It is based only on the *KMedia* payload and the 24 bytes long *KMedia* Address Field, but during the encryption process some additional padding data are always added to the message. As it was explained in Section 3.2.5 on the page 31, the maximum padding size which can be added consists of 16 bytes, so the real encryption size will be between the value shown in the table and the value summed up with 16. The encryption size is mentioned in the table just for information to see the approximate data length which must have been transformed by cryptographic algorithms in the tested *KMedia*-DUT.

The minimum message size comes from the minimum *KMedia* payload, which has only one byte. According to the RFC [BM99] such unrealistically small sizes help to characterize the per-message processing overhead of the DUT. On the bottom of the table is the maximal size computed from the maximum IP datagram size[5]. The upper half of the table, except the first line, is inspired by the recommended frame sizes from the "Benchmarking Methodology" RFC [BM99].

Due to an cryptographic overhead occurs the fragmentation of the ciphertext messages earlier then of the messages in plaintext state. From the sizes used for testing, already the 1500 bytes long plaintext messages were fragmented after a cryptographic transformation. To see clearly the effect of fragmentation on the performance of a device and on the resulting throughput, a message just four bytes above the MTU was used for plaintext message forwarding without encryption.

5.3.2 Message Formats

Three types of messages were used: *forward*, *plaintext* and *ciphertext*. The first one was for testing the forwarding capability of the DUT and such cleartext messages were not cryptographically transformed. The second type messages were in the plaintext state which the DUT should encrypt and authenticate. Finally, the third type represented messages in the ciphertext state which should be decrypted only after a successful authentication check.

The types *forward* and *plaintext* had the same Address Field. This field was encrypted together with the *KMedia* payload in the *ciphertext* messages. The configuration of the *KMedia* header depending on the message type can be found in Appendix I.2 where also the full content of the *KMedia* Extended Header as used in the *forward* messages during *KMedia*-DUT measurements is placed.

All bytes following the *KMedia* Extended Header belong to the *KMedia* payload and carry user data. To find the right content for these bytes two documents on benchmarking methodology were studied. According to the Appendix C in the RFC [BM99], it is recommended to fill the UDP data part of a "UDP echo request on Ethernet" test frame with incrementing bytes, repeated if required by frame length. This RFC explains also that "the data should not be all bits off or all bits on since these patters can cause a 'bit stuffing' process to be used to maintain clock synchronization on WAN links. This process will result in a longer frame than was intended." The internet draft "Benchmarking IPsec-Methodology" [KH09] mentions that "instrumentation data should be present in the payload" and "it is optional to have application payload".

[5]The computation steps are shown in Section 3.2.5.

Adding instrumentation data for identifying the messages in the *KMedia*-DUT testing environment in order to allow flow identification and/or timestamping was not implemented because it would increase the overhead needed to develop the testing tools which would be able to generate, collect and analyze such data. It would also put more load on the Tester's resources. The above mentioned draft does not specify closely the "application payload" so it was decided to follow the advice from the RFC and to fill the *KMedia* payload of a plaintext *KMedia* message with the repeatedly incrementing bytes, i.e. 0x00, 0x01, 0x02, .., 0xFE, 0xFF, 0x01, 0x02, ...

5.4 Performance Characteristics

Basically, the following three performance characteristics of the device under test with the *KMedia* as a benchmarking application were measured: *throughput*, *CPU load* and *latency*. What they characterize, what methods and tools were used to determine their values, and in what units of measurement, will be described in the following paragraphs.

Throughput *Throughput* is the maximal data rate at which the number of test messages transmitted by the DUT equals the number of messages sent to it by the test equipment. According to the RFC [BM99] none of the offered messages must be dropped by the DUT for at least 60 seconds. This duration was extended to 120 seconds for finding out the throughput on *KMedia*-DUT in order to get more accurate results. The throughput values were obtained by the network benchmarking tool *Netperf* and are expressed either in Messages Per Second (m/s) or in units of bits per second, Megabits Per Second (Mbit/s). It should be not forgotten that the bits are computed according to the sizes of messages, and not frames, transmitted by test equipment.

Note that, according to an article on the website [13], this "zero loss" testing is "generally ignored by networking equipment vendors when testing their end systems". It is due to the fact that acceptable loss rates produce better results. As reported by the same website, the tests performed by the *Freescale* company shown that results at loss rate smaller then 0.001 % were about 25 % better compared to zero loss. Some vendors use loss rates up to 0.1 %. Nonetheless, when determining the throughput and also the following two performance characteristics at *KMedia*-DUT no message could be discarded.

CPU Load In order to get the utilization tax of the processor, *KMedia*-DUT was first put under a constant network load induced by a certain rate at which messages were sent from the testing equipment. After approximately 30 seconds the *KMedia CPU Looper* started to measure the processor load for the duration of 60 seconds. The calibration of the tool was performed on the DUT, described in Section 5.2, before *KMedia* module insertion and the measured local maxrate was 738030 incremental counts per second. The CPU Load results are provided at throughput and at various smaller-than-throughput data rates as a value between 0 and 100 %, with 100 % representing the most loaded state.

Encryption/Decryption Latency Since this measurement is based on the timing data obtained through the *KMedia* sysfs interface, the module must have been compiled with the enabled time measuring functionality which was described in Section 3.4.2 on the page 38. Afterwards, such module was put under a constant network load caused by the messages coming from the *Tester* at a certain steady state rate lower than the throughput. Getting the time information at a maximal rate would lead to dropping of the messages because

the timing functionality increases the processor load and was not present at the time of the throughput measurements. For reading out the relevant sysfs files the utility *KMedia Statistics Reader* was deployed. The tool's task was also to compute and display the latencies, caused by the *KMedia* processing and cryptographic transformation, on the user's terminal. The test took 60 seconds, but the results were obtained just from the last 30 seconds. In the first 30 seconds the *KMedia*-DUT should get stabilized and settled.

When a message intended for a cryptographic transformation is received by the *KMedia*, a number of instructions, as e.g. determining what kind of crypto processing is required or how many padding bytes must be added, is performed before it is passed to an appropriate crypto driver. This *KMedia* overhead exists independently of the driver, and its underlying crypto functions. Therefore, it was also measured how much time *KMedia* spends with preparing a message for a transformation and how long does the real transformation, out of the scope of *KMedia*, performed by the crypto interfaces and a relevant crypto driver, take.

Ideally, the "Encryption/Decryption Latency" would be obtained as described in the draft called "Benchmarking IPsec Methodology" [KH09] in the sections "IPsec Encryption Latency" and "IPsec Decryption Latency". It is recommended, that in the topology depicted in Figure 5.2 the testing equipment sends to the DUT a 120 seconds long stream of messages, in which, after 60 seconds, one message is tagged and the time when it is fully transmitted is recorded. Afterwards, the *Tester* must record the time when the tagged message, already transformed by the DUT, comes back. Thereafter, the *Tester* computes the latency according to the noted transmit and reception time. This test should be repeated 20 times with the resulting latency value being the average of the recorded values.

Unfortunately, implementing this "tagging" functionality in the *KMedia*-DUT testing environment could be quite complex. Since during the cryptographic transformation the whole *KMedia* message undergoes a change, there is no place left for the tag. The best position for it would be in the *Option field* of the IP header which is variable in length and offers an option class "debugging and measurement" as specified by the RFC [Pos81]. Although implementing this tagging at the internet layer would be desired, it is out of the scope of this thesis which handles everything at the application layer.

5.5 Test Environment Parameters

The three performance characteristics introduced above: *throughput*, *CPU load* and *latency*, were measured under various conditions composed of various environment variables in order to characterize the *KMedia*-DUT accurately. This section begins with a specification of the security parameters, i.e. what crypto algorithms and crypto drivers were utilized. Afterwards, the set of the data rates, at which the CPU load was measured, is defined. Finally, it will be described what combinations of the test parameters were used for getting the three performance characteristics of the tested device.

5.5.1 Security Parameters

Following parameters must be considered to get an accurate overview of the *KMedia*-DUT crypto performance: the crypto driver, the cryptographic algorithm, its security key size and also the

message size. Furthermore, it must be also distinguished between an encryption and a decryption operation.

All tests concerning the crypto performance were executed separately, either by utilizing the crypto algorithms implemented in the Linux kernel and accessible through a Linux software driver, or by employing the crypto algorithms of the Freescale's Security Engine (SEC) accessible through the crypto driver *Talitos*, available in the Linux kernel since the version 2.6.27.

As it was the primary goal of the *KMedia* module development to test the performance of the SEC unit utilizing the driver *Talitos*, all its algorithms, as offered in the kernel 2.6.29, with at least one key length were employed in the measurements. They are actually combinations of three authentication and two encryption algorithms each having a secret key of a defined length. To illustrate the impact of the key length on the device's performance, the Advanced Encryption Standard (AES) algorithm was tested with two different key sizes in combination with the SHA256. *KMedia* does not support the AES with a 256 bits long secret key by default, but it can be set through the *KMedia* header file: `kmedia.h`. Key sizes of all other algorithms were constant in all combinations as also Table 5.3 shows. Since all the AEAD ciphers execute in the same operation mode, shortened names, as listed in the first column of the table, will be used in order to refer to them in the remainder of this thesis, as also in the test results and in the diagrams. The Section 2.3 on the page 9 gives more details about the AEAD algorithms and explains also the used "CBC" and "HMAC" notation in the table.

Notation	Authentication Alg. -	Key Size	Encryption Alg. -	Key Size
SHA1-AES	HMAC-SHA1	160	CBC-AES	128
SHA1-DES3	HMAC-SHA1	160	CBC-DES3_EDE	192
SHA256-AES	HMAC-SHA256	256	CBC-AES	128
SHA256-AES256	HMAC-SHA256	256	CBC-AES	256
SHA256-DES3	HMAC-SHA256	256	CBC-DES3_EDE	192
MD5-AES	HMAC-MD5	128	CBC-AES	128
MD5-DES3	HMAC-MD5	128	CBC-DES3_EDE	192

Table 5.3: Algorithm (Alg.) Combinations Used in Measurements (All Sizes Are in Bits)

Values of the secret keys were set at the initialization time of the *KMedia* module and were not changed thereafter. They consist, in their full length, of bytes filled with the hexadecimal value "0xAA".

5.5.2 Data Rates Smaller Than Throughput

The various, smaller-than-throughput, data rates[6] at which the CPU Load was measured, were chosen in order to get an optimal characterization of the DUT's performance for all message sizes. To obtain enough values for small messages, low data rates with small increasing steps must have been used. Larger messages did reach higher data rates and also did not need that low granularity as used for the small messages. Therefore, at the beginning, the data rates increase

[6]The various data rates at which the CPU load was measured are: 0.1; 0.2; 0.3; 0.4; 0.8; 1.2; 2.4; 4.8; 9.6; 14.4; 19.2; 25; 30; 35; 40; 45; 50; 60; 70; 80 and 90 Mbit/s.

just with 0.1 Mbit/s steps but afterwards these steps increase evenly until the difference between two successive tested rates reaches 10 Mbit/s. Messages over 1472 bytes were not tested on some of the smallest rates, since the CPU was mostly loaded under 1%.

5.5.3 Tested Combinations of Parameters

Quite many combinations of parameters can define the context in which the transformations could be tested. There are seven cryptographic algorithms, each of it has two modes, the encryption and decryption, they all can be executed either by the software or by the *Talitos* driver and, finally, 22 message sizes are prepared for testing. These make together 616 possible combinations which could be tested on latency, throughput, CPU load at throughput and CPU load at 21 other, lower-than-throughput, message rates. This is a large set to test without having a full automated testing environment available, as it was the case with the *KMedia*-DUT testing. In order to shorten the time needed to perform the measurements, neither all algorithms, nor all message sizes were engaged in the combinations used to obtain all three performance characteristics of the *KMedia*-DUT.

The *throughput* performance characteristic together with the *CPU load* at the throughput was measured for all the algorithms listed in Table 5.3 and for the following plaintext message sizes: 214, 1238, 4096 and 32768 bytes, and their respective sizes in the ciphertext state. The four sizes were chosen in order to characterize the device under test with at least two messages which are smaller than the MTU and with another two which are longer and must be therefore fragmented. Additionally, the SHA256-DES3 and the SHA1-AES were tested on throughput at encryption with all message sizes as listed in the Table I.1.

The *CPU load* at various message rates was obtained only with the SHA1-AES algorithm with plaintext message sizes lower than the MTU, 29–1472 bytes, and with two larger ones, 4096 and 32768 bytes. In this case, the main goal was to get the behavior of the *KMedia*-DUT during transformation of messages with lower sizes. The 4096 and 32768 bytes long messages were chosen as representatives of the large messages for the following reasons: when the DUT was tested on throughput during encryption with the *Talitos* driver, begun the CPU load to drop at 4096 bytes long messages. The 32768 byte messages were one of those at which the 100 Mbit/s link limit was already reached.

To conclude, the *Latency* measurements were also done only with the SHA1-AES but for all message sizes as shown in Table I.1 and at a data rate smaller than the throughput at the respective message size.

Now it will be explained why the SHA256-DES3 and primarily the SHA1-AES algorithms were chosen for the tests, in which not all available crypto algorithms were engaged. As the results will show, the SHA256-DES3 is the worst case, the most heavy, AEAD algorithm for the software crypto driver. Its lighter version, which generates a shorter message signature, the SHA1-DES3, is still widely used [fre08]. According to the website dedicated to the AES development [26] and the RFC [FGK03], the cipher algorithm AES was developed to replace the “single” DES and the Triple-DES (DES3). Not that only it is proved that the AES is more secure, it performs also much better than the DES3, what was confirmed also by the *KMedia*-DUT performance measurements. The SHA1 with the AES is already used in the default AEAD algorithm of the Secure Real-Time Transport Protocol (SRTP) [BMN+04] and every IPsec implementation must support authentication by the HMAC-SHA1 and encryption by the CBC-AES [Man07].

The MD5 hash function used for authentication was not considered since, as it was explained in Section 2.3.1, it is not secure enough and should not be used.

5.6 Testing Methods

Until now it was described what was tested and in what context, the next subsections explain the technical realization of the measurements along with the reason for performing the test. The command line options and arguments needed to configure the utilities employed in the tests will be not presented since they can be derived from the use-case examples introduced to each particular tool in the Appendix A. Also note, that all messages of all three types and of all sizes were generated[7] before the tests took place.

5.6.1 Forwarding

Forwarding in the context of the *KMedia*-DUT measurements stands for receiving a data packet and transmitting it without any UDP payload modification to a new network address. Measuring the behavior of the device whose only purpose is to do forwarding was important in order to get some reference data, to which the results from crypto transformation performance measurements could be compared. Two forwarding tests were performed which will be now explained:

Network Address Translation (NAT) Forwarding At this type of test the DUT was configured with the *Iptables*-NAT to enable packet forwarding within the Linux routing table. As this forwarding happens on the internet layer it could be also described as a "plain IPv4 forwarding". It shows nicely the performance degradation associated with the extra overhead caused by receiving/transmitting messages through UDP sockets from the application layer. The *Iptables*-NAT configuration is discussed in Appendix I.4. Only throughput and processor utilization at throughput was measured with *KMedia* forward messages with sizes of up to 1472 bytes.

***KMedia* Forwarding** This test type could be called also as "UDP socket forwarding" since the purpose was to test the performance of the *KMedia*-DUT in situations when the messages were received and then transmitted in the same state through a UDP socket. *KMedia* forwarding can be considered to represent the theoretical best case performance of all cryptographic transformations, be it through a security engine or software implemented algorithms. In case a crypto transformation would be "perfect" with zero processing latency, it could equal *KMedia* forwarding performance, but not exceed it. At this type of test, the throughput along with the CPU load at throughput and at different data rates using *KMedia* forward messages of all sizes, as defined in Table I.1, was measured.

The following Figure 5.4, with "KM" denoting *KMedia*, illustrates the device under test topology for the *KMedia* Forwarding. Also the employed benchmarking tools, with the port numbers at which they were listening for the incoming messages, and the *KMedia* forward message flow is depicted. For the NAT Forwarding the topology was the same, except that no *KMedia* module was inserted in the DUT.

[7] The message generation is described in Appendix B.

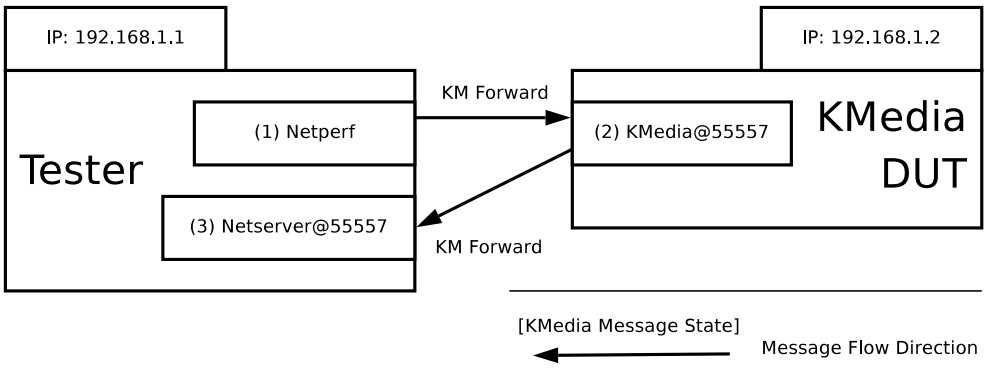

Figure 5.4: *KMedia* Forwarding Test Set Up

For the *KMedia* Forwarding the procedure is as follows. First, *Netperf* transmits at a defined rate messages of a forward type to the *KMedia*-DUT. *KMedia* checks whether the messages have a correct *KMedia* header and transmits them back to the *Tester*'s device to the *Netserver*'s port. After a test run finishes, *Netserver* informs *Netperf* how many messages were received so that *Netperf* can calculate the drop rate. According to the number and size of the successfully forwarded messages displays *Netperf* also the achieved data rate in bits per second.

5.6.2 Encryption Transformation

In this kind of tests *KMedia* plaintext messages are transmitted to the *KMedia*-DUT. At *KMedia* they are encrypted and authenticated with an appropriate algorithm and sent back to the *Tester* as *KMedia* ciphertext messages.

Since the incoming plaintext messages instruct *KMedia* to use a default algorithm for their transformation[8], the crypto algorithm under test is simply configured by defining the default algorithm through the *KMedia sysfs* interface. During the transformation process the ID of the used algorithm is noted in the *KMedia* header which is an important information for the decrypting module.

The Figure 5.5 displays the *KMedia*-DUT test topology along with the utilities involved in the measurements together with the port numbers at which the tools were waiting for the incoming UDP messages. To save some place *KMedia* is denoted in the figure also in an abbreviated form as a "KM". First, *Netperf* transmits periodically, in defined intervals, plaintext messages to the *KMedia*-DUT. The messages are filled with data read in from a file containing the *KMedia* plaintext message. At the DUT the messages are encrypted using the configured default AEAD algorithm and sent to the *Tester*'s *KMedia* module where they are decrypted. Afterwards, the resulting plaintext messages are transmitted to the *KMedia Message Checker* to compare their content with the content of the plaintext message which was originally transmitted by the *Netperf*. After a successful message check they are finally received by the *Netserver* which increments the number of successfully encrypted messages by the *KMedia*-DUT. At the test end obtains the *Netperf* the number of received messages from the *Netserver* and computes the achieved data rate according its value and the size of the transmitted plaintext messages.

[8]The "Crypto Algorithm ID" field in their *KMedia* headers is set to zero.

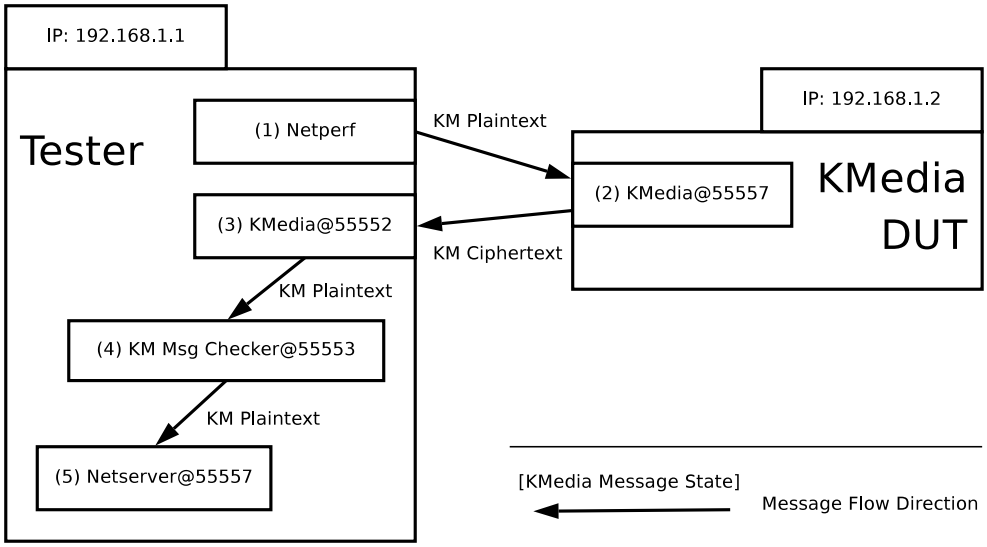

Figure 5.5: Encryption Transformation Test Set Up

Since the *Tester*'s decryption was performed by software, the decryption of the ciphertext messages increased highly the load on the *Tester*'s two CPUs. In situations when the DUT was encrypting also by software no problems raised, since the DUT's processor is much less powerful. However, when the *KMedia*-DUT used the crypto accelerator, the *Tester* was not able, in some cases, to decrypt the messages at the same high rate as they were encrypted by the DUT. This was the fall when the algorithms SHA1-DES3, SHA256-DES3 and MD5-DES3 were tested with messages longer than 2048 bytes. Therefore, in these situations, when the *Tester* was not able to decrypt the messages as fast as the *KMedia*-DUT was encrypting them, no message check was performed and the *KMedia*-DUT transmitted the encrypted messages directly to the *Netserver*.

Note, that the last two addresses in the Address Field of the used *KMedia* plaintext message, depicted in the Example I.1, equal. The equal addresses have sense for the *KMedia* ciphertext message generation and the following decryption tests at the DUT, but during encryption tests they would cause that the *Tester*'s *KMedia* would send the decrypted messages back to itself. Therefore, the *KMedia* module on the *Tester* was compiled with the enabled macro `CHANGE_MSG_DST`. This instructed *KMedia* to send the messages to another destination address as proposed by the Address Field in the *KMedia* message. This macro and a new destination can be set in the *KMedia* header file.

5.6.3 Decryption Transformation

To test the decryption transformation a set of unique ciphertext messages was transmitted to the *KMedia*-DUT which forwarded them after a successful transformation in a plaintext state back to the *Tester*. Since after the decryption all messages were carrying the same content, they could be easily checked on correctness by comparing them with a *KMedia* plaintext message which was used for the creation of the testing ciphertext messages.

No default algorithm must have been defined at the *KMedia*-DUT because the particular algorithm needed for the decryption transformation was identified by the ID in the messages' *KMedia* header, set at the encryption time.

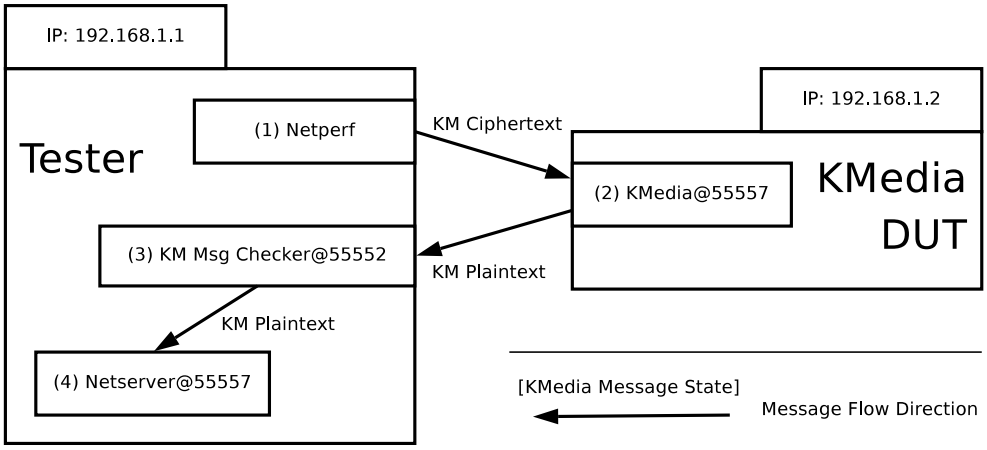

Figure 5.6: Decryption Transformation Test Set Up

The test topology, along with the engaged utilities, is illustrated in Figure 5.6. For lack of space, the name *KMedia* was shortened to "KM". At the beginning of each test run copies *Netperf* by default the first 32 *KMedia* ciphertext messages from a file into its buffers dedicated for sending. These read in ciphertext messages are of equal sizes but of different content. Afterwards begins *Netperf* to transmit them in a ring one after another[9] in defined intervals, e.g. it sends one message every one millisecond, to the *KMedia*-DUT. Thereafter the messages are decrypted with an appropriate AEAD algorithm and transmitted to the *KMedia Message Checker* utility residing on the *Tester*. The tool verifies whether the messages correspond with the *KMedia* plaintext message used for generation of the ciphertext messages. Finally, when their content is correct, they end up in the *Netserver* which increments the counter of successfully decrypted messages by the *KMedia*-DUT. At the end of a test run calculates *Netperf* the achieved data rate according to the number of messages received by the *Netserver* and the size of the transmitted ciphertext messages.

[9] *Netperf* transmits messages rotating through a ring of buffers. This *Netperf*'s feature is explained in Appendix A.1 on the page 89.

6 Results

On the basis of the various measurements described in Section 5 the following questions will be answered in this chapter: Does the cryptographic accelerator improve the performance of the security services offered by *KMedia*? If yes, how many times more efficient is the system with the accelerator compared to a pure software solution?

The *Tester* could send 100 Mbit/s at maximum and therefore the highest possible data rate the *KMedia*–DUT had to deal with was 100 Mbit/s uni-directionally. Since the reported data rates consider only message sizes[1] they do not correspond to the complete data traffic between the DUT and the *Tester* and therefore they exhibit full network utilization already at sizes lower than 100 Mbit/s.

All algorithms employed in the measurements are of the Authenticated Encryption with Associated Data (AEAD) type and were composed according to the generic composition scheme described in Section 2.3.2. These tested algorithms provide authentication with a hash function in the HMAC mode and confidentiality with a block cipher in the CBC operation mode. Throughout this section only the hash function in combination with the block cipher name will be used to describe the respective AEAD algorithm. The Table 5.3 summarizes the used notation.

First in this section, the device is characterized at forwarding when no crypto operations are performed. Afterwards, throughput of various crypto algorithms implemented in the security engine controlled by the *Talitos* is compared to the maximum data rates obtained with the software version of the algorithms. The evaluation of the latency measurements will define the highest possible advantage of the SEC over cryptographic software and the final section will analyze the device's processor load at distinct data rates and message sizes.

6.1 Forwarding

In the first tests, the *Throughput* and *CPU Load* at *Throughput* was obtained with no *Security Context* utilizing the *NAT- and KMedia-Forwarding* measurement method. The obtained values are presented in Table J.2 on the page 151.

[1] No Ethernet, IP nor UDP header and no preamble and trailer bytes are contained in the size.

6.1.1 *KMedia* vs. NAT Forwarding

The Figure 6.1 visualizes the comparison of the *KMedia*-Forwarding with the NAT-Forwarding Throughput. As can be seen, the degradation caused by extra instructions during the *KMedia*-Forwarding is obvious for the message sizes from 29 to 726 bytes. Compared to the NAT-Forwarding provides the *KMedia*-Forwarding 2.5 times less throughput at 214 bytes long messages. This difference grows to 3 times at the smallest 29 bytes messages, where protocol stack overheads dominate, and converges to 1 at the larger sizes due to the Ethernet link limit.

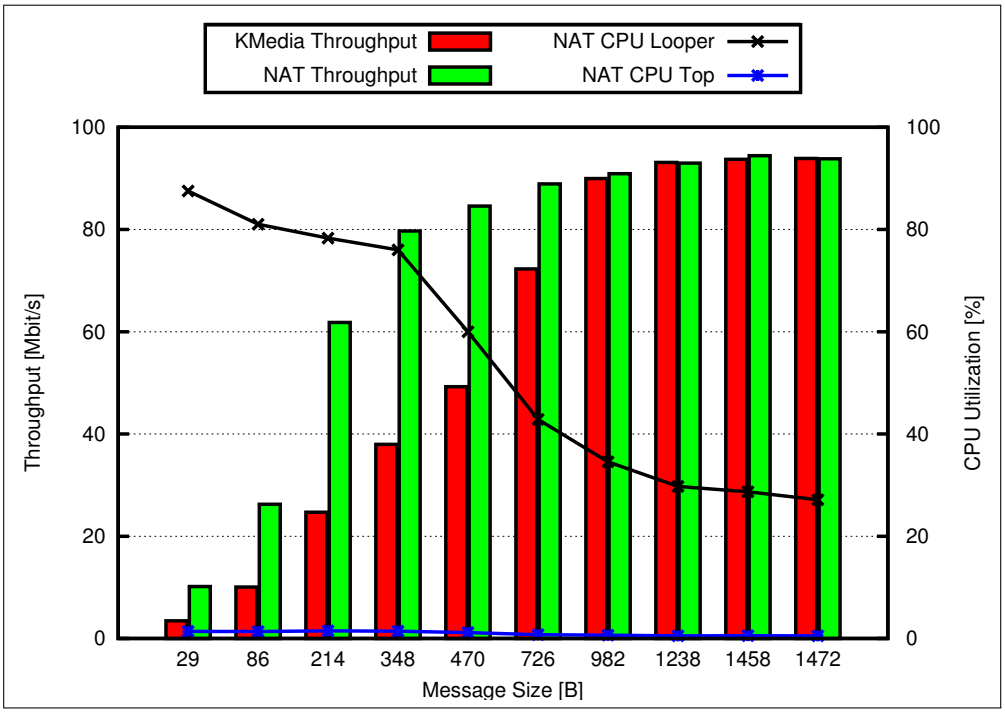

Figure 6.1: NAT vs. *KMedia* Forwarding

The smaller the messages the higher number of them must be forwarded and the more instructions must be executed by the processor in order to reach certain data rate. Therefore, until the 100 Mbit/s throughput limit of the network test equipment is reached, the processor performance is mostly the bottleneck which does not allow higher data rates. Theoretically, the processor should be utilized on 100% by the instructions associated with the forwarding of every single message. First after the network limit is reached the utilization should start to drop, since the processor is able to forward more messages of large sizes than the network can transmit.

As Figure 6.1 demonstrates on the CPU Load characteristic, the *Tester* device employed in the measurements did not have enough resources to fully utilize the *KMedia*-DUT at small message sizes during NAT-Forwarding. Although the link limit was not reached, the CPU Load, as measured by the *KMedia CPU Looper*, does not cross 88%. There are still some free processor cycles which could be engaged in the message forwarding and therefore, in case there are no other constraints in the system, it is believed that the real throughput could be a bit higher compared to the obtained results.

To examine two different CPU utilization measurement methods implemented in the tools *KMedia CPU Top* and *KMedia CPU Looper*, described in Section 5.1, Figure 6.1 displays results obtained

during NAT-Forwarding by both tools. According to the data, the *KMedia CPU Top* shows significantly different results as the *KMedia CPU Looper*.

According to the *KMedia CPU Top*, 38151 messages with a 86 byte length were transmitted per second at the CPU utilization of only 1.37%. From this follows, that $\frac{38151}{1.37} \approx 27847$ messages utilize the processor on 1%. Therefore, at 100% 2784700 messages should be forwarded. This corresponds to the throughput of 1.9 Gbps[2] which is 4 times more than the throughput obtained on the *Freescale*'s MPC8548E device running at 1.33 GHz[3] although the *KMedia*-DUT processor executes with 528 MHz.

These results confirm what was stated in Section 5.1.6, that the *Top*'s processor utilization measuring mechanism is not suitable for obtaining an accurate CPU load in an environment with high data rates. The next Figure 6.2 compares also the results of the *KMedia CPU Top* with the *KMedia CPU Looper* obtained at *KMedia* forwarding, but afterwards the tool was not further used.

6.1.2 *KMedia* Forwarding: All Message Sizes

The Figure 6.2, with the data from Table J.2, illustrates the *KMedia*-DUT's processor utilization during *KMedia*-Forwarding and the influence of the message fragmentation on the performance of the system. In this case the *Tester* was able to fully utilize the device under test, which can be seen on the values of the CPU load reaching 100%. Higher message rates would cause the device to drop the messages due to insufficient resources. First, when the throughput hit the ethernet link limit, the CPU load started to fall.

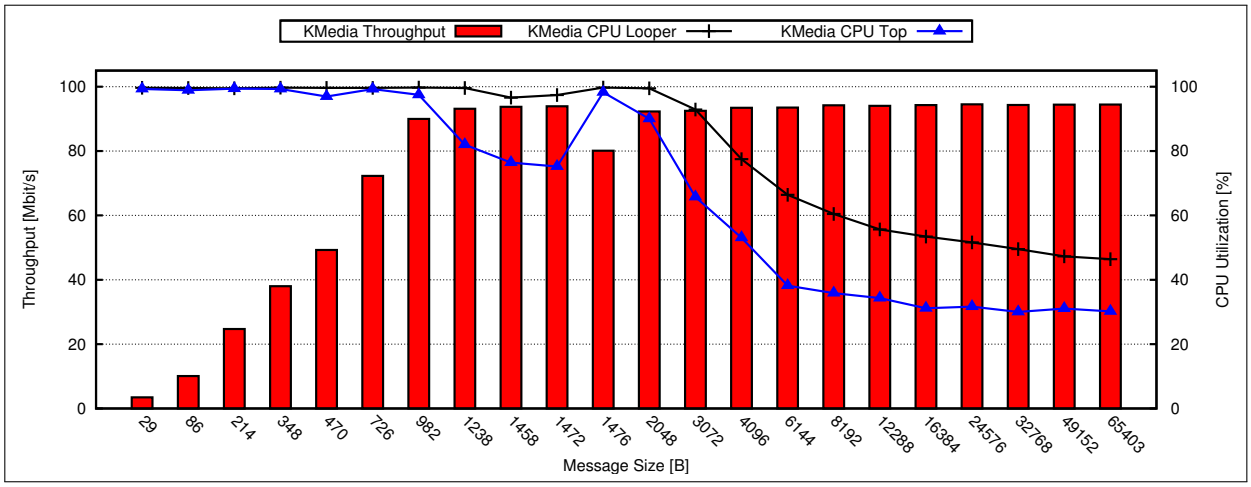

Figure 6.2: *KMedia* Forwarding

The performance degradation caused by the message fragmentation is obvious at a 1476 bytes long message. At this message the throughput drops to 80.11 Mbit/s from the 93.91 Mbit/s measured at a 4 bytes smaller message and the processor load which was slowly falling since 1238 byte messages jumps back to a value close to 100%. Nonetheless, with larger messages the overall overhead caused by the message forwarding becomes smaller and at the largest messages,

[2] $2784700\ messages\ per\ second * 86\ bytes\ per\ message * 8\ bits\ per\ byte = 1.9\ Gbps$

[3] Tested by the *Freescale* company in the document [fre08, Section 1.2].

according to the *KMedia CPU Looper*, the processor utilization drops already to 50%. The remaining 50% could be used by other tasks running on the processor.

Again, the *KMedia CPU Looper* is compared to the *KMedia CPU Top* measurement method and, as it can be seen, their results are similar when the processor is fully utilized, but in case the load starts to drop the *KMedia CPU Top* shows values about 20% lower than the *KMedia CPU Looper*. From this view, the *KMedia CPU Top* is more optimistic and estimates more idle CPU time.

6.1.3 Summary

The Figure 6.2 shows clearly that the system performance during forwarding is limited by the CPU at small messages and by the network link limit at large messages. By dividing the CPU frequency with the number of forwarded messages it can be computed that the *KMedia*-Forwarding consumes ∼2.8 times more CPU cycles per message than the NAT-Forwarding. Also, the fact that at the message size of 1472 bytes only ∼2.5% of the CPU is idle at *KMedia*-Forwarding while during the NAT-Forwarding already ∼70% of the processor can be used by other tasks, shows that forwarding on the application layer using UDP sockets is much more heavier than the simple NAT forwarding on the IP layer.

Most of this performance degradation is caused by the message copying between the UDP socket buffer, created and used by the *KMedia*, and a buffer utilized by the protocols from the TCP/IP network stack. The NAT-Forwarding does not need the UDP socket buffer and therefore no such message copying must take place.

6.2 Cryptographic Transformation Throughput

In this section results are presented which were obtained during encryption and decryption transformations performed by all crypto algorithms from Table 5.3 and executed either by the software or *Talitos* driver. The throughput and the CPU load at throughput was measured at message sizes of 214, 1238, 4096 and 32768 bytes. All acquired values are shown in the Tables J.3 and J.4 on the page 152 and 153.

6.2.1 Encryption vs. Decryption Throughput

As it was observed during the measurements, the two modes, encryption and decryption, place a different load on the processor which leads to different values of the throughput. In a real scenario, when *KMedia* is used to secure a communication channel, both encryption and decryption is utilized and the overall performance depends on the mode with the smallest throughput. Therefore, this section compares first the encryption- with the decryption-throughput and analyzes afterwards only the “heavier” operation, i.e. the operation which places more load on the system and exhibits therefore a lower performance.

The Table 6.1 lists values obtained by subtracting the number of messages encrypted per second from the number of messages decrypted per second at various message sizes, crypto drivers and algorithms. The resulting differences in the throughput show that mostly more messages were transformed during decryption than during encryption. There are only three cases when the

Driver	Msg Size	SHA1-AES D-E	SHA1-DES3 D-E	SHA256-AES D-E	SHA256-AES256 D-E	SHA256-DES3 D-E	MD5-AES D-E	MD5-DES3 D-E
SW	214	27	-63	25	98	20	34	2
	1238	64	-2	18	64	13	87	5
	4096	36	2	34	38	5	28	-4
	32768	5	1	6	6	1	7	0
Tal	214	246	540	685	805	609	633	1156
	1238	48	-67	23	22	-60	-58	-104
	4096	2	2	17	16	-20	9	-22
	32768	0	1	1	0	1	1	1

Table 6.1: Decryption Minus Encryption (D-E) Throughput of AEAD Algorithms in Messages Per Second (m/s)

encryption was more powerful at software driver, and six cases when the *Talitos* was employed. No algorithm shows better performance only at encryption. Either the decryption dominates in all message size and crypto driver combinations, or sometimes performs the encryption and sometimes the decryption better. The most stable algorithm in both transformations is the MD5-DES3 when executed by the software driver, its encryption and decryption throughputs are almost similar. Overall, it can be said that the encryption transformation is, with some exceptions, "heavier" than the decryption transformation. Therefore only the encryption performance will be discussed further in this section.

6.2.2 Encryption Throughput

The Figure 6.3 illustrates the throughput obtained during encryption transformations by using the software or the *Talitos* driver with the same set of crypto algorithms. At each message size every algorithm is represented by two bars, the first one shows throughput achieved by the *Talitos*, and the underlying SEC, and the second one by the software driver. There are seven such pairs ordered by the throughput obtained with the software driver, from the lowest to the highest. In the legend of the figure the algorithm names are listed in the same order as they appear in the cluster centered at each message size on the x coordinate, i.e. the first algorithm name in the first column of the legend is the first algorithm on the left side of each cluster centered at each message size. The last bar in each cluster represents a value measured at the *KMedia*-Forwarding, it is the theoretical highest throughput which could be achieved by the cryptographic transformations.

As can be seen in Figure 6.3, the *Talitos* outperforms clearly the software implemented algorithms in all cases. However, at the 214 byte message achieve the algorithms SHA1-AES and MD5-AES in software a throughput which is very close to the throughput obtained by the *Talitos*, the difference is less than 0.1 Mbit/s. With larger messages higher throughputs are gained and with the *Talitos* at 32768 bytes long messages even the maximum throughput is hit.

These results show, that when the encryption transformation is performed by the *Talitos*, the achieved throughputs depend more on the message size than on the used crypto algorithm. The differences among the algorithms' throughputs at any message size are within 1.5 Mbit/s. On the other side, when the encryption is performed in software, the reached throughputs depend, except

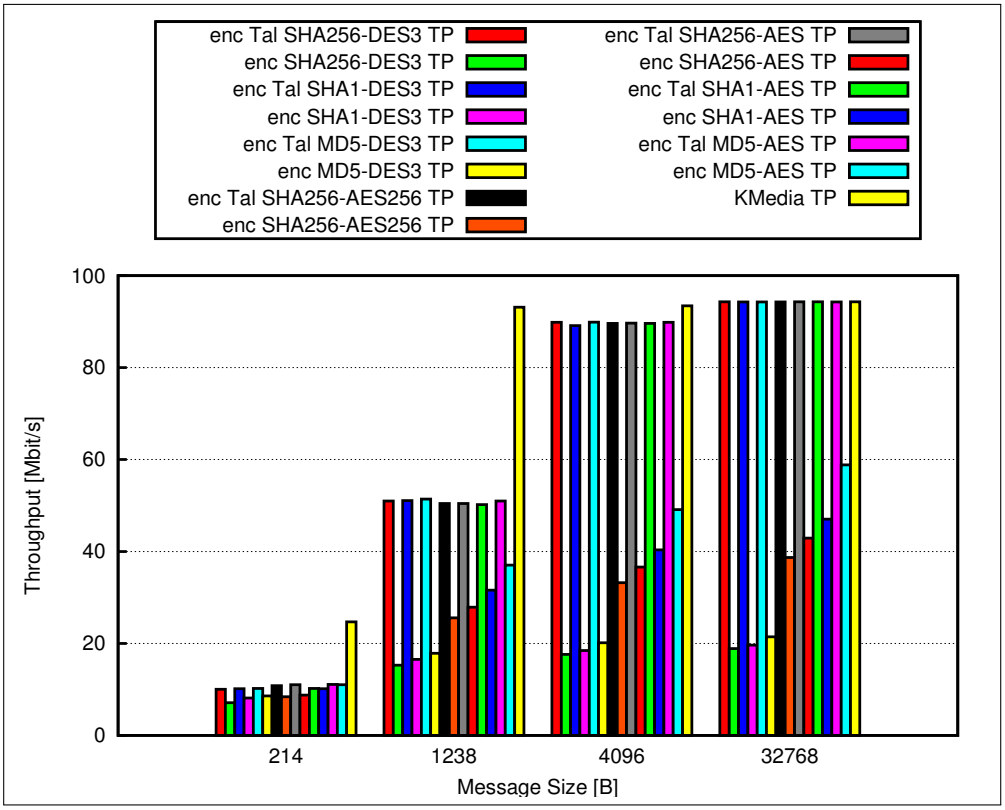

Figure 6.3: *Talitos* vs. SW Driver Encryption Throughput for Various AEAD Algorithms

of the message size, strongly on the employed algorithm. This illustrates what crypto algorithms perform well and which are quite heavy for the underlying system when no crypto accelerator is used.

6.2.3 CPU Load at Encryption

The following Figure 6.4 shows the processor load produced by the *Talitos* or software driver executing various crypto algorithms in order to encrypt messages of different sizes at the maximal message rate. As for the software driver, the CPU is fully utilized in every situation as the lines in the figure describe. The only line which descends from the 100% stands for the CPU load during *KMedia*-Forwarding and therefore it illustrates the processor usage when no cryptography takes place.

When messages smaller than, or equal to, 4096 bytes are encrypted by the *Talitos* driver, the CPU load values, depicted as bars in the figure, show utilization around 92%, although the maximal possible throughput is not reached. This signals that the *KMedia* crypto module spends some time waiting for the cryptographic accelerator, controlled by the *Talitos*, until it finishes the message transformation. The more computationally challenging the requested crypto algorithm for the accelerator is, the longer *KMedia* has to wait and the lower is the throughput and the resulting processor utilization. At the small message sizes are the differences among the *Talitos*' algorithms best to notice. Interesting is, that SHA1-AES performs worse, it encrypts less messages per second, than SHA256-AES and SHA256-AES256 which use longer secret keys. First, with

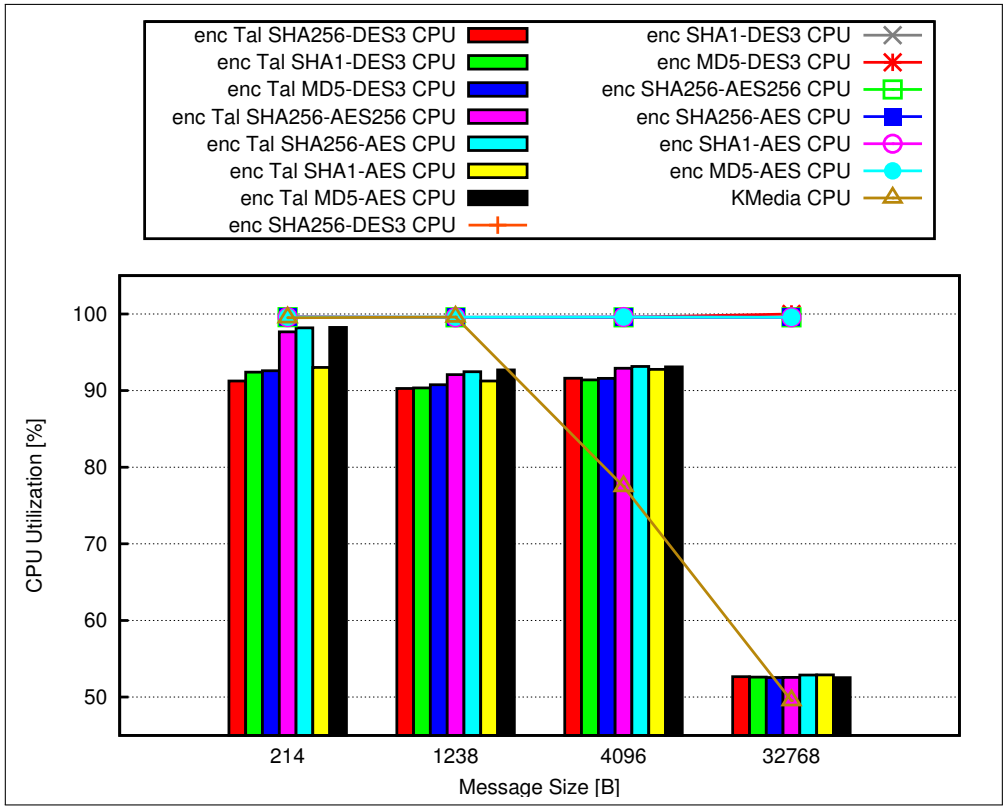

Figure 6.4: *Talitos* vs. SW Driver Encryption CPU Load for Various AEAD Algorithms

plaintext messages longer than 4096 bytes, when the throughput of the network test equipment saturates, begins the CPU load to drop below 90%.

6.2.4 SHA1-AES and SHA256-DES3 Encryption Throughput

To get a full characterization of the *KMedia*-DUT, two algorithms, SHA256-DES3 and SHA1-AES were tested on throughput during encryption at all prepared message sizes listed in Table I.1. As the previous Figure 6.3 illustrated, with a software driver achieves the SHA256-DES3 algorithm the worst throughput and the SHA1-AES makes the second best. The SHA1-AES was favored over the best one, the MD5-AES, for reasons noted already at the end of Section 5.5.1 on the page 61. As also the next Figure 6.5 depicts, the SHA1-AES implemented in the software performs much better than the SHA256-DES3.

All values used to draw Figure 6.5 can be looked up in Table J.5 placed on the page 154. Each cluster of bars centered at a particular message size on the x axis of the diagram represents the measured throughputs of the two tested crypto algorithms, first with the engaged software and then with the *Talitos* driver. The fifth bar in each cluster serves as a reference value and was obtained at the *KMedia*-Forwarding. From the lines in the figure the processor utilization at each message size, crypto algorithm and crypto driver, can be read out.

If the software driver is used, the CPU is fully loaded for all message sizes, the SHA1-AES throughput does not cross the 50 Mbit/s rate and the SHA256-DES3 stays under the 20 Mbit/s data rate. With the *Talitos* driver, the performance of the both crypto algorithms is nearly equal

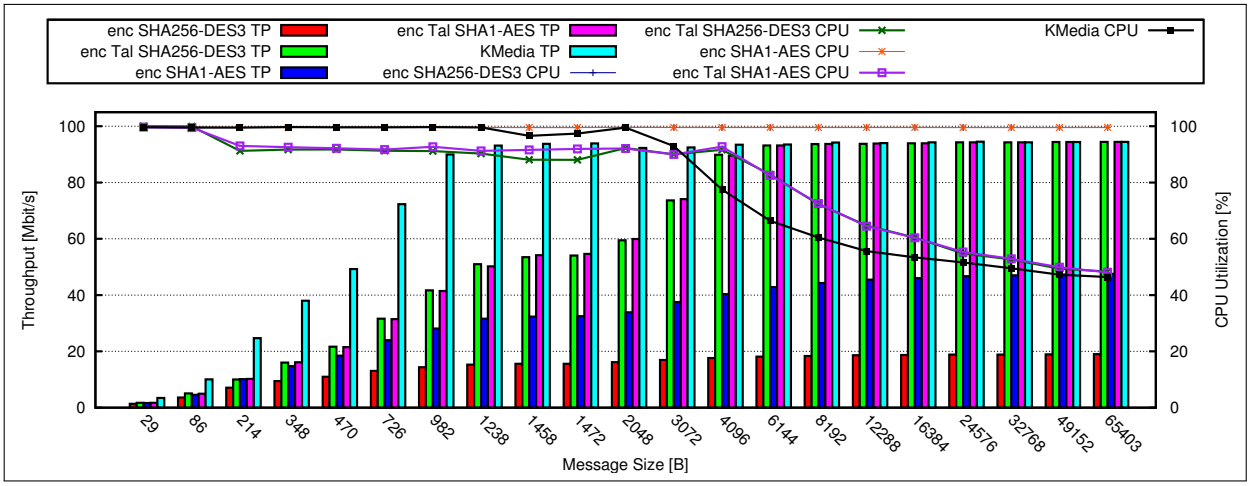

Figure 6.5: *Talitos* vs. SW Driver Encryption Throughput for SHA1-AES and SHA256-DES3

as their differences lie within 0.8 Mbit/s. It is also nice to see that, if messages longer than 4096 bytes are sent for encryption transformation, the throughput of the network test equipment is reached. Therefore, the CPU load begins to drop and converges to the utilization value obtained during the *KMedia*-Forwarding.

6.2.5 Talitos Performance Gain

The performance gain through the use of the *Talitos* and the underlying SEC depends on the message size and the engaged cryptographic algorithm. To compare the processor utilization of the *KMedia*-DUT when the software driver is used with the situation when the *Talitos* is employed, a *load factor* was computed[4]. It describes how many times more processor cycles per message does the software driver need for an encryption transformation compared to the *Talitos* driver. The Table 6.2 presents the results for message sizes 29–2048 bytes.

	Message Size in Bytes										
Algorithm	29	86	214	348	470	726	982	1238	1458	1472	2048
SHA1-AES	1.03	1.09	1.08	1.18	1.26	1.43	1.58	1.73	1.82	1.82	1.91
SHA256-AES	-	-	1.28	-	-	-	-	1.95	-	-	-
SHA256-DES3	1.24	1.39	1.54	1.84	2.13	2.64	3.17	3.68	3.87	3.92	3.97

Table 6.2: Software vs. *Talitos* Driver CPU Load Factor

According to Table 6.2, when messages with small sizes are sent for the SHA1-AES encryption, there is only a small performance gap between the *Talitos* and the software driver. For example, at 348 bytes long messages consumes the *KMedia*-DUT with the software driver only 1.18 times more CPU cycles per message than with the *Talitos*. On the other side, when the encryption of the 348 bytes long messages is performed by the SHA256-DES3 algorithm, the CPU cycles per message consumption is, compared to the *Talitos*, 1.84 times higher with the software driver.

[4] $\text{Load Factor} = \frac{(\frac{CPU\ Utilization\ at\ SW\ Driver[\%]}{Throughput\ at\ SW\ Driver[m/s]})}{(\frac{CPU\ Utilization\ at\ Talitos\ Driver[\%]}{Throughput\ at\ Talitos\ Driver[m/s]})}$

The SHA256-AES was added to the table to fill partly the transition from the SHA1-AES to the SHA256-DES3. It can be seen how a stronger hash function, SHA256-AES vs. SHA1-AES, and a different encryption algorithm, SHA256-AES vs. SHA256-DES3, influences the load factor. From the shown message sizes only the two, 214 and 1238 bytes, were measured on the SHA256-AES throughput.

6.2.6 Summary

The results obtained during the throughput measurements showed, that, although some exceptions exist, the encryption process places more load on the processor than the decryption, independently from the type of the crypto driver. If the algorithms are executed through the *Talitos* crypto driver, their throughputs at equal message sizes reach similar values and for messages smaller than 4096 bytes is the processor utilized on 92%. When the network saturates, the load starts to converge to the CPU load obtained during the *KMedia*-Forwarding. On the other side, when the algorithms execute through the software driver, their throughputs are very diverse, they do not achieve Ethernet link limit, and they utilize processor on more than 99%. The at the end calculated CPU load factor made apparent that for small message sizes the crypto accelerator performance gain can be very small if it executes a crypto algorithm which software implementation is effective and does not put much "per message" load on the processor. From this follows, the more heavy a crypto algorithm is in software, the higher gain can be achieved by the security engine.

6.3 Cryptographic Transformation Latency

This section presents results from latency measurements carried out during SHA1-AES encryption and decryption transformations of various message sizes using either the software or the *Talitos* cryptographic driver. The measured latency values are published in Table J.6 on the page 155. At the end of Section 5.5.1, on the page 61, the reasons are noted why the SHA1-AES was chosen for these measurements.

6.3.1 Socket-to-Socket Latency

The Figure 6.6 illustrates the duration a message of a particular size spent in the *KMedia* module during the SHA1-AES encryption. From the time point at which *KMedia* could read out the content of the message from the buffer of the UDP socket, to the time point at which the buffer was just about to be sent. It is called a "socket-to-socket" time and is expressed in microseconds. As can be seen in the figure, the duration consists of two parts: "KMedia delta" and "SHA1-AES tfm". The first part, the "KMedia delta", stands for the *KMedia* overhead caused by preparing the message for a transformation and a transmission, i.e. choosing the crypto algorithm, adding the padding bytes, adjusting the *KMedia* Header, reading out the *KMedia* Address Field. On the other side, the "SHA1-AES tfm" part, responsible for the major message delay, corresponds to the cryptographic transformation overhead caused by the *KMedia* Crypto API, the Linux Crypto API, by a respective crypto driver, the software or *Talitos*, and finally, also by the software or hardware implementation of the SHA1-AES algorithm.

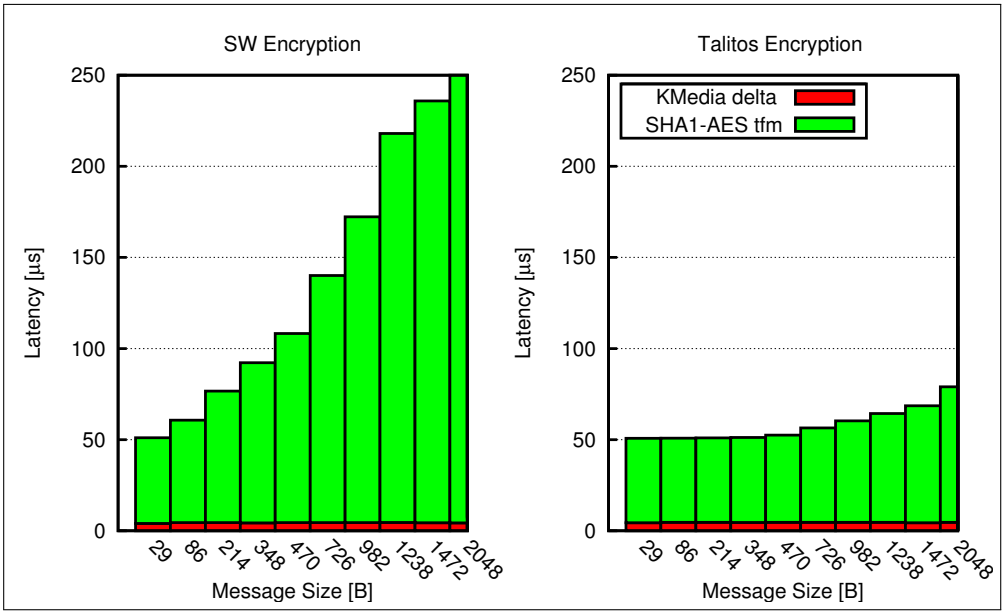

Figure 6.6: SHA1-AES Encryption Transformation (tfm) Time in Microseconds

Obviously, with the growing message size grows also the difference between the encryption latencies of the two implementations. For small message sizes, in which the overhead associated with the crypto APIs and the crypto driver dominates, shows the transformation with the *Talitos* driver quite equal performance for up to 470 bytes long messages. Therefore, overheads play in these situations more important role than the message sizes. First with longer messages begins the transformation delay to increase, but compared to the software solution is this increase very moderate. Also, when using the software implementation of the algorithm, the latencies grow from the lowest message lengths. Already a small change in the size, from 29 to 86 bytes, causes a prolongation of the encryption time.

The *KMedia* overhead stays basically constant. For the message sizes depicted in the figure and independently from the used crypto driver the overhead consumes 4.5 μs in average. For longer messages grows the latency up to 6 μs. This increase could be caused by the mechanism which suffixes padding bytes to the end of every message, since no other operation from the *KMedia* overhead depends on the message size.

6.3.2 Pure Cryptographic Transformation Latency

The following Figure 6.7 compares times consumed by the software implemented SHA1-AES algorithm with its hardware counterpart. In the first case the delays are caused by the *KMedia* Crypto API, Linux Crypto API along with the software driver and the software version of the algorithm and are labeled "SW dec tfm" and "SW enc tfm", depending on whether the times were measured during decryption or encryption transformations. The bars labeled "Tal dec tfm" and "Tal enc tfm" depict the time a message spent in the *KMedia* Crypto API, Linux Crypto API, *Talitos* driver and in the processor's SEC during the decryption or encryption transformation. The lines in the Figure illustrate the "*Talitos* Speedup", how much faster is the encryption or decryption with the *Talitos* compared to the software driver[5].

[5] $dec\ speedup = \frac{SW\ dec\ tfm[\mu s]}{Tal\ dec\ tfm[\mu s]}$, $enc\ speedup = \frac{SW\ enc\ tfm[\mu s]}{Tal\ enc\ tfm[\mu s]}$

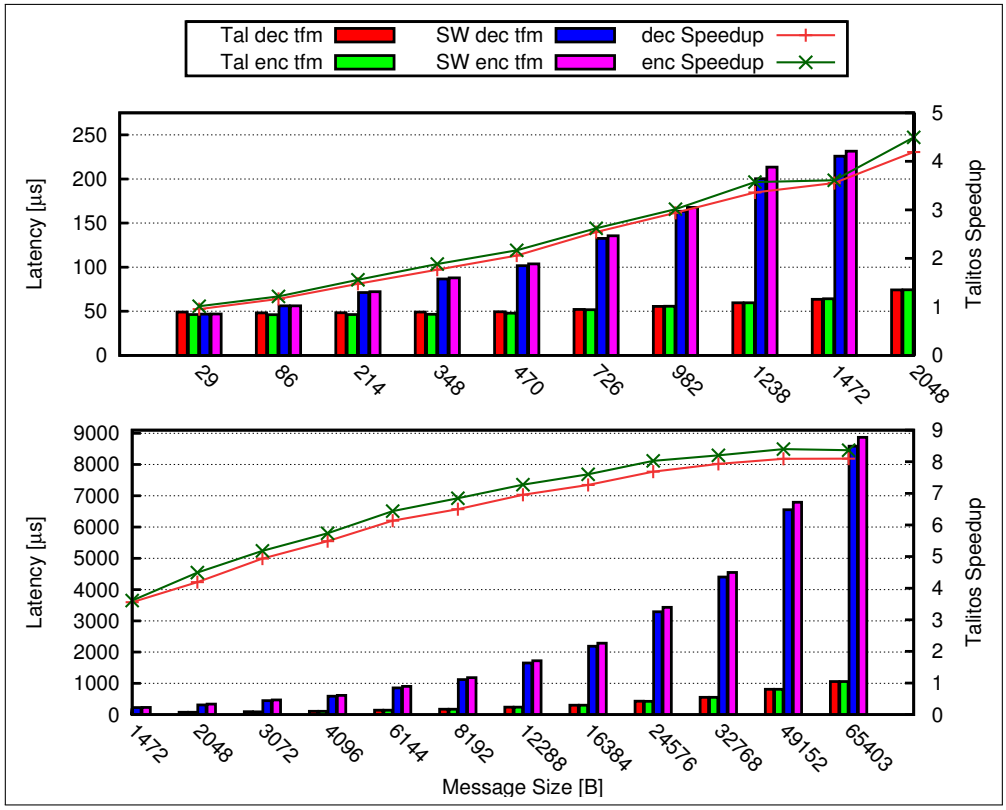

Figure 6.7: SHA1-AES Encryption/Decryption Latency in [μs]

As can be seen in Figure 6.7, the measured times confirm data obtained at the throughput tests and depicted in Table 6.1. The encryption is more CPU intensive, it consumes more time, than the decryption. However, this holds only for the software driver. If *Talitos* is employed, the differences are very small, less than 3 μs, and, moreover, the decryption is sometimes the slower one. Because the latencies caused by the crypto hardware are almost equal, and the software encryption is slower than the software decryption, is the *Talitos* gain higher for the encryption transformations, as also the lines in the figure illustrate.

Since Figure 6.7 compares delays caused solely by the operations involved in the crypto transformations, the resulting cryptographic transformation gain achieved by the crypto accelerator is, on the system level, the highest possible performance improvement reachable only if the additional processing latency, caused by the Linux kernel, would be zero. A "perfect" system having the same crypto overhead can come close to the shown speedup, but will not exceed it.

The Figure 6.8 on the page 79 depicts for various message sizes the just explained pure crypto gain, labeled "Crypto Gain", together with the *load factor*[6], the "TP CPU Gain". The third variable in the figure, the "CPU Gain", will be explained in Section 6.4.1 and can be ignored for now.

In other words, Figure 6.8 compares the speedup which could be brought to the system by using the crypto accelerator, the "Crypto Gain", with the achieved one, the "TP CPU Gain". As can be seen, with growing message size increases also the gap between these gains. Furthermore, from these results it is apparent that transforming messages on the application level of the TCP/IP

[6]The load factor was calculated in Section 6.2.5 on the page 74.

network stack, as *KMedia* does it, cannot efficiently utilize the performance offered by the crypto engine.

An uncommon datapoint is at the 29 byte message. At this point is the CPU gain slightly higher than the crypto transformation gain. As the high overheads connected with the processing of such small messages do not allow *Talitos* to have much better performance than the software driver, the performances are quite equal for both drivers and therefore also the gain varies around 1.

6.3.3 Summary

Encryption using *Talitos* shows balanced latency times for small messages which are more dependent on the overheads caused by the crypto APIs and the driver than on the actual message length. Delays caused by the *KMedia* processing of messages of up to 2048 bytes take 4.5 μs in average and are not dependent on the used crypto driver or the transformation operation, be it encryption or decryption. *Talitos* shows quite equal latencies at encryption and decryption what cannot be said about the software version with higher delays during encryption transformations. The last figure in this section illustrated that the SEC controlled by the *Talitos* has the power to bring higher advantage over the software solution than the *KMedia* is able to utilize. In other words, the *Talitos* with the SEC could encrypt more messages per second than the *KMedia* can deliver.

6.4 CPU Load at Various Message Rates

The results which are discussed in this section were obtained within the scope of the CPU Load measurements performed during the SHA1-AES encryption transformations of various message sizes at distinct data rates. The SHA1-AES crypto algorithm was chosen for the tests for reasons noted already at the end of Section 5.5.1 on the page 61. The results are written in Appendix J in the following three Tables: J.7, J.8 and J.9, beginning on the page 156.

6.4.1 Constant Message Size, Various Data Rates

This section shows the influence of the increasing data rate on the processor utilization at a constant message size. Eleven message sizes were considered in the tests and therefore eleven plots were generated. They all are placed inside Figure 6.9 on the page 81.

Each plot in the figure is labeled with a text defining the message size under test. The values on the x-axes denote the data rate and on the y-axes the CPU load. At each data rate consists a bar from three parts. The lower part, labeled as “CPU KM FWD”, depicts the processor load when no cryptography was applied and it was measured during the *KMedia*-Forwarding. The middle part, “CPU Tal”, illustrates the additional load induced by the *Talitos* encryption. Lastly, the “CPU SW” upper part stands for the additional load needed by the software encryption. In cases when the DUT did not have enough power to encrypt messages by software or hardware at the desired data rate, the relevant part, mostly the “CPU SW”, was left out. Except for the 32768 byte message, all plots show at the highest rate only the “CPU KM FWD” part, because no more resources were available for the SW or the *Talitos* encryption. The numbers on top of the bars denote the processor gain achieved when the *Talitos* was employed for the cryptographic

work instead of the software driver[7]. This gain can be expressed as: how many times more loaded is the processor, how many times more CPU cycles does it need, when the encryption is done by the software compared to the crypto accelerator. Naturally, this also means, how many times faster is the *Talitos* with the SEC over the software implemented algorithms running on the general purpose processor.

As can be read out from Figure 6.9, the gains within one message size are quite similar. This states clearly that the *Talitos* advantage over the software driver does not depend on the data rate but only on the message size. The gain values at each data rate, obtained with one message size, were used to compute an average *Talitos* gain. Afterwards, such average gains at each message size were compared with the *Talitos* speedup factors computed during the throughput and latency measurements. The resulting Figure 6.8 illustrates the results.

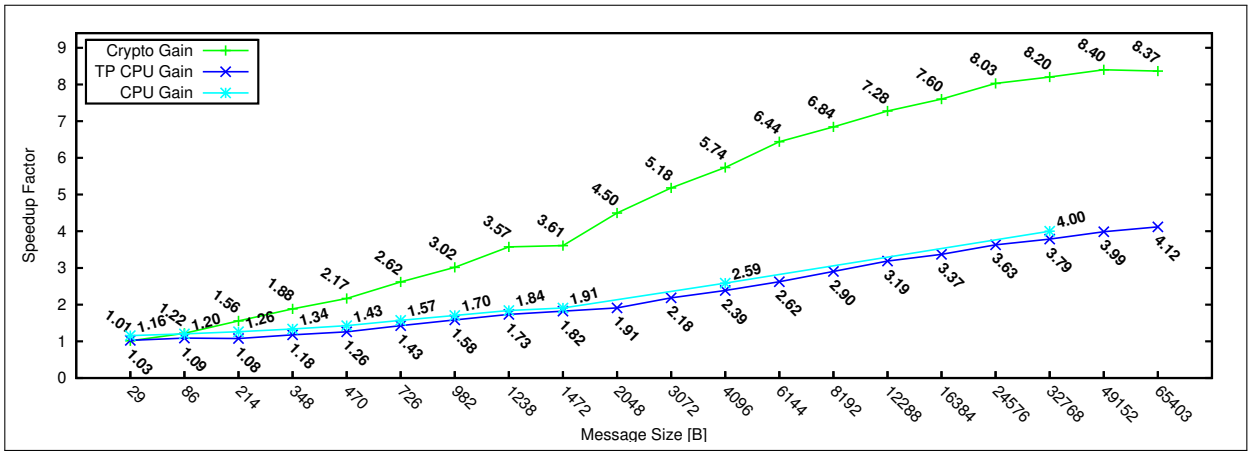

Figure 6.8: Speedup Calculated at Throughput (TP CPU Gain), Various Message Rates (CPU Gain) and Latency (Crypto Gain) Measurements with the SHA1-AES Encryption

The comparison in Figure 6.8 shows slightly higher values for the average gains than for the speedups calculated from the throughput tests. The difference could lie in the applied measurement methodology. While in the fist case the gain was computed from the CPU utilization at a constant data rate, in the throughput tests the gain must have been calculated from the achieved data rate and the corresponding CPU load. Furthermore, when the *Talitos* was employed in the throughput measurements, the DUT had to deal with a higher number of messages. Therefore, the DUT's resources needed for a non-cryptographic network data processing were more utilized during the hardware encryption than during the software encryption and this could reflect itself in the results. When comparing different processes or systems, the underlying test environment should not change, from this point of view, testing under equal data rates seems to be more appropriate than at throughput.

Another point which should be considered is that the *KMedia*-DUT shows at the throughput slightly another behavior than at the lower data rates. This can be clearly seen also in Figure 6.10 on the page 82. The figure displays the additional processor load when the software implemented SHA1-AES crypto algorithm is favored over its hardware version, i.e. the difference between the CPU load at software encryption and the CPU load at hardware encryption. With higher data rates increases, with approximately equally high steps, also this additional CPU load. However, when the tested data rate comes close to the achieved software encryption throughput, is this

[7] Tal Gain=$\frac{CPU\ SW[\%]}{CPU\ Tal[\%]}$

additional load obviously damped, in some cases even lower than the additional CPU load at the previous data rate. This behavior can be seen at the message sizes of 214, 348, 1238, 1472 and 4096 bytes.

This phenomenon reflects itself also in some calculated processor gains noted above the bars in the plots of Figure 6.9. For example, at the 214 byte message and the 9.6 Mbit/s data rate is the gain only 1.14, although this gain does not drop below 1.22 at the lower data rates. From this follows, that the DUT changes its behavior in the situations when its resources are almost exhausted. Some more tests would be needed in order to describe how the device deals with the transition from a state when enough resources are available, over the throughput state, when the device is on its maximum, to the overloaded state, in which already some messages must be dropped. Although it would be interesting to perform such tests, they were not done since they would exceed the scope of this thesis.

6.4.2 Constant Data Rate, Various Message Sizes

The Figure J.1 on the page 159 depicts the same data as Figure 6.9 but the other way round. It shows the influence of the message size on the CPU utilization at a constant data rate. Apparently, as the message size grows the CPU utilization drops.

6.4.3 Summary

This section has shown that the advantage of the security accelerator SEC controlled by the *Talitos* over the software implemented crypto algorithms controlled by the software driver depends on the message size and not on the data rate. With longer messages a better *Talitos* performance gain can be achieved. Also, the *KMedia*-DUT exhibits higher *Talitos* speedups at data rates smaller than the throughput[8].

[8] According to the throughput definition in Section 5.4 on the page 58.

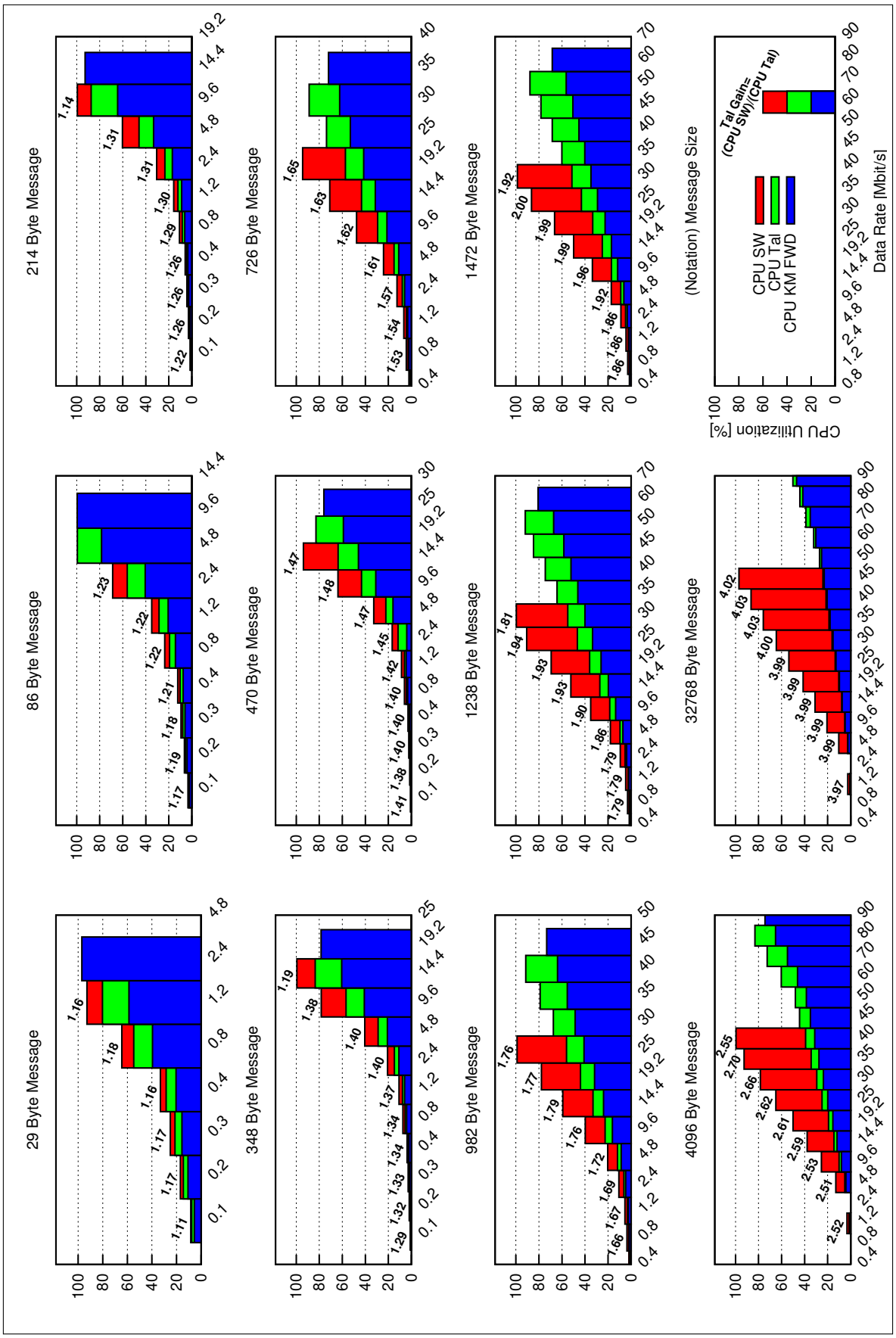

Figure 6.9: Data Rate Influence on the CPU Utilization at a Constant Message Size (Measured at *KMedia*-Forwarding and SHA1-AES Encryption with the SW- and *Talitos*-Driver)

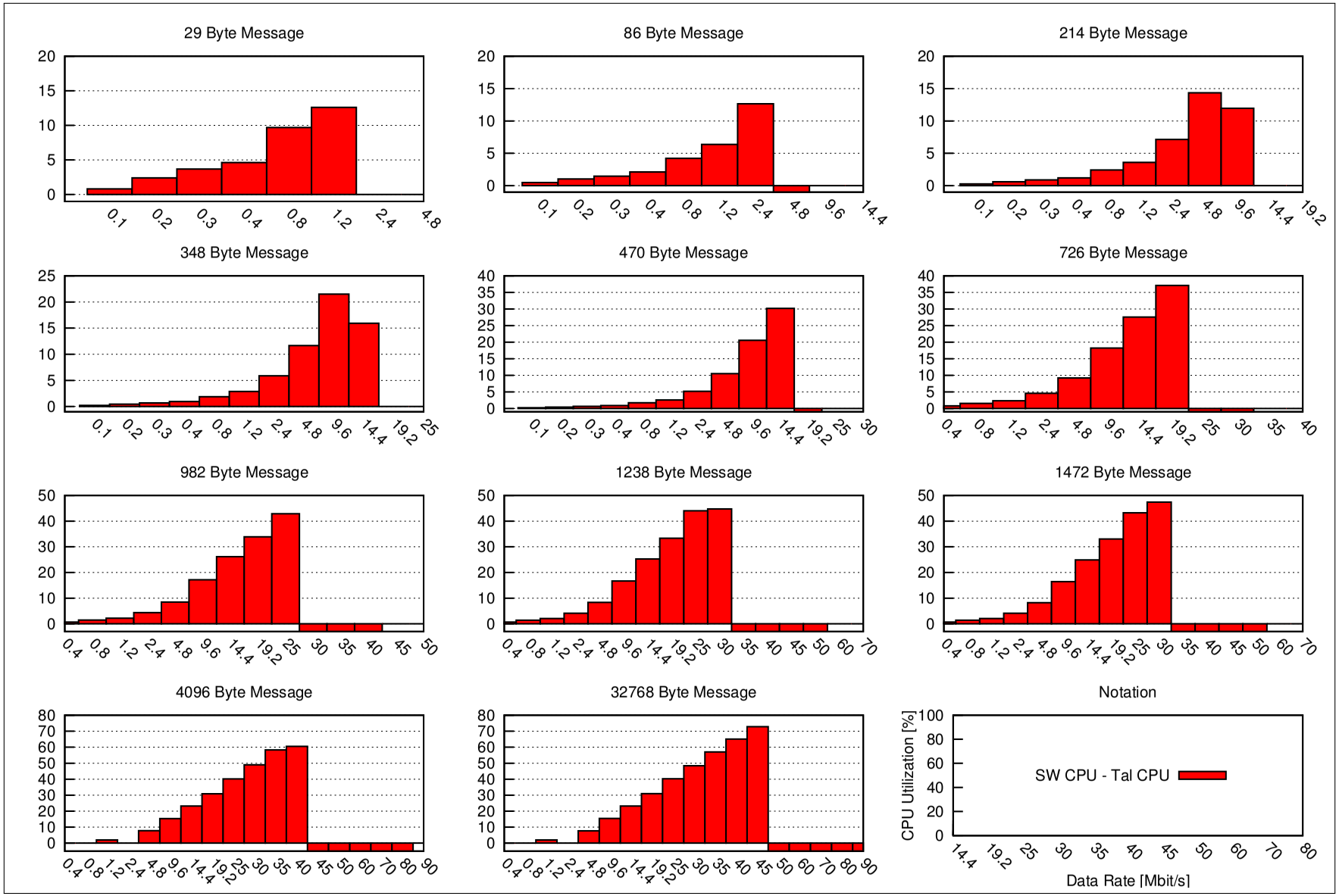

Figure 6.10: How Much More Loaded is the Processor when the SHA1-AES Encryption is Performed in Software Instead of Hardware (the Negative Values Signal Insufficient System Resources for the SW Encryption)

7 Conclusion

All work done in the scope of this thesis is summarized in this final chapter, from the *KMedia* kernel module description over to the measured performance gain achieved through the use of a cryptographic accelerator available on *Freescale*'s processors. Afterwards, some ideas for future work are given that would help to evaluate the advantage of a crypto hardware for a communication device with a Secure Real-Time Transport Protocol (SRTP) more precise. At the end also some methods are proposed that would help to improve the cryptographic performance of a communication system.

7.1 Summary

The focus of this master thesis was to analyze the advantage a crypto accelerator would bring to a communication system which secures real-time multimedia data transmitted over the Internet with the SRTP security protocol. The communication system was composed of a System on a Chip (SoC) processor from *Freescale* with an integrated Security Engine (SEC) and of a Linux operating system with a standard kernel distribution. However, instead of using an SRTP implementation, a network application, called *KMedia*, was specified and implemented. The application's purpose is to cryptographically transform received messages with algorithms offering authenticated encryption and transmit them afterwards to their next destination. This way, confidentiality, integrity and data origin authentication security services are provided to the network traffic. The cryptographic operations can be performed either by the crypto algorithms implemented in the Linux kernel or can be off-loaded to the *Freescale*'s SEC controlled by the Linux crypto driver *Talitos*, distributed with the Linux kernels since the 2.6.27 version. Since the Linux native crypto interface is utilized in order to access the *Talitos* and software crypto driver, the module should work also with other cryptographic drivers but only the two were tested in the scope of this thesis.

Although the security engine supports the algorithm that should be used by any SRTP implementation per default, according to the SRTP specification [BMN+04], the *Talitos* driver does not. Moreover, only the native Linux cryptographic interface, which does not have a user space support in the actual kernel distributions, can be used to access *Talitos* and therefore the driver's security services can be used only within the Linux kernel space. For these reasons, *KMedia* was implemented as a kernel module which uses crypto algorithms not default to SRTP.

However, the cryptographic algorithms which *Talitos* offers and SRTP uses, belong to the same family of authenticated encryption algorithms built according to the *Generic Composition* scheme.

These crypto algorithms provide confidentiality, integrity and data origin authentication security services. To ensure confidentiality, a standard symmetric block cipher, e.g. AES, executing in a specific operation mode and a standard cryptographic hash function, e.g. Secure Hash Algorithm (SHA), running also in a specific mode, is used. These algorithms are explained in detail in Section 2.3.

The difference between the cryptographic algorithms which *Talitos* offers and SRTP should use per default is only in the execution mode of the block cipher. While in *Talitos*, all offered algorithms use the execution mode called CBC which is inherently sequential, the SRTP's default algorithm uses the execution mode called CTR which allows for parallelism. Therefore, not much more better performance is expected from the CTR mode in software, since the employed processor has only one core, but in hardware, where the parallelism can take effect, the CTR mode should be significantly more efficient than the CBC mode. This implies also, that the advantage a crypto accelerator could bring to a system when running a block cipher in the CTR mode should be higher than with the CBC mode. At least in theory. Unfortunately, no works or results were found on the Internet that would measure and compare the performance of these two modes executing in software and also in hardware.

Since the *KMedia* kernel module executes a very simple kernel thread and exchanges messages with the UDP protocol through a socket interface, as the SRTP protocol also does, the results of the performance measurements obtained during software encryption should present the best possible performance a kernel space implemented SRTP protocol can achieve on the given single-core device. However, in case the accelerator is used for the cryptographic operations, it is assumed that the SRTP's default algorithm will perform better. Therefore, also the advantage of the crypto accelerator that is presented by this thesis should be higher under the SRTP's default algorithm.

Although the *KMedia* module's primary purpose was just to determine the cryptographic performance of the tested device, it can be used to secure any UDP traffic transmitted between two hosts on a network. In Appendix A.4.4 a use case with the *Netcat* network tool demonstrates how a bidirectional communication secured by *KMedia* could be established.

For the performance measurements of the module and device some aiding utilities must have been implemented or, if they were already available on the Internet, they must have been slightly modified. One lesson learned from preparing the tools for the tests is that calculating processor load according to the information from the Linux *proc* interface is inappropriate for network specific environments with high interrupt rates. A more accurate way is to use a mechanism based on "looper" processes to track the processor usage. This technique is briefly described in Section 5.1.6 and more detailed in Appendix A.6.2.

The measurements have shown that offloading cryptographic operations from a general purpose processor to a hardware crypto accelerator increases the overall system performance. With a security engine higher data rates can be achieved at lower processor utilization. However, as reported also by other works mentioned in Chapter 2.4, the system speedup caused by the crypto offloading is obvious only for larger messages.

As the tests executed in the scope of this master thesis with the HMAC-SHA1-CBC-AES algorithm show, with 32768 byte messages is the network device executing the *KMedia* module four times more efficient when the crypto accelerator is employed[1], with 214 byte messages only

[1]For example, at an equal data rate, the processor is four times less loaded when the crypto accelerator is employed than in case when the accelerator is not used.

1.26 times more. When encryption times were taken into consideration, thus excluding delays caused by the network stack, even higher speedups were calculated, from 1.22 at 86 bytes up to 8.40 at 49152 bytes long messages. It was also observed that during processor load measurements at various data rates slightly higher speedups could be obtained than at throughput measurements when all the device's resources were exhausted. The Figure 6.8 depicts this. Apparently, the device performs better in case it does not have to run on maximum. The hardware encryption comes already with 4096 byte messages close to the link limit of the testing network (100 Mbit/s) while in software not even the half of the bandwidth with the longest messages is reached.

The authors of the work [TRC08] implemented the IPsec security protocol in the Linux user space and achieved comparable, although slightly better, speedup values with offloading the HMAC-SHA1-CBC-DES3 algorithm as were obtained with *KMedia* with HMAC-SHA1-CBC-AES at various message sizes in Section 6.4, Figure 6.8. *KMedia* and the user-space IPsec use a socket in order to receive and transmit data packets, which seems to be the only component connecting these two measurements performed in different testing environments, since the IPsec runs on a mobile application processor chip with three processor cores, Montavista Linux OS, an unknown crypto interface and its highest data rate did not cross 8 Mbit/s.

A finding, not reported by any related work that was studied, is, that while in software different algorithms cause different throughputs at encrypting equal number of bytes, in the tested hardware this difference in performance was negligible. For example, as can be seen in Figure 6.5, HMAC-SHA1-CBC-AES achieves in software twice so high throughput than HMAC-SHA256-CBC-DES3 but in hardware their throughputs are almost similar. The Figure 6.3 denotes this even more clearly on all seven tested algorithms. From this follows, the more heavy a crypto algorithm is for software the higher gain yields the security engine. The above presented speedup values achieved with the *KMedia* module would have higher values if instead of the AES block cipher the heavier, more resource consuming, DES3 would be considered.

7.2 Outlook

Foremost, to get a better evaluation of the advantage a cryptographic accelerator could bring to a network device on which the SRTP security protocol should execute, it would be needed to perform measurements in a more SRTP related testing environment. This incorporates using the SRTP's default crypto algorithm and executing the testing application in the Linux user space. As it was already explained in the previous section, using the SRTP's default algorithm which allows for parallelism should bring higher encryption efficiency of the security engine compared to the sequential algorithms tested in the scope of this thesis. Therefore, also the crypto accelerator advantage over software encryption on a single-core processor should be higher. On the other side, accessing the cryptographic hardware from the user space would involve additional overheads nonexistent in the kernel space, e.g. the user-kernel context switches, which could cause a degradation of the crypto hardware advantage. Moreover, the overhead associated with the crypto hardware access from the user space could be so high that for some data packet sizes it would be more efficient to encrypt them by software implemented algorithms.

Since not only the need for securing network traffic is growing but also the amount of data that must be encrypted, it is important to study new strategies for supporting security protocols. Just adding a crypto accelerator is often not enough. Some optimizations are needed to obtain reasonable performance. The kernel module *KMedia* could perform better at lower message sizes

when instead of interrupts, used by the security engine to inform *KMedia* on a completion of a crypto operation, the polling mechanism would be utilized. However, this would block the processor at larger message sizes from doing some other work. So what are the other possibilities to improve the crypto performance of a network device?

For example, changing the block cipher operation mode can help a lot. As it was already explained, the block based ciphers as e.g. AES and DES3 can run in different modes. Except of the above mentioned CBC and CTR mode, more powerful ones in hardware and software are available and free of intellectual property. David A. McGrew and John Viega present in their scientific paper [MV04] the Galois/Counter Mode (GCM) and show that it is ideal for SRTP since it is very powerful for small message sizes, e.g. in software for 40 byte messages is the mode about 15 times more efficient than the mode tested by *KMedia* in this thesis. Moreover, the authors claim that with GCM speeds of 10 Gigabits per second and more can be achieved in hardware. Another, Carter-Wegman Counter (CWC) mode analyzed in the paper [KVW04] offers also high data rates at small message sizes. These, and some more others, were presented in Section 2.3.

In case the overhead associated with small packet encryption at a crypto accelerator should become so high that encrypting the packets in software on a general purpose processor would be more efficient, the following optimizations could be applied. While [MIK02] suggests a *hybrid approach*, to encrypt small packets in software and larger in hardware, the work [TRC08] implements an optimization based on this idea and calls it *adaptive crypto offloading*. The latter method decides according to the crypto algorithm and the size of an incoming packet whether hardware accelerator or directly the processor should be utilized. The works [CFP04, AAV06] go even further and develop a sophisticated packet scheduler which provides support for quality of service, load balancing over multiple crypto cards and processors and thus avoids, among others, inefficient small packet encryption in hardware. In the paper [FPC05] a scheduler bundles small packets with equal security requirements to form bigger packets before sending them to a crypto engine.

To conclude, the measurements performed during this thesis have shown that a cryptographic accelerator can help to improve system's performance significantly. However, the advantage depends strongly on the performance of the system components, as the TCP/IP network stack, operating system, security application using the cryptographic functionality and, naturally, also from the processor, network interface and memory subsystems. Therefore, in order to find the right accelerator and to define the speedup it could bring to a system, all these components must be considered. The best way is to perform some performance measurements directly on the system for which the accelerator should be used. It should be also not forgotten that the crypto algorithms can run in many various operation modes and it is up to the user to choose the right one, most suited, for a given system.

A Benchmarking Tools

In order to prove that a Device Under Test (DUT) works as expected and its performance is sufficient, an adequate testing environment is needed that can produce qualitative results. Such an environment consists of apparatus which, except of producing the test input data, also collects and evaluates the received data transmitted from the device. Unfortunately, it must be often built by the investigators, as there is no "off the shelf" equipment suitable for performing all required tests on any type of device.

The benchmarking tools described in this chapter were developed for, or adapted to, the performance measurement and verification of a network device with a cryptographic functionality. The DUT specification is presented in Section 5.2.1 and the kernel module responsible for cryptographic network data transformation, called *KMedia*, is described in Chapter 3.

Two of the presented tools, *Netperf* and *Socat*, are freely available applications which support a huge list of options and parameters. They can be easily downloaded from their home websites and only *Netperf* must have been slightly modified to conform to the test requirements. All other tools are small utilities, not larger than one file, developed in the scope of this thesis. Their source codes are listed in Appendix H.

The *KMedia Message Checker* together with the two tools for measuring the CPU utilization, *KMedia CPU Top* and *KMedia CPU Looper* can be used also on systems without the *KMedia* module. They might compile and run under different UNIX like operating systems, although they were tested only on a Linux platform, kernel 2.6.24 and higher. On the other hand, *KMedia Header Engine* and *KMedia Statistics Reader*, are "*KMedia*-bound", as they will not work on systems without the *KMedia* kernel module.

A.1 Netperf

This section is based on *Netperf*'s version 2.4.5 [17] and the manual [Jon07]. First a general info is given and afterwards the properties which determined this tool for being appropriate for the needed performance tests on *KMedia*-DUT are described. Unfortunately, the *Netperf*'s source code must have been modified little bit in order to provide an optimal testing environment, therefore also the performed changes are described and explained. Finally, a summary of steps executed during the installation and configuration process closes up the description of this tool.

Netperf is a network performance benchmark with the primary focus on "unidirectional data transfer and request/response performance using either TCP or UDP and the Berkeley Sockets

interface" [Jon07]. It works on a client-server model and consists of two executables the `netperf` and `netserver`. While `netserver` runs as a standalone daemon in the background and is started just once at the beginning of a test session[1], the `netperf` configures and initiates each test.

For *KMedia*-DUT testing the following three features of *Netperf* were important: it enables to regulate easily the data load sent to the DUT, it provides also several mechanisms for processor load measurement and, finally, it uses two types of connections, one for control data and one for test traffic. The next paragraphs describe these properties in more detail.

The *Netperf*'s command line interface offers two options, `-b <size>` and `-w <time>`, in order to define the number of sends per burst and the inter-burst time. This way one can transmit `b` frames back-to-back with a minimum inter-frame gap every `w` millisecond. In *KMedia*-DUT testing these options were used to scale the bandwidth the device should deal with. The burst contained just one message (`-b 1`) and only the inter-burst time was regulated.

Since it was not possible to measure the CPU utilization with *Netperf* in *KMedia*-DUT performance tests, because neither `netperf`, nor `netserver` was running on the device under test, a small tool with this purpose was developed. It is called *KMedia CPU Looper* and uses the *Netperf*'s source code to implement the "looper processes" mechanism. This mechanism together with the tool is described in Section A.6.2

A.1.1 Control vs. Test Data Connection

At the beginning of each test `netperf` sets up a control connection with `netserver` which runs on a remote host. This is always a TCP connection and is used to pass test configuration information and results to and from the `netserver` running on the remote system. After all set up data are exchanged, the connection for test traffic, the so called "data" connection, is opened using the sockets and protocols appropriate for the type of test to be run. Once the test completes the "data" connection is closed and the results are transmitted from `netserver` to `netperf` via the control connection. `netperf` compares the data received from `netserver` with its own data and displays the test results to the user. Since the control connection is not used while a test is in progress, it has no impact on the test results.

Important for the measurements of *KMedia*-DUT was that the two connections could be explicitly configured through the command line and that the control data could be sent to another destination as the test data. In other words, the host on which the `netserver` executes can differ from the host to which test data are sent.

Since *KMedia*-DUT was configured only to receive, transform and send UDP traffic, it was not possible to establish a TCP control connection that would go through the device. One solution was to turn down the control connection. *Netperf* supports it, but this way it is not possible to configure the second party, the `netserver`, nor to retrieve its results. The other, and better, solution was to send UDP test messages to *KMedia*-DUT which forwards them after transformation to the `netserver` and to establish the control connection directly with the `netserver` bypassing the device under test. Thus `netserver` gets the messages transformed by *KMedia* and sends the retrieved results to `netperf` using the separate control connection. The following use case illustrates the described configuration depicted also in Figure A.1.

[1] A *test session* consists of one or more tests.

A.1.2 *Netperf* Test Example

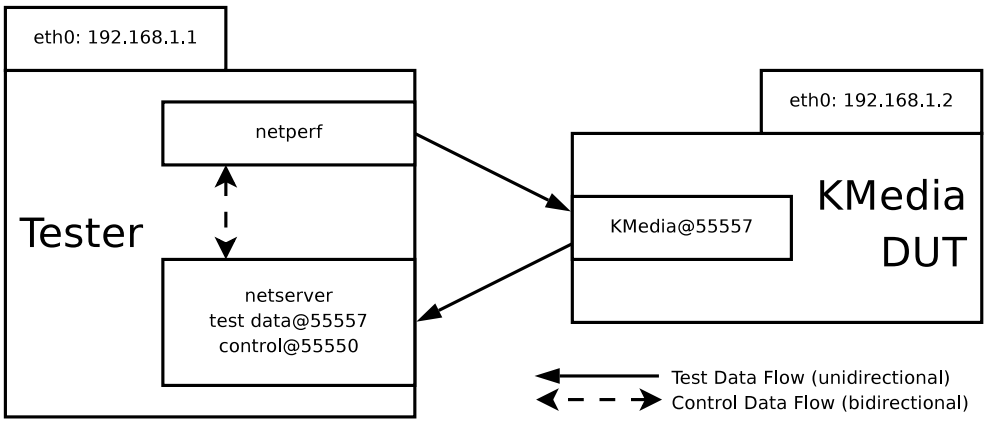

Figure A.1: *Netperf* Set Up

```
tester$ netserver -p 55550
tester$ netperf -t UDP_STREAM -b 1 -w 5m -l 10 -F KMedia.msg -H 192.168.1.1 \
               -p 55550 -- -P ,55557 -H 192.168.1.2 -m 470
```

The first line tells `netserver` the port number on which it should listen for a control connection. In the following command line the parameters for `netperf` are split into two parts with the "`--`" separator. The first part, called "Global Command Line", configures the control connection and specifies the test to be run. In this fall a control connection will be built with `netserver` listening on port `55550` and running on host `192.168.1.1`, thus the same host as `netperf` is running on. The test itself will consist of UDP messages filled with data from `KMedia.msg` file and transmitted by one with 5 millisecond interval for the duration of 10 seconds. The second part with test specific options configures the test data connection. According to the parameters above, 470 bytes long messages will be transmitted to the port number `55557` at host `192.168.1.2`. This information instructs the `netperf` at first to tell the `netserver` through the control connection that it should await test data on the port number `55557` and second to send the data to the given network address, `55557@192.168.1.1`.

Two more options worth of mentioning are "`-N`" and "`-W` <sizespec>" which should be set in the first "global" command line part. "`-N`" instructs *Netperf* not to use the control connection in which case *Netserver* does not take part in the measurements and *Netperf* just sends messages to a defined destination but does not care whether they really came there.

According to specification [Jon07]: "Unlike some benchmarks, netperf does not continuously send and receive from a single buffer. Instead it rotates through a ring of buffers." The number of buffers in the ring can be configured through the "<`sizespec`>" parameter of the "`-W`" option. Defaultly the ring consists of 32 buffers, as it is specified in the *Netperf*'s source code[2] although the specification says something else.

The buffers' lengths correspond to the message size and, as it is reported in the source code, the reason for the ring with buffers is to ensure that a buffer from which data were transmitted will not be used again before the data leave the device.

[2]Function `send_udp_stream()` in file `nettest_bsd.c`.

When a file is used to fill-in the sending buffers, the "`-F`" option, it is read out in a loop until all buffers are full. This implies that when the file has content just for one buffer its data will be copied from the beginning as many times as many buffers are in the ring. On the other hand, when the file is larger than one buffer, its content will be split into more buffers. This way it is possible to configure *Netperf* to send so many messages with various content in a row as many buffers the ring has. This feature was used when testing decryption performance of *KMedia*-DUT.

For a detailed explanation of these and much more other options supported by *Netperf* one should consult the *Netperf* manual [Jon07].

A.1.3 Modifications in the Source Code

Two features of the tool must have been adapted in order to support all requirements for the *KMedia*-DUT performance testing. First, the inter-burst time granularity was too high allowing only 10 ms large time steps on the used *Tester*[3] system. Second, the receiving part, the `netserver`, is informed from the transmitting part, the `netperf`, what size the test messages in transit have and thereupon messages received by `netserver` which do not have that proclaimed size are dropped. This is a problem because messages are resized when transformed by *KMedia*-DUT.

The performed source code modifications, described in the next paragraphs, were done just to adapt *Netperf* to the requirements of *KMedia*-DUT measurements and were not checked for any other test configurations. The patch with the applied changes can be found in Appendix A.7.

A.1.3.1 Scaling Down the Inter-Burst Time

On the command line one can define the value passed to the inter-burst time option to be expressed in seconds (`-w <time>s`), milliseconds (`-w <time>m`) or microseconds (`-w <time>u`). Defaultly, when no character is passed behind the `<time>` value, it represents milliseconds. Nontheless, *Netperf* running on *Tester* supported only 10ms precision, which was insufficient for *KMedia*-DUT tests.

Before coming to the needed changes in the source code two terms must be explained first, `HZ` and *jiffies*. `HZ` is an architecture dependent value defined by kernel developers[4] and represents number of timer interrupts per second [CRKH05]. The *Tester*'s platform used in *KMedia*-DUT measurements runned at 100 interrupts per second, as most other similar platforms also do. In the Linux kernel *jiffies* represent the state of an internal kernel counter. This 64 bits long value contains the number of occurred timer interrupts since last boot, because its value is initializes to 0 at boot time and is incremented with each timer interrupt.

Netperf uses an interval timer for generating periodical send events. Since this timer is set up with a library call `setitimer()`[5] which, according to its manual page[6], uses values represented in jiffies, the *Netperf*'s developers do not allow lower precisions than 1/HZ for an inter-burst time. From this follows that for the *Tester* system used for *KMedia*-DUT testing the smallest unit corresponds to the unsufficient 10ms[7].

[3] described in Section 5.2.2

[4] The `HZ` value is set in `linux/param.h` or a subplatform file included by it.

[5] Defined in file `kernel/itimer.c`.

[6] Release 2.77, 27 April 2006

[7] $\frac{1}{HZ} = \frac{1}{100\frac{[interrupts]}{[s]}} = 0.01\frac{[s]}{[interrupt]} = 10\frac{[ms]}{[interrupt]}$

The source code was modified in order to allow also values in microseconds to be passed to the `setitimer()` in the hope that if the function call cannot really handle precisions lower than 1/HZ it will raise an error message or convert the value to jiffies itself. Luckilly, the performed change did help a lot as it allowed to define the inter-burst time in the preferred microsecond units. The intervals in which *Netperf* was sending the messages were verified by the network tool *tcpdump*.

A.1.3.2 Any Size Message Reception

Since messages undergo a change not only in content but also in size when being transformed by *KMedia* it naturally implies that ingress messages differ from the egress ones which must be considered also by the employed testing tool. As *Netperf*'s part responsible for counting of received messages was developed to accept only messages having the same size as at the time of transmitting, the source code must have been adequately adapted for *KMedia*-DUT requirements. After applying the patch from Appendix A.7 `netserver` will accept and count messages of any size.

A.1.4 Summary

Example A.1 summarizes the steps needed to download, patch[8], configure and compile *Netperf* for the purpose of *KMedia*-DUT performance testing. The two last commands are executed without parameters and if they do not raise any error message then the installation was successful.

```
$ wget ftp://ftp.netperf.org/netperf/netperf-2.4.5.zip
$ unzip netperf-2.4.5.zip
$ cp netperf-2.4.5.patch netperf-2.4.5
$ cd netperf-2.4.5
$ patch -p1 -i netperf-2.4.5.patch
$ ./configure --enable-intervals=yes --enable-burst=yes
$ make   #eventually also $make install
$ ./src/netserver
$ ./src/netperf
```

Example A.1: Summary of Commands for *Netperf*'s Installation

A.2 Socat

The tool *Socat* described in this section was in version 1.7.1.1, downloaded from the webpage [14], where also its manual together with many useful examples can be found. After a brief description a reason for choosing it for *KMedia*-DUT measurements is given. A use case example together with a summary of steps needed for installing *Socat* from source code closes up this section.

In words of the author of the tool, *Socat* is a "*Netcat++*", an enhanced version of the tool which describes itself as a "TCP/IP swiss army knife". Both tools can establish network connections of different types and configurations and transfer data through them but *Socat* has, compared to

[8] The patch is in Appendix A.7.

Netcat, an extended design with new implementations. It can be used for many different purposes as it offers a large set of different types of data sinks and sources and also because many address options may be applied to the established data streams [Rie10].

For *KMedia*-DUT tests a tool was needed that could receive a UDP message of any size up to 65467 bytes and save it into a file. The 65467 bytes correspond to the maximal possible size of a UDP payload when considering the worst case IP and UDP header lengths, 60 and 8 bytes respectively[9]

Since *Netcat* uses maximal 8 KiB [18] large reads and writes to receive and transmit data across network connections it does not send a 65467 bytes long message at once. Instead the message is transmitted and received in 8 KiB data chunks. This is fine as long as *Netcat* is used for transmitting as well as for receiving but a problem occurs when another utility, such as *Netperf* A.1, transmits a message larger than 8 KiB which *Netcat* should receive. In this case *Netcat* reads just the first 8KiB and ignores the rest of the message. Therefore *Netcat* must have been replaced by *Socat* which enables to configure the size of reads and writes and therefore can be configured to receive also larger messages sent by other applications. The following example illustrates this.

```
$ socat -b 65467 udp4-listen:55552 file:outfile,create
```

This simple command instructs `socat`, the executable of the correspondent utility, to listen on the port number 55552 of the localhost for a UDP message using the IPv4 network protocol. The incoming message will be saved in a file named "outfile" which should be created in case it does not exist. And finally, the most important option, `-b` configures `socat` to use 65467 bytes long buffer for ingress messages. Any message larger than that size will be truncated. *Socat* offers large set of command line options in order to define data sinks and sources in many ways and they can be found in the tool's manual page, also available at [14].

The Linux commands in Example A.2 show the recommended way how to download and configure *Socat* from source code as it was used for *KMedia*-DUT testing. As one can see, it is a standard way of *Socat* installation with no special configuration options. Alternatively one could use the following aptitude command to install *Socat*: `$ apt-get install socat`.

```
$ wget http://www.dest-unreach.org/socat/download/socat-1.7.1.1.tar.gz
$ tar xzf socat-1.7.1.1.tar.gz
$ cd socat-1.7.1.1
$ ./configure
$ make  #eventually also $make install
```

Example A.2: Summary of Commands for *Socat*'s Installation

A.3 KMedia Message Checker

The purpose of this tool is, as can be derived from its name, to check the content of a message on correctness. At the beginning *KMedia Message Checker* reads in a file which is the exact copy of

[9] 65467 bytes of UDP payload + 60 bytes of IP header + 8 bytes of UDP header = 65535 bytes, which is the maximum IP datagram size.

an expected UDP message that should be checked. Afterwards, everytime the *KMedia Message Checker* receives a UDP message, it compares it with the read-in file. In case the message does not match the file it is dropped and the user is notified through terminal. If the message is of correct size and content it is forwarded either to a default network address defined in the source code as a macro, or to an address defined at the command line. The following text considers the version 1.0 of the tool from December 2009 and explains the reason why this tool was developed and what command line parameters it needs. Its source code is listed in Appendix H.2. Finally, a small use case together with a compilation step closes up this section.

A test equipment should always check whether messages coming from a tested device are of correct size and content [BM99, Section 10, "Verifying Received Frames"]. This way it should be ensured that only the messages forwarded by the DUT are considered in the test results and it is also verified whether the tested device transforms the network traffic correctly. *KMedia Message Checker* is a utility developed for the purpose of providing the verification on messages utilizing the UDP protocol.

A.3.1 Command Line

The tool's command line has the following syntax:

```
./kmedia_msg_checker listening_port file_name file_size(in bytes) \
                          [destination_IP_address destination_port]
```

Example A.3: *KMedia Message Checker*: Command Line Syntax

As for reasons of keeping the code simple no command line options were used, the parameters must be inserted always in the order as shown in Example A.3 above. The syntax of the command line is also displayed as a "usage" notification message in case the *KMedia Message Checker* is executed with none or with wrong number of parameters. The parameters are of following meaning:

listening_port The port number on which *KMedia Message Checker* should be listening for incoming UDP messages.

file_name The name of the file that contains the "right" message. Each ingress message will be compared in its whole length with the content of this file.

file_size(in bytes) The size in bytes of the "file_name" file corresponding to the size which all "to be checked" messages should have. Every received UDP message having different size as proclaimed by this parameter will be dropped.

The next two parameters are optional, but if used, they must be both defined. The default destination address is set to 127.0.0.1:55557 and can be changed in the source code.

destination_IP_address The IP address to which a message after a successful check should be sent.

destination_port Sets the port number of the destination address to which a message corresponding to the content of file "file_name" and having the size "file_size(in bytes)" will be sent.

A.3.2 Use Case And Output

The source code of the utility can be found in Appendix H.2 and the next Example A.4 shows the compilation process together with a small use case example.

```
$ gcc -Wall -o kmedia_msg_checker kmedia_msg_checker.c
$ ./kmedia_msg_checker 55552 982udpp.KM 982 192.168.1.1 55555
./kmedia_msg_checker: Listening on 55552, sending to 192.168.1.1:55555
```

Example A.4: *KMedia Message Checker*: Compilation and Use Case

After a standard compilation with the GNU C compiler the tool was instructed to listen for ingress messages on the port number 55552, to compare them with the content of the file "982udpp.KM" and to check whether they are 982 bytes long. The messages of correct size and content will be forwarded to the host 192.168.1.1 and port number 55555. The third line in Example H.2 displays the output of the *KMedia Message Checker* tool which informs the user that it was correctly configured and is ready to receive and check UDP messages.

A.4 KMedia Header Engine

This section describes the *KMedia Header Engine* tool[10], developed to simplify the usage of the *KMedia* cryptographic module for applications which can already exchange data through the UDP protocol. First, it will be explained how it works and how does it obtain all needed data. Afterwards, it is described what parameters are passed to the tool at command line which will be illustrated also by some examples. At the end a use case example with *Netcat* utility shows how a standard application can use *KMedia* security services through the *KMedia Header Engine*. The source code of this tool can be found in Appendix H.3.

A *KMedia* header together with an *Address Field* as described in the Section 3.2 is an inherent prefix of a *KMedia* message that should be transmitted to the *KMedia* kernel module. Without the information in this *extended header*, how the *KMedia* header together with the *Address Field* is called, *KMedia* module cannot transform the received UDP message and it will drop it. Therefore, the applications which want to use the security services offered by the *KMedia* module must prefix their UDP plaintext messages with the *extended header*. In order to prevent adding code to, or re-writing, applications that already use UDP messages for data transfer the applications can use the *KMedia Header Engine* tool.

The Figure A.2 illustrates the intended position of the *KMedia Header Engine* between an application, which wants to communicate with its peer through a channel secured by *KMedia*, and the *KMedia* module. As can be seen it uses two port numbers, the first is called *Application Port* and is for UDP messages that are transmitted between an application and the *KMedia Header Engine*. The second port, called *KMedia Port*, serves for *KMedia* messages received from or transmitted to the *KMedia* module. This way the utility knows to what messages an extended *KMedia* header should be added and from which messages the header should be removed.

The *Application Port* is set by default to 55550 and the *KMedia Port* to 55552 and cannot be defined neither through the configuration file nor the command line. Thus to change them, the source code must be modified and then recompiled.

[10] version 1.0 from December 2009

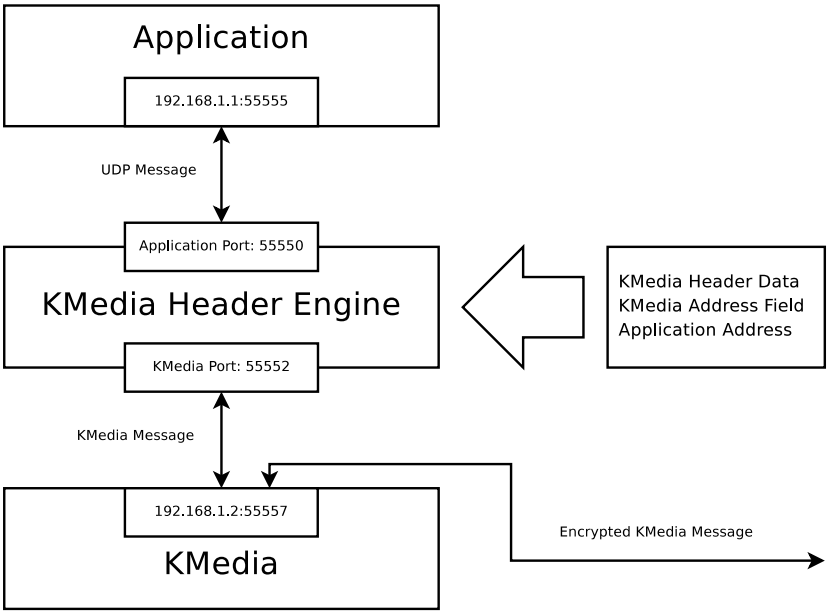

Figure A.2: *KMedia Header Engine*: Work Scheme

A.4.1 Configuration File

All data needed to build a proper extended *KMedia* header are read in at the tool's initialization time from a configuration file, displayed as a big arrow in Figure A.2. The content of the file consists of pairs of parameter names and its values. Each pair must be in a separate line and must be set just once, the order is not important. Comments begin with the "**#**" character and end with the end of the line. An example of a configuration file can be found in Appendix A.8.

The parameters from the input file have following meaning:

The first three parameters are used to compose *KMedia* header, which is defined in the Section 3.2.1.

msg_state Value assigned to this parameter is inserted into the "Message State" field of *KMedia* header. For simple forwarding, without using the security service of *KMedia*, value "3" should be used. To transmit messages through a secured communication channel provided by *KMedia* this value should be set to "1".

crypto_alg_id The "Crypto Algorithm ID" field of *KMedia* header will be filled with the value assigned to this parameter. To use a default algorithm for message encryption and authentication this value should be "0".

addr_id In most cases, this parameter should have the value "2" assigned, which will be written by *KMedia Header Engine* into the "Address ID" field of *KMedia* header. This value instructs *KMedia* what address from the *Address Field* should be used as a next destination for the transformed message.

All following IP addresses are in "numbers-and-dots" notation, e.g. "192.168.1.2" and the port numbers are in range from 1024 to 65535, inclusive. These network addresses, but the last one,

are used to compose the *KMedia Address Field* described in *KMedia Specification* on the page 27. The next text describes the purpose of the addresses when used in combination with the *KMedia Header Engine*.

source_host_ip, source_host_port Values assigned to these parameters define "Source Host IP Address" and "Source Host Port" in the *KMedia Address Field*. The two parameters together represent a network address of the *KMedia Header Engine* to which the configuration file is passed as it is the source of the *KMedia* message. It is not used by the header engine or the *KMedia* module, it has only an informative character. This way the final recipient knows who is the originator of the *KMedia* message and can use this address to send back a response.

source_crypto_ip, source_crypto_port Values assigned to these parameters define "Source Crypto Server IP Address" and "Source Crypto Server Port" in the *KMedia Address Field*. Since it is the network address of the first *KMedia* crypto server *KMedia Header Engine* sends all received messages, after prefixing them with the extended *KMedia* header, to this address. It could be paraphrased as the beginning of *KMedia* crypto channel.

destination_crypto_ip, destination_crypto_port Values assigned to these parameters define "Destination Crypto Server IP Address" and "Destination Crypto Server Port" in the *KMedia Address Field*. Used by the *KMedia* module to transmit transformed, mostly encrypted and authenticated, messages to its next destination. This address could be interpreted also as the end of *KMedia* crypto channel.

destination_host_ip, destination_host_port Values assigned to these parameters define the "Destination Host IP Address" and "Destination Host Port" in the *KMedia Address Field*. The *KMedia Header Engine* at this address strips off the extended *KMedia* header from the received *KMedia* messages, which were checked on authentication and decrypted at the previous address, and forwards the resulting UDP messages to application at "destination_appl_ip" and "destination_appl_port" described next.

destination_appl_ip, destination_appl_port IP and port number of a network address of an application which uses *KMedia* security services through the *KMedia Header Engine*. This address is used for internal purpose of *KMedia Header Engine* and is not transmitted by any message. After messages coming from the *KMedia* module get rid of *KMedia* data they are sent to this address.

The Figure A.3 illustrates on an example of two local networks A and B, which exchange messages utilizing the cryptographic services offered by *KMedia*, how the parameter names from configuration file passed to the *KMedia Header Engine* in the local network A are used. In the figure the term "local network" was taken for describing the place where *KMedia*, *KMedia Header Engine* and an *Application* interact, because, although they all could execute at one host system, each of them could be also running on a separate host in the network. From their point of view it does not matter, they just receive and transmit UDP messages from any network address, to any network address. But, since a security requirement is that unsecured messages are transmitted only in local networks the playground of the three utilities was accordingly bounded.

After the *KMedia Header Engine* in the local network A was appropriately configured with the input file described above, it opens its *Application* and *KMedia Port*. An application from the local

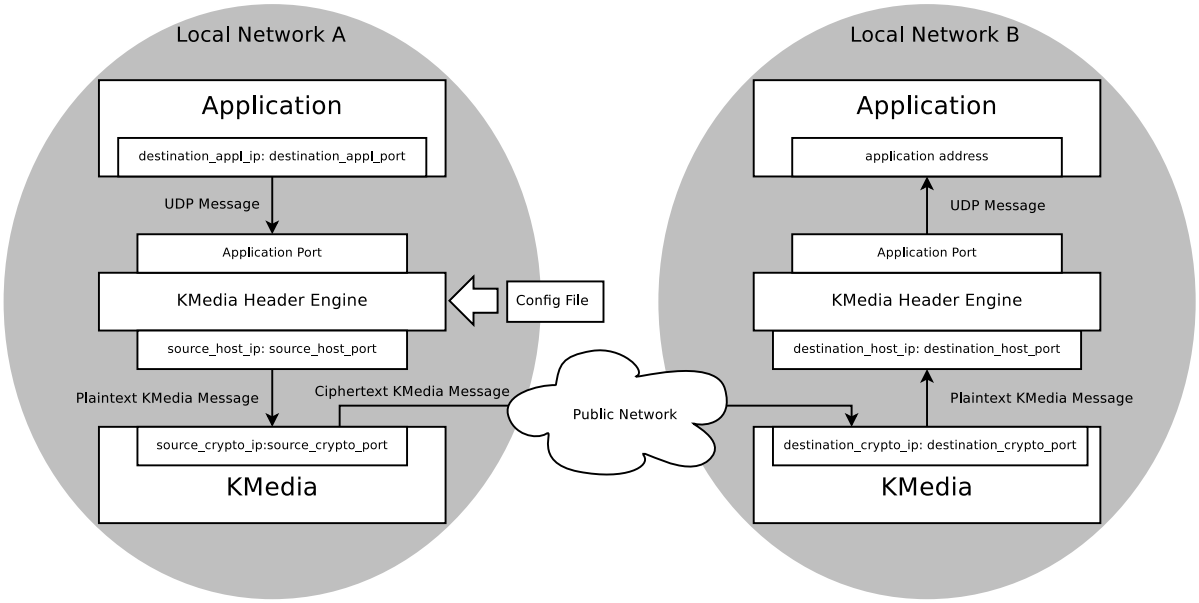

Figure A.3: *KMedia Header Engine*: The Role of Network Addresses in Configuration File

network *A* sends its UDP messages to the *Application Port* of the *KMedia Header Engine* where they are prefixed with extended *KMedia* header according to the configuration file and forwarded to the *KMedia* module at the address "source_crypto_ip: source_crypto_port". Afterwards the module transforms them appropriately using some cryptographic operations and transmits them to its peer in the network *B* at the address "destination_crypto_ip: destination_crypto_port". When messages are received at the other end of the cryptographic channel, they are transformed back to its original state and sent to the *KMedia Port*, "destination_host_port", of the *KMedia Header Engine* at "destination_host_ip". The engine removes their extended *KMedia* header and forwards them to the application in the network *B* as normal UDP messages. If *KMedia Header Engine* in local network *A* receives messages from *KMedia* at its *KMedia Port* (source_host_ip: source_host_port) it strips off the *KMedia* data and transmits them to the application at "destination_appl_ip: destination_appl_port" address.

This mechanism enables to an application a transparent use of *KMedia* cryptography, because it does not have to care about any additional headers. All it has to do is to send its UDP messages to the *KMedia Header Engine* instead of to the *KMedia* module, and, of course, it must configure the *KMedia Header Engine* appropriately.

A.4.2 Command Line

```
./kmedia_header_engine input_file [to_appl|check_msg]
```

Example A.5: *KMedia Header Engine*: Command Line Syntax

The Example A.5 shows the syntax of the tool's command line. In case the *KMedia Header Engine* is executed with none, or with a wrong number of parameters this information is also printed to the terminal as part of a "usage" information message. As can be seen, only the

"input_file", which is actually the configuration file, is mandatory, its meaning was described above in the "Configuration File" section. The two following optional arguments, of which only one can be used at a time, enable to change slightly the functionality of the engine.

If the tool is started with "to_appl" argument on the command line, it will not forward traffic to the *KMedia* module at "source_crypto_ip: source_crypto_port" network address. Instead of it, the messages equipped with a proper extended *KMedia* header will be sent to an application at the address defined by "destination_appl_ip: destination_appl_port". This way it is possible to generate easily, for example, *KMedia* test messages. An application sends a UDP message to the header engine which transforms it into *KMedia* message and forwards it to another application which then saves the *KMedia* message for a later use[11]. These pre-generated messages can be used by testing applications, such as *Netperf* A.1, to test *KMedia*-DUT performance.

It would be possible to generate *KMedia* messages "on-the-flow", *Netperf* could send UDP messages to the header engine which would in turn send them together with the extended header to the *KMedia* module, but as this implementation produces more load on the processor, it is better, especially when doing performance testing, to bypass the *KMedia Header Engine*. Therefore also in *KMedia*-DUT performance testing the *KMedia* messages were generated before the actual measurements began.

The argument "check_msg" causes that *KMedia* headers of all messages received from *KMedia* crypto module will be checked on correctness. Whether the message size proclaimed in the header corresponds to the real size of the received message and also whether the message state and address ID fields in the header are filled with the right values. If correct, the header together with the address field will be removed and the message forwarded. Otherwise, the message will be dropped and an appropriate message, describing the reason for dropping, written on the terminal. For performance reasons the checking is not done by default.

A.4.3 Use Case Examples

The first Example A.6 shows how *KMedia Header Engine* could be compiled with the GNU C compiler and how it is executed if the configuration file is called "example.kmhdr". The here mentioned config file can be found in Appendix A.8. After a successful initialization the utility prints out some information, which is described next, and is ready to do its work.

```
$ gcc -Wall -o kmedia_header_engine kmedia_header_engine.c
$ ./kmedia_header_engine example.kmhdr
KMEDIA port 55552
APPL port 55550
Child APPL_to_KMEDIA with ID 7528 was created.
Child KMEDIA_to_APPL with ID 7529 was created.
press ctrl-c to exit
```

Example A.6: *KMedia Header Engine*: Compilation and Execution

The output of the header engine notifies the user on which port number it expects messages from an application and on which from *KMedia*, called *Application* and *KMedia Port*. The next two

[11] As can be seen, *KMedia Header Engine* can receive messages from any application, but it will forward them only to the application at the address "destination_appl_ip: destination_appl_port".

lines reveal how the tool works. At initialization, the main process, the parent, creates two child processes. One, called "APPL_to_KMEDIA", has the task to add the extended *KMedia* header to messages received through the *Application Port* and transmit them to the *KMedia* module using the *KMedia Port*. The other one, called "KMEDIA_to_APPL" modifies messages the other way round. This child process receives messages through the *KMedia Port*, removes the unneeded header, and sends the modified messages to an application through the *Application Port*. Both processes run in an infinite loop which can be interrupted either by an error or explicitly by the user by pressing "ctrl-c". The parent process, which created the child processes, is responsible for resources allocation at the initialization time and for releasing them at the end, when the processes finish their execution.

The following Example A.7 shows the output of the header engine when executed with the "to_appl" argument. It indicates that only one child process is started, which will add the extended *KMedia* header to the ingress messages and transmit them to an application instead of to *KMedia*. The other process is not needed as in this case the header engine is used just to transform UDP messages into *KMedia* messages.

```
$ ./kmedia_header_engine example.kmhdr to_appl
APPL port 55550
Child APPL_to_KMEDIA with ID 8535 was created.
press ctrl-c to exit
```

Example A.7: *KMedia Header Engine*: Used Only for UDP- to *KMedia*- Message Transformation

A.4.4 Using Netcat with *KMedia Header Engine*

Since "*Netcat* is a simple Unix utility which reads and writes data across network connections, using TCP or UDP protocol." [18], it can be easily configured to send and receive UDP data to/from *KMedia Header Engine*. For this reason is *Netcat* an ideal application for demonstrating the full functionality of the header engine together with the *KMedia* kernel module.

For the following example a topology as depicted in Figure A.4 should be considered. As can be seen there are two hosts and in each reside all three programs, the *KMedia* kernel module, the *KMedia Header Engine* and *Netcat*. Also the port numbers, noted in, or on outside of, the box labeled with an utility name, equal in both hosts. Only their IP addresses differ and at one host, *Netcat* behaves as a client and on another one as a server.

To enable an easy exchange of data between *Netcat*'s client and server application using a UDP communication channel secured by *KMedia* the *KMedia Header Engine* on both hosts must get a proper configuration file. In this example it will be called "nc_server.kmhdr" for the server side and for client side "nc_client.kmhdr". In Appendix A.9 only "nc_server.kmhdr" is published, because the client's config file differs only in one line. The "destination_crypto_ip" in the client's config file contains the IP of the server and in the servers' config file it is the IP of the client. All other lines equal.

First of all, the *KMedia* module must be initialized and ready to receive messages on both hosts. This normally includes inserting the *KMedia* crypto interface, "`insmod kmedia_crypto_if.ko`", and then loading the *KMedia* module itself, "`insmod kmedia.ko`". Afterwards, on the server side

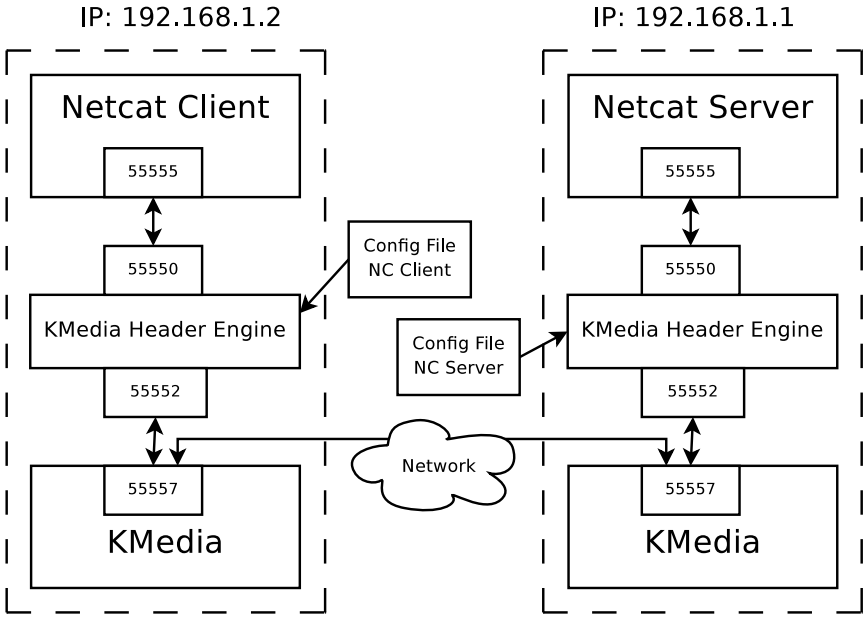

Figure A.4: Netcat Use Case Topology

```
$ ./kmedia_header_engine nc_server.kmhdr &
$ nc -lunvv -p 55555
listening on [any] 55555 ...
```

Example A.8: Using *Netcat* with *KMedia Header Engine*: The Server Side

the steps shown in Example A.8 are performed and, finally, the client is initialized with commands from Example A.9.

In Example A.8 the *KMedia Header Engine* configures itself with the "nc_server.kmhdr" configuration file and *Netcat* starts listening on the port number 55555 for UDP data. On the other side, in Example A.9 the header engine reads in the client's configuration file and *Netcat* opens a UDP connection to the port number 55550 at localhost. The additional requirement is that *Netcat* transmits and receives data through the port 55555.

```
$ ./kmedia_header_engine nc_client.kmhdr &
$ nc -unvv 127.0.0.1 55550 -p 55555
(UNKNOWN) [127.0.0.1] 55550 (?) open
```

Example A.9: Using *Netcat* with *KMedia Header Engine*: The Client Side

When the server and client started successfully they print out a message shown in the examples and a communication can begin by typing some characters to the client's side terminal. By pressing the "Enter" key on keyboard the characters will be transmitted to the server. After a first message is received by the server, it can respond by sending a message to the client. This way a bidirectional chat using *KMedia* encryption is opened. This is the simplest form of using *Netcat* in UDP mode, to see how data files or even live streams could be transmitted by *Netcat* see its manual pages [18] or the website [36].

A.5 KMedia Statistics Reader

This utility's task is to collect and evaluate some statistical information about the *KMedia* module. Namely the average times of *KMedia* message transformations and the message drop rates are periodically displayed on terminal by the *KMedia Statistics Reader*. First, it is described how and what data are gathered and how it helps the *KMedia*-DUT testing process. Afterwards this section looks at tool's configuration possibilities, be it through command line by passing appropriate values to the parameters or by defining some macros in source code. An use case example follows together with a compilation step and a description of the utility's output format. As some discrepancies can occur while gathering and evaluating data, the source of such behavior is described and an advice how to prevent it is given in the last part. The source code of this tool can be found in Appendix H.4.

"sysfs" is a virtual file system in Linux operating systems which provides an interface between userspace and devices and drivers from the kernel device model. Since this file system is also used by the *KMedia* kernel module a user can obtain some information about *KMedia* and/or configure the module through appropriate sysfs files. Particularly important for *KMedia Statistics Reader* are sysfs files which are created only if *KMedia* was compiled with the `TIMESTAMPING` macro which brings a time measuring functionality to the module. With the help of the dedicated files one can obtain information how much time in average *KMedia* spent with cryptographic transformation of a message. The sysfs files' content is described in *KMedia Specification* on the page 37 and it is recommended to consult it before continuing with this section.

For reasons of precise measurements the times captured in the sysfs "time measuring" files are represented in processor clock ticks. And for performance reasons, *KMedia* does not compute the average times, it only makes available the total sum of processor clock ticks spent with message transformations from the time of module initialization up to the time of reading. As the number of messages integrated in the time measurement is also available through a distinct sysfs file, a user can compute the average value itself. In order to convert a value from CPU clock ticks into seconds there is also a sysfs file which provides the information how many clock ticks occur per a second, but, in this case, the `/proc/cpuinfo` file could give a better value under the parameter "timebase", if available. Finally, the user should take care when reading the files, because they are not updated in an atomic manner.

From the paragraph above it follows that if one wants to get an average time *KMedia* spent with message transformation, first the appropriate sysfs files must be read-out, then the average must be computed and afterwards changed into seconds. This is exactly the task of *KMedia Statistics Reader*, which periodically performs the mentioned steps and displays the computed values.

Additionally, the tool also computes the number of messages dropped by the *KMedia* module and the number of messages dropped by the UDP socket. The difference is that in the second case the messages are discarded before they enter the *KMedia* module processing space, thus *KMedia* does not know about their existence. Such situations occur when system, in which *KMedia* executes, receives on the *KMedia* port more messages as it can handle. Therefore it signals that the system's resources, such as processor, message queues, but also *KMedia* itself, are exhausted. This is in contrast to the reason why *KMedia* fires the messages, as it happens mostly because of wrong format or content.

The number of *KMedia* dropped messages is easily obtained by subtracting the number of transmitted messages by *KMedia* from the number of received, both values are also provided by the

sysfs interface. The second value, the number of UDP dropped messages, is read out from a file called "/proc/net/udp" which is a part of a Linux pseudo file system, the "procfs". The file system serves primarily as an interface between userspace and tasks running on the system and therefore it provides a lot of useful information about the state of the system.

All the information provided by *KMedia Statistics Reader* helps in the *KMedia*-DUT testing to analyze the behavior of the device under various measurement configurations. While the timing data characterize performance of the *KMedia* module and of the individual cryptographic algorithms applied in message transformation, the information on dropped messages signals system overloading or incorrect *KMedia* message composition.

A.5.1 Command Line

The utility executed with wrong number, or wrong values of parameters displays the following "usage" message:

```
$ ./kmedia_stat help
usage: ./kmedia_stat [kmedia_port] [number of iterations] [refresh time in secs]
```

As can be seen, all parameters are optional, but if used, they must be given exactly in the given order, as no command line options are used. So, if one wants to define the "number of iterations" he must enter the "kmedia_port" first. Their meaning is as follows:

kmedia_port The port number on which *KMedia* waits for incoming *KMedia* messages. Defaultly it has the value 55557.

number of iterations How many times the relevant files should be read and the computed values displayed (10 times by default).

refresh time in secs Interval, in seconds, between two successive readings of files. It is set to 5 seconds if not otherwise defined at command line.

KMedia Statistics Reader gets the content of files, waits for "[refresh time in secs]" seconds, then reads the files again and computes afterwards the results according to the data obtained by the previous two readings. Finally the results are printed on the terminal. This happens "[number of iterations]" times or even less if the user kills the utility, by e.g. pressing "ctrl-c". As it was mentioned, the computation step together with printing out the results happens after the second reading and therefore the performance of the device under test during the interval when the data are gathered should be influenced only minimally.

A.5.2 Configuration With Macros

The tool can be configured, expect through the command line, also by three macros at the beginning of the source file, which are called `TIMEBASE`, `DROP_RATE` and `TIMESTAMPING`. To the first one a timebase, the number of CPU clock ticks per second, should be assigned. It can be read out from a *KMedia* sysfs file, or better, from "/proc/cpuinfo". Unfortunately, not every system

provides the timebase value in the "cpuinfo" file, but if it does, it should be preferred before making use of the sysfs file value. Naturally, the tool must be recompiled after each change.

Not all mentioned sysfs files must be read if the user does not need all the information at once. The different combinations of macros `DROP_RATE` and `TIMESTAMPING` allow to configure what data should be gathered and, as a consequence thereof, what results should be displayed. The tools' set up to the particular combinations is as follows:

`DROP_RATE` **defined** Independent of the macro `TIMESTAMPING`, i.e. it can be defined or undefined, the "UDP drop rate only" mode is enabled. This means that only the drop rate and the number of dropped messages by a UDP socket bound to the port number defined at the command line, or 55557 by default, will be obtained and displayed. This mode requires the files "/proc/net/udp" and "msgs_rx" from the statistics directory of the *KMedia* sysfs interface and can be also used with the *KMedia* module without time measuring functionality.

`DROP_RATE` **undefined,** `TIMESTAMPING` **undefined** In this mode the tool does the same as if `DROP_RATE` is defined, but it reads additionally also the "statistics/msgs_tx" *KMedia* sysfs file and provides except of the number of received messages also the information on number of dropped messages by *KMedia*. As in the previous mode, *KMedia* does not have to be compiled with enabled time measuring functionality.

`DROP_RATE` **undefined,** `TIMESTAMPING` **defined** This is the "full functionality" mode, besides the files from previous modes also sysfs files with timing information are read and evaluated. *KMedia* must provide such files and be therefore compiled with defined macro `TIMESTAMPING`. The complete string of displayed results in this mode will be discussed in the next subsection dedicated to the utility's use case example and to the output format.

A.5.3 Use Case And Output

KMedia Statistics Reader can be compiled the standard way with a GNU C compiler as the first line in Example A.10 illustrates. After a successful compilation the utility can be started with none or with some parameters as it was mentioned in the "Command Line" subsection. The example demonstrates the case when all three are employed.

```
$ gcc -Wall -o kmedia_stat kmedia_stat.c
$ ./kmedia_stat  55557 10 2
Using port number: 55557
Number of iterations: 10
Time interval: 2 seconds!
tm[s]  KMrx  KMdrp  sk2sk[ms] tfm[ms]   d[ms] scktdropp drprate
2.00   5564   0      0.077112 0.072577 0.004535 0      0.00
2.00   5565   0      0.077091 0.072587 0.004504 0      0.00
```

Example A.10: *KMedia Statistics Reader* Example

After a successful initialization the tool displays some control information, so the user can check whether the configuration is correct, and prints a string with abbreviations indicating the meaning

of the values that will appear below. Afterwards, it starts to collect data from the appropriate system files. The first results will come after the first interval times out, thus after 2 seconds in Example A.10.

If *KMedia Statistics Reader* was compiled with macros set to support "full functionality" mode the string with abbreviations will be as shown in the example. For the other two possible modes there will be less results printed but their names will be the same as in the "full functionality" mode. The labels have following meaning:

tm(s) Indicates the real time in seconds between two successive reads of system files used for computing the results. If the system is under heavy load this value can differ slightly from the desired interval.

KMrx Number of received messages by *KMedia*.

KMdrp Number of dropped messages by *KMedia*.

sk2sk(ms) Average time in milliseconds a message spends in *KMedia*. For each message the duration is measured between the time-point at which the message was received and the time-point at which it was transmitted by UDP socket used by *KMedia*, also termed as "socket-to-socket time".

tfm(ms) Average duration in milliseconds of a message cryptographic transformation.

d(ms) Average time per message in milliseconds needed solely for *KMedia* computation, also called "delta time": $d = sk2sk - tfm$.

scktdrop Number of dropped messages by a UDP socket used by *KMedia*.

drprate UDP drop rate in %.

A.5.4 Constraints

Care must be taken when gathering data from *KMedia* with this tool, because neither *KMedia* updates the data, nor the *KMedia Statistics Reader* gets the data in an atomic manner. Therefore discrepancies can, and will, occur. In the "full functionality" mode 6 files must be read which places additional load on the device under test, which could be already overwhelmed by the network test traffic. Such situation could cause not only that the utility's reading interval will be much higher than demanded, but also, that the six values got from the files would actually describe six different states of the system, and not, as ideally it would be, just one.

The preferred way is to execute tests within two successive readings of the same files, i.e. the interval should take longer than the actual test. Thus *KMedia Statistics Reader* reads needed system files first, then a test on *KMedia* is fully performed and afterwards the utility reads the files second time and can evaluate the obtained data and present results.

A.6 Tools for CPU Utilization Measurement

CPU utilization is one of the important indicators showing how suitable is to place certain network device into a desired environment. Particularly if the device should have enough resources also for some other tasks except networking, it might be useful to know how loaded is the processor at diverse data rates. Therefore tools implementing CPU utilization mechanisms should be also used in performance measurements of network devices.

Interestingly the two RFC documents on benchmarking methodology, the [BM99] and [KH09], do not even mention that measuring CPU utilization could be helpful. The reason therefore maybe is that the RFCs consider the devices as units whose processor resources are fully dedicated for network data forwarding and/or transformation and it is not expected that some other tasks will ever run on them. Nonetheless, measuring the CPU load was desired in the *KMedia*-DUT performance measurements, as described in the Section 5.4 and therefore also an appropriate tool must have been found or developed.

First *KMedia CPU Top*, influenced by the Linux program *Top*, was developed, but as the implemented CPU load measurement technique did not yield satisfactory results, another tool, *KMedia CPU Looper*, inspired by *Netperf* benchmark for measuring various aspects of networking performance, was written and then finally used in *KMedia*-DUT performance testing. The following two sections describe these tools in more detail, e.g. according to what data source they compute the CPU loads, how they work, what parameters can be passed to them at command line and in what situations they could have problems to return appropriate results.

A.6.1 KMedia CPU Top

The tool described in this section computes CPU utilization factor according to the data available in the "/proc/stat" file. Its source code is listed in Appendix H.5. After a brief description what *KMedia CPU Top* has in common with the Linux program *Top*, which in real time displays system summary information as well as some details of the tasks managed by the Linux kernel, it is explained why *Top* was not employed and instead time was invested into writing a new utility. The command line parameters, an example of how the tool can be executed and thereafter also its output is explained in the next paragraphs. Finally this section concludes with a remark on inappropriateness of using the mechanism implemented in this tool for measuring CPU utilization in situations with high interrupt rates, as it is the case when testing network devices.

As already the name indicates, the design of the utility was influenced by the standard Linux program called *Top*. The mechanism how *Top* computes the processor state, and what data it uses for that, is implemented also in *KMedia CPU Top*. It integrates the reading of the "/proc/stat" file and computing the fraction[12] the processor spent in idle mode.

For *KMedia*-DUT testing a simple utility was needed that would measure processor utilization. The Linux tool *Top* was a hot candidate, but it has some drawbacks that lower the precision of the obtained data. For example, the tool executes repeatedly operations in the following order: (1) get CPU state data, (2) compute CPU utilization and display the result, (3) wait for a defined time and continue afterwards again with (1). Such procedure integrates the load caused by the tool's computations into the result. This is correct, if one wants to know the total load all running tasks in the system together with *Top* place on the processor and it is not important if

[12](the amount of time which the processor spent in the idle mode) / (time slot of measurement)

the *Top*'s computations influence the execution of the other tasks. In cases when the processor's load should be measured just under influence of one or a specific set of tasks which should not be disturbed by another task running on the same processor the *Top*'s order of execution steps is inappropriate[13]. Particularly in performance testing when the whole processor's capacity is nearly completely, or already totally, exhausted just by the task under test, there is no place for additional computations. *Top* computes many system parameters and as it cannot be configured to calculate only the CPU utilization value its load on processor cannot be completely minimized.

For the reasons mentioned above it was decided to develop a tool that would execute the same steps as *Top* does but in different order and it would concentrate solely on the CPU usage value. In *KMedia CPU Top* the order is as follows: (1) get CPU state data, (2) wait for a defined time, (3) get CPU state data, (4) compute CPU utilization according to the data obtained in step (1) and (3) and display the result, continue afterwards with step (1). This way the load the utility puts on the processor influences the result only minimally.

Some lines from the source code of *Top* from the package "procps-3.2.8" [33] were used in the *KMedia CPU Top* code. Namely those for reading the appropriate data from the "/proc/stat" pseudo file and then those which compute the CPU idle value in percents, as *Top* does not provide the utilization factor. The desired CPU utilization can be easily obtained by the following equation: 100[%]-cpu_idle[%]. Although *Top* can display info about multiple processors in system, *KMedia CPU Top* is coded to show only the total utilization of all processors, independent from how many they reside in the system[14].

A.6.1.1 Command Line

The tool can be executed with two optional parameters as can be also seen in the following "usage" output, which is printed if wrong number or wrong values of parameters were passed.

```
./kmedia_cpu_top [number of iterations] [refresh_time in secs]
```

The first parameter, called "`number of iterations`", defines how many times the CPU utilization measurement should be repeated and the second, "`refresh_time in secs`", sets the interval in seconds between two successive reads of the CPU data according to which the utilization is computed. By default, *KMedia CPU Top* will 10 times display CPU load of the last 5 seconds.

A.6.1.2 Use Case and Output

The tool's compilation and execution is quite simple as the following Example A.11 illustrates.

First, a control message is displayed which can be used to check whether *KMedia CPU Top* was correctly configured and afterwards the results of measurements follow. The "delta" is a time in seconds and represents the real time measured between two consecutive CPU data reads. If the processor is under a heavy load this value can differ from the desired time interval. The CPU load measured during the "delta" time is labeled just as "CPU" and is expressed as a value between 0 and 100%.

[13] After all, one wants to know the load a particular task places on the processor and not the load of the task together with *Top*.

[14] *KMedia CPU Top* reads only the first line in "/proc/stat".

```
$ gcc -Wall -o kmedia_cpu_top kmedia_cpu_top.c
$ ./kmedia_cpu_top 20 2
./kmedia_cpu_top: Number of iterations: 20, Time interval: 2 seconds!
delta: 2.00   CPU: 15.08
delta: 2.00   CPU: 25.00
```

Example A.11: *KMedia CPU Top* Example

A.6.1.3 Constraints

During tests with *KMedia*-DUT testing it was observed that reading of "/proc/stat" to obtain CPU utilization data is not always the appropriate method. Particularly when the device under test has to deal with much network data the tool shows inadequate results. For example, when *KMedia*-DUT's task was just to forward incoming messages using NAT address and port forwarding, *KMedia CPU Top* was showing very low utilization as can be seen in Figure 6.1 on the page 68.

Also the manual to the network benchmarking tool *Netperf*, described briefly in the Section A.1, assesses this method based on reading the "/proc/stat" file as "thought but not known to be reasonably accurate"[Jon07, section: 3.1 CPU Utilization]. Moreover the *Netperf*'s author admits that the method could have "issues with time spent in processing interrupts", which could be critical because network performance tests produce high interrupt rates. Website [12, Methods of measuring] confirms this too.

For these reasons *KMedia CPU Top* was excluded from *KMedia*-DUT performance testing and a tool based on another, more appropriate, CPU utilization measurement mechanism was developed. It is called *KMedia CPU Looper* and is described in the next Section A.6.2.

A.6.2 KMedia CPU Looper

The mission of this tool is similar to *KMedia CPU Top*, it measures processor utilization. The difference is in the mechanism how it is performed. While *KMedia CPU Top* relies on data offerred by the Linux kernel system in "/proc/stat" file, *KMedia CPU Looper* trusts its own source of information and that is a "looper" process, from which also the tool's name is derived. The tool's source code is listed in Appendix H.6. This section explains briefly why *KMedia CPU Looper* was developed instead of using already available utilities and looks afterwards at the functioning of the tool's fundamental part, the "looper" process. The practical part will follow with explanation of command line arguments and of the two different run modes of the tool. Also the way how the results are displayed and what they mean will not absent. Some constraints one should consider when working with the tool will make the final point to the *KMedia CPU Looper*'s description.

The, under Linux administrators probably well known, program for monitoring some system's resources, called *Top* and its derivative *KMedia CPU Top* would be a simple solution to CPU load measuring, but as the first *KMedia* tests showed, their measuring mechanism is not suitable for situations when the processor has to deal with high interrupt rates, which is induced by high network data rates. This issue is described in Section A.6.1.3.

Luckilly, the manual [Jon07] of the network benchmark tool *Netperf*, described shortly in Section A.1, offers a list of many methods how the processor utilization could be suitably measured

also in network specific situations on diverse processor platforms. Moreover, *Netperf* implements them and thus a user can choose what method to use.

The *KMedia*-DUT served just for forwarding with transformation, it was not a final destination for any network message. A network testing tool on one host sent some testing packets to the device and then received them on another host, or on the same as from where they were sent. For this reason no benchmarking tool needed to be placed directly on the device under test. This imposed a limitation because when the employed benchmarking tool *Netperf* was not running on the device, its CPU load measuring functionalities could not be utilized. Therefore it was decided to make a small utility that would run on the device and measure the CPU utilization by utilizing mechanisms implemented in *Netperf*'s source code.

Since no special platform specific measuring methods were available for the *KMedia* device under test composed by the Linux OS running on PowerPC processor, the only last possibility offered by *Netperf*, except of that one based on reading the "/proc/sys"[15], was to use the mechanism based on the "looper" processes.

A.6.2.1 The "Looper" Process

Basically the "looper" process executes in tight little loops and counts as fast as it can. The processor utilization computation is based on the ratio of the value reckoned out when system was under load and a value generated when the system was believed to be in idle state. This implies that a calibration step, to let the "looper" count on the system when it is idle, must be performed before the actual measurements. But it is sufficient to do it at a particular system just once.

As Example A.12 shows, the job of the "looper" process is, except counting, also ensuring that it is, in the Linux terminology, very "nice" to all other processes running on the system. This corresponds to the lowest priority possible which is important because the "looper" process must not block the other ones. It should count only if no other task needs the processor.

```
void sit_and_spin()
{
        uint64_t my_counter;
        for (my_counter = 0; ; my_counter++) {
               nice(39);
        }
}
```

Example A.12: Looper Process

The "looper" process is started at initialization time of *KMedia CPU Looper* and is never interrupted in its counting. When a measurement takes place the state of the "looper" counter is simply noted at the beginning and then, after some time, at the end. The difference represents the reckoned out value during a given time slot.

In order to minimize the influence of the tool's computations in the measured CPU load *KMedia CPU Looper* evaluates the processor the following way: First (1) get the "looper" counter state,

[15] implemented already in *KMedia CPU Top*

then (2) wait for a certain time, afterwards (3) read the "looper" counter state again and (4) compute CPU utilization according to the counter values obtained from step (1) and (3), finally display the result and continue with step (1).

In *Netperf* a "looper" process is created for, and executes on, each known CPU in the system. This way a utilization in a multiprocessor environment is computed. As *KMedia*-DUT is just a one processor system, the *KMedia CPU Looper* became code from *Netperf* but without the support of multiple processors.

A.6.2.2 Command Line

KMedia CPU Looper can be started with the optional parameters as shown in the following "usage" informational message in Example A.13. It is displayed in case wrong values or invalid number of parameters are passed to the tool.

```
./kmedia_cpu_looper [debug_mode] [local_cpu_rate] \
                      [number of iterations] [time_interval in secs]
```

Example A.13: *KMedia CPU Looper*: Usage

As no command line options are used the parameters must be defined in the order as the example shows. The meaning of the parameters is described next:

debug_mode The verbosity level, i.e. how much information should be printed on the terminal during execution. Three states defined by values: "0", "1" and "greater than 1"[16] are possible. With zero level representing the essential data and "greater than 1" being the most talkative mode.

local_cpu_rate The calibration value obtained by *KMedia CPU Looper* when processor was in idle state. It should be the maximal possible rate at which the "looper" process can count on the dedicated processor. When this parameter is not defined *KMedia CPU Looper* will switch into the calibration mode and returns after a successful execution the CPU maximal rate.

number of iterations How many times the CPU load should be evaluated and displayed. Defaultly the processor will underlie three measurements.

time_interval in secs The time slot in seconds a measurement takes place, i.e. the time between two successive reads of the "looper" process' counter, according to which the resulting processor utilization value is computed. Defaultly this time is set to 10 seconds.

A.6.2.3 Use Case And Output

The Example A.14 shows how the tool could be compiled with the GNU C compiler and afterwards how the calibration is started which returns after about 70 seconds the maximal processor rate needed for following CPU load evaluations. This must be done with processor in idle state.

[16] i.e.: 2, 3, 4...

```
$ gcc -Wall -o kmedia_cpu_looper kmedia_cpu_looper.c
$ ./kmedia_cpu_looper
Starting calibration, it will take about 40 seconds!
starting in 30 seconds!
CPU max rate: 738030
```

Example A.14: *KMedia CPU Looper*: Compilation And Calibration

The output of the tool in Example A.14 should be understood as, "in 30 seconds a calibration process will start that will need 40 seconds", therefore the 70 seconds in total. According to the comment in the part of *Netperf*'s source code where these start up seconds are defined "we need to have the looper processes settled-in before we do anything with them". No special reason is given why 30 seconds were chosen and not any other value.

To provide CPU load measurements *KMedia CPU Looper* is started with the CPU rate returned by calibration process. The next Example A.15 illustrates this case when also "`number of iterations`" and "`time_interval in secs`" are defined.

```
$  ./kmedia_cpu_looper 0 738030 10 5
Starting measurements!
Local CPU rate: 738030.00 per second
Number of iterations: 10
Time interval: 5 seconds
starting in 30 seconds!
delta: 5.00   CPU: 10.68
delta: 5.00   CPU: 10.00
```

Example A.15: *KMedia CPU Looper*: Processor Load Measurement

After some informative messages, showing how the tool was configured, it is also, as at calibration process, stated that the measurements will start in 30 seconds. The reason therefore is the same as mentioned above when discussing calibration and it is done only once at the beginning. Finally, after a successful initialization and "settling in" of the "looper" process, the tool begins with measurements and displays the obtained results. The "delta" expresses in seconds the real time the measurement took place, which, in cases when processor is under a heavy load, can differ from the desired value potentially defined through "`time_interval in secs`" parameter. The "CPU" label in the output displays the processor load in percents as measured in the previous time slot of "delta" time duration.

A.6.2.4 Constraints

Since the "looper" process cannot squeeze 100% processing power from the processor during calibration time slot, as there will always be a task running in background doing some "housekeeping" work, it cannot get an ultimate maximal CPU rate. Therefore it can happen sometimes when the system is under very light load that the processor has less work to do at measurement time than during calibration, in which cases the computed processor utilization will be a negative value. However, if all possible processes running on the system were turned down before calibration, these negative values, if they happen, should be very small.

When comparing the results of *KMedia CPU Looper* with results of other CPU utilization measurement tools one should consider that the "looper" basically measures just the amount of free computing power available to tasks which are not running during the duration of a measurement. The value computed by the tool should be better described as representing the amount of used resources that were available at calibration, because the ratio of the obtained free processor resources when the processor was under load and of the free resources encountered when the processor was believed to be idle is used to compute the resulting utilization. It can differ from the real total CPU utilization in which also the tasks doing "housekeeping" work are considered.

A.7 *Netperf*'s Patch File

This Appendix contains a patch, called `netperf-2.4.5.patch`, for *Netperf*'s 2.4.5 discussed in Section A.1. Maybe the patch works also with some other versions of *Netperf*, but it was not tested. It should be applied from inside of the *Netperf*'s directory as the following commands illustrate:

```
$ cd netperf-2.4.5
$ patch -p1 -i netperf-2.4.5.patch
```

```
1  diff -ur netperf-2.4.5/src/netlib.c netperf-2.4.5-kmedia/src/netlib.c
2  --- netperf-2.4.5/src/netlib.c  2009-05-28 00:27:34.000000000 +0200
3  +++ netperf-2.4.5-kmedia/src/netlib.c   2010-01-31 19:26:22.000000000 +0100
4  @@ -1094,19 +1094,19 @@
5     struct itimerval old_interval;
6
7     /* if -DWANT_INTERVALS was used, we will use the ticking of the itimer to */
8  -  /* tell us when the test is over. while the user will be specifying */
9  -  /* some number of milliseconds, we know that the interval timer is */
10 -  /* really in units of 1/HZ. so, to prevent the test from running */
11 -  /* "long" it would be necessary to keep this in mind when calculating */
12 -  /* the number of itimer events */
13 +  /* tell us when the test is over. */
14 +
15 +
16 +
17 +
18 +
19 +  ticks_per_itvl = interval_wate;
20
21 -  ticks_per_itvl = ((interval_wate * sysconf(_SC_CLK_TCK) * 1000) /
22 -                    1000000);
23
24    if (ticks_per_itvl == 0) ticks_per_itvl = 1;
25
26    /* how many usecs in each interval? */
27 -  usec_per_itvl = ticks_per_itvl * (1000000 / sysconf(_SC_CLK_TCK));
28 +  usec_per_itvl = ticks_per_itvl;
29
30    /* how many times will the timer pop before the test is over? */
31    if (test_time > 0) {
32 diff -ur netperf-2.4.5/src/netsh.c netperf-2.4.5-kmedia/src/netsh.c
33 --- netperf-2.4.5/src/netsh.c   2008-10-25 01:38:57.000000000 +0200
34 +++ netperf-2.4.5-kmedia/src/netsh.c    2010-01-31 19:28:20.000000000 +0100
```

```
35 @@ -861,7 +861,7 @@
36         if (arg1[0]) {
37  #ifdef WANT_INTERVALS
38          interval_usecs = convert_timespec(arg1);
39 -        interval_wate  = interval_usecs / 1000;
40 +        interval_wate = interval_usecs;
41  #else
42          fprintf(where,
43                  "Packet rate control is not compiled in.\n");
44  diff -ur netperf-2.4.5/src/nettest_bsd.c netperf-2.4.5-kmedia/src/nettest_bsd.c
45 --- netperf-2.4.5/src/nettest_bsd.c      2009-06-04 02:31:15.000000000 +0200
46 +++ netperf-2.4.5-kmedia/src/nettest_bsd.c       2010-01-31 19:22:14.000000000 +0100
47 @@ -7112,7 +7112,8 @@
48              send_response();
49              exit(1);
50          }
51 -        break;
52 +        if (len <= 0)
53 +          break;
54        }
55        messages_recvd++;
56        recv_ring = recv_ring->next;
57  diff -ur netperf-2.4.5/src/nettest_unix.c netperf-2.4.5-kmedia/src/nettest_unix.c
```

A.8 *KMedia Header Engine* Configuration File Example

All parameters used in the shown configuration file for *KMedia Header Engine* are discussed in Section A.4.1.

```
1  # config for kmedia header engine
2  # filename: example.kmhdr
3
4  # message states:
5  # plaintext : 1
6  # forward   : 3
7  msg_state 1
8
9  # crypto algorithm ID
10 # 0 stands for default, chosen by KMEDIA
11 crypto_alg_id 0
12
13 # address ID
14 # Identifies the first address in address field
15 # to read out by KMEDIA.
16 addr_id 2
17
18 #addr_id 0
19 source_host_ip   192.168.1.1
20 source_host_port 55552
21
22 #addr_id 1
23 source_crypto_ip   192.168.1.1
24 source_crypto_port 55557
25
26 #addr_id 2
27 destination_crypto_ip 192.168.1.2
28 destination_crypto_port 55557
29
30 #addr_id 3
31 destination_host_ip   10.16.10.12
32 destination_host_port 55552
33
34 # address of application using kmedia header engine
35 destination_appl_ip 192.168.1.1
36 destination_appl_port 55555
```

A.9 *KMedia Header Engine* Configuration File for Netcat Use Case

The "nc_server.kmhdr" from the example "Using Netcat with *KMedia Header Engine*" in Section A.4.4:

```
1  # config for kmedia header engine
2  # filename: nc_server.kmhdr
3
4  # message states:
5  # plaintext : 1
6  # forward   : 3
7  msg_state 1
8
9  # crypto algorithm ID, 0 default
10 crypto_alg_id 0
11
12 addr_id 2
13
14 #addr_id 0
15 source_host_ip   127.0.0.1
16 source_host_port 55552
17
18 #addr_id 1
19 source_crypto_ip   127.0.0.1
20 source_crypto_port 55557
21
22 #addr_id 2
23 destination_crypto_ip 192.168.1.2
24 destination_crypto_port 55557
25
26 #addr_id 3
27 destination_host_ip   127.0.0.1
28 destination_host_port 55552
29
30 # address of application using kmedia header engine
31 destination_appl_ip 127.0.0.1
32 destination_appl_port 55555
```

To get "nc_client.kmhdr", which is also needed for the example, just copy "nc_server.kmhdr" and replace the line `destination_crypto_ip 192.168.1.2` with `destination_crypto_ip 192.168.1.1`.

B Message Generation

This appendix describes the methods used for the generation of messages which were used in the *KMedia* verification and performance tests. The size and format of the test messages is described in the Section 5.3.

First of all, a 100 KiB template file was created containing a 400 times repeated sequence of one-byte-size values incrementing from 0x00 to 0xFF. This file, named `payload.100KiB`, served for creating *KMedia* payload for all sizes of messages and was generated by a very tiny application. Its code can be found in Appendix I.3.

B.1 Forward Messages

The messages of a *forward* type were produced as illustrated in Figure B.1 with the commands from the Example B.1. First, *Netperf* reads in the `payload.100KiB` file to fill in the content of messages transmitted to *KMedia Header Engine*. Coming there, the messages are prefixed with an Extended Header according to the data in *KMedia Header Engine Configuration File* and forwarded to *Socat* which saves them finally in an appropriate file. The employed tools are described together with their command line options in the Appendix A.

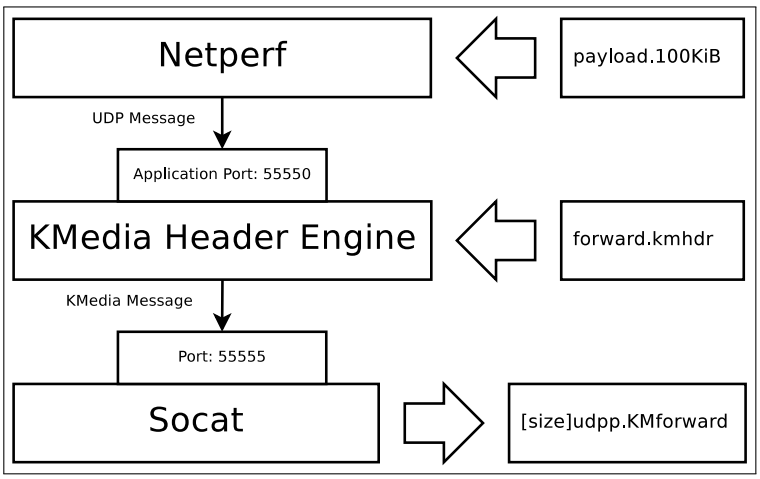

Figure B.1: *KMedia* Forward Message Generation

```
$ socat -b 65500 udp4-listen:55555 file:348udpp.KMforward,create
$ kmedia_header_engine forward.kmhdr to_appl
$ netperf -t UDP_STREAM -N -b 1 -w 1s -l 1 -F payload.100KiB -- \
          -P ,55550 -m 320
```

Example B.1: *KMedia* Forward Message Generation

The Example B.1 is an exact reference how a 348 bytes long *KMedia* message was built. First the *Socat* is instructed to write an incoming message into the file called `348udpp.KMforward`, denoting the fact that the file will contain a *KMedia* message (KM) of the *forward* type carrying 348 bytes long UDP payload (udpp)[1]. Then the *KMedia Header Engine* prepares *KMedia* Extended Header according to the data in the file `forward.kmhdr`. The file's content is shown in the Example I.1. Finally, *Netperf* sends a 320 bytes long message filled with data from the `payload.100KiB`. After the message is prefixed with Extended Header its size grows on to the final 348 bytes. The similar way messages of all sizes of the *forward* type were created.

B.2 Plaintext Messages

Because messages of the type *plaintext* differ from the *forward* ones just in the "Message State" field of the *KMedia* header, they were created by copying the *forward* messages and editing their *KMedia* headers[2] with the tool `hexedit` [32].

B.3 Ciphertext Messages

The messages of the last type were created by sending the already generated plaintext messages to the *KMedia* module and saving them afterwards into a file. Since *KMedia* utilizes crypto algorithms that are "randomized", i.e. every time the same block of data is encrypted the resulting ciphertext is different[3], the same instance of a plaintext message of a defined size was sent to *KMedia* multiple times. This way a bunch of unique ciphertext messages which represented one single message of the plaintext type was created in order to better verify the decryption functionality of *KMedia*.

In the following Figure B.2 a mechanism for generating the ciphertext messages is illustrated. First, *Netperf* transmits a "`[size]udpp.KMplaintext`" message, with "size" defining the message size, to *KMedia* where it is appropriately transformed with a default algorithm set through the *KMedia* sysfs interface. *Socat* receives such message from the *KMedia* and could save it directly into a file, but since it was desired to create multiple ciphertext messages, the *Socat* puts the received ciphertext message into a First In-First Out (FIFO) pipe. An application such as the `cat`[4] reads out the pipe and puts all the messages into a file with the long name `[size]udppp_[num]x[size]udppcA[ID].KMciphertext`. The "udppp" is an abbreviation for the UDP-Payload-Plaintext and thus the "size" in front of it indicates how long the message

[1] "UDP payload" is actually a synonym to "message".
[2] The first byte must have been changed from "0x32" to "0x12".
[3] Explained in the Section 2.3 on the page 9.
[4] `cat`'s task is to concatenate files and print on the standard output.

was before encryption. As the "udppc" denotes the UDP-Payload-Ciphertext, the size before it informs on the size of the message after encryption. Finally, the number "num" informs how many ciphertext messages, one after another, were saved in the file and the "ID" identifies the algorithm "A" which was used for the message encryption.

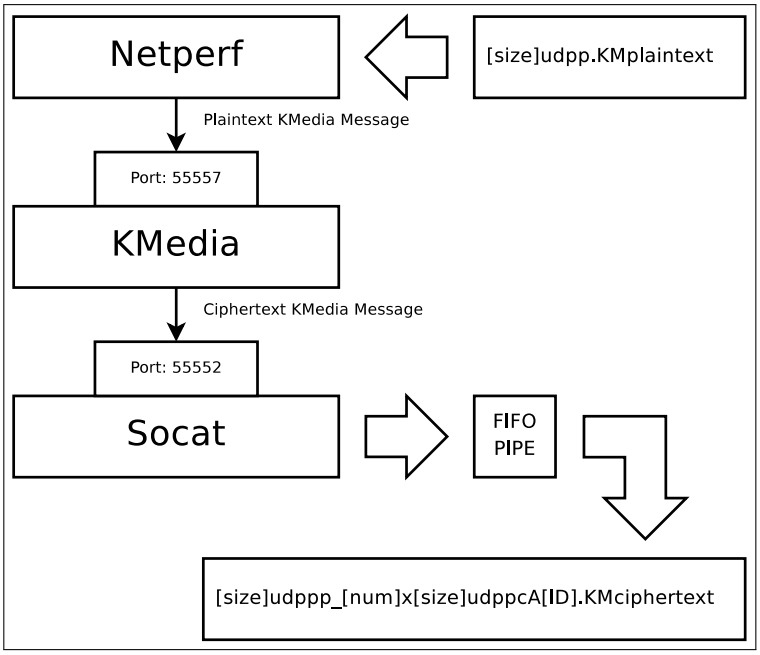

Figure B.2: *KMedia* Ciphertext Message Generation

The Example B.2 shows what command line options and parameters can be used to generate 40 ciphertext messages from a 348 bytes long plaintext message. First, the *KMedia* module must be inserted, then the named pipe created and the `cat` must start writing everything it reads out from the pipe into a file with an appropriate name. After also the *Socat* is ready to write all received messages into the pipe, the *Netperf* can start sending plaintext messages to the *KMedia* module. From the *Netperf*'s command line options it follows that a 348 bytes long message filled with the data from the `348udpp.KMplaintext` file will be sent every 50 milliseconds for the duration of 2 seconds. This implies that 40 messages will be cryptographically transformed by algorithm with the ID "100" and saved for a later use.

```
$ insmod kmedia_crypto_if.ko
$ insmod kmedia.ko max_msg_size=65403
$ mkfifo socat-pipe
$ cat socat-pipe > 348udppp_40x388udppcA100.KMciphertext
$ socat -b 65500 udp4-listen:55552 pipe:socat-pipe,wronly
$ netperf -t UDP_STREAM -N -b 1 -w 50m -l 2 -F 348udpp.KMplaintext -- \
          -P ,55557 -m 348
```

Example B.2: *KMedia* Ciphertext Message Generation

C *Freescale* MPC8349E Processor

High-level overview of the *Freescale*'s MPC8349E PowerQUICC II Pro Processor features is presented in this appendix. Figure C.1, downloaded from the website [11], shows the functional units of MPC8349E.

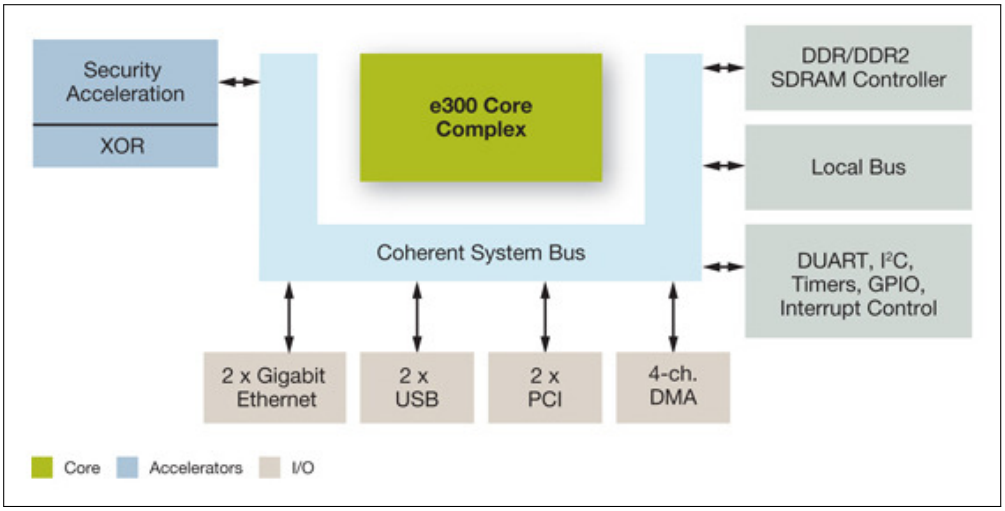

Figure C.1: MPC8349E SoC Block Diagram, ©*Freescale*

Major features of the MPC8349E processor according to the reference manual [Fre05b]:

- e300c1 PowerPC processor core
  - Enhanced version of the MPC603e core
  - High-performance, superscalar processor core with a four-stage pipeline and low interrupt latency times
  - Floating-point, integer, load/store, system register, and branch processing units
  - 32-Kbyte instruction cache and 32-Kbyte data cache with lockable capabilities
  - Dynamic power management
  - Enhanced hardware program debug features
  - Software-compatible with the Freescale processor families implementing the PowerPC architecture

  - Separate PLL that is clocked by the system bus clock
    - ATM/POS through the UPC
    - Serial ATM through the serial interface
    - HDLC/Transparent (bit rate up to 70 Mbit/s)
    - HDLC BUS (bit rate up to 10 Mbit/s)

- Parallel I/O
  - General purpose I/O
  - Open drain capability
  - Interrupt capability)

- Security engine optimized to handle all the algorithms associated with IPSec, SSL/TLS, SRTP, 802.11i, iSCSI, and IKE processing. The security engine contains four crypto-channels, a controller, and a set of crypto execution units (EUs). The execution units are:
  - Public key execution unit (PKEU) supporting the following:
    - RSA and Diffie-Hellman algorithms
    - Programmable field size up to 2048 bits
    - Elliptic curve cryptography
    - F2m and F(p) modes
    - Programmable field size up to 511 bits
  - Data encryption standard execution unit (DEU)
    - DES and 3DES algorithms
    - Two key (K1, K2, K1) or three key (K1, K2, K3) for 3DES
    - ECB and CBC modes for both DES and 3DES
  - Advanced encryption standard unit (AESU)
    - Implements the Rinjdael symmetric key cipher
    - Key lengths of 128, 192, and 256 bits
    - ECB, CBC, CCM, and counter (CTR) modes
  - ARC four execution unit (AFEU)
    - Implements a stream cipher compatible with the RC4 algorithm
    - 40- to 128-bit programmable key
  - Message digest execution unit (MDEU)
    - SHA with 160- or 256-bit message digest
    - MD5 with 128-bit message digest
    - HMAC with either algorithm
  - Random number generator (RNG)
  - Four crypto-channels, each supporting multi-command descriptor chains
    - Static and/or dynamic assignment of crypto-execution units through an integrated controller
    - Buffer size of 256 bytes for each execution unit, with flow control for large data sizes

- DDR SDRAM memory controller
  - Programmable timing supporting DDR-1 SDRAM
  - 32- or 64-bit data interface, up to 333-MHz data rate
  - Up to four physical banks (chip selects), each bank up to 1 Gbyte independently addressable
  - DRAM chip configurations from 64 Mbits to 1 Gbit with x8/x16 data ports
  - Full ECC support. When configured as 2x32-bit DDR memory controllers, both support ECC.
  - Support for up to 16 simultaneous open pages
  - Contiguous or discontiguous memory mapping
  - Read-modify-write support
  - Sleep mode support for self refresh SDRAM
  - Supports auto refresh
  - On-the-fly power management using CKE
  - Registered DIMM support
  - 2.5-V SSTL2 compatible I/O
- Dual three-speed (10/100/1000) Ethernet controllers (TSECs)
  - Dual IEEE 802.3, 802.3u, 820.3x, 802.3z, 802.3 AC compliant controllers
  - Support for different Ethernet physical interfaces:
    - 1000 Mbit/s IEEE 802.3 GMII/RGMII, 802.3z TBI/RTBI, full-duplex
    - 10/100 Mbit/s IEEE 802.3 MII full- and half-duplex
  - Buffer descriptors are backwards-compatible with MPC8260 and MPC860T 10/100 programming models
  - 9.6-Kbyte jumbo frame support
  - RMON statistics support
  - Internal 2-Kbyte transmit and 2-Kbyte receive FIFOs per TSEC module
  - MII management interface for control and status
  - Programmable CRC generation and checking
- Dual PCI interfaces
  - PCI specification Revision 2.2 compatible
  - Data bus widths (2 options):
    - Dual 32-bit data PCI interface that operates at up to 66 MHz
    - Single 64-bit data PCI interface operates at up to 66 MHz
  - PCI 3.3-V compatible
  - Not 5-V compatible
  - PCI host bridge capabilities on both interfaces
  - PCI agent mode supported on PCI1 interface

  - Support for PCI-to-memory and memory-to-PCI streaming
  - Memory prefetching of PCI read accesses and support for delayed read transactions
  - Support for posting of processor-to-PCI and PCI-to-memory writes
  - On-chip arbitration, supporting five masters on PCI1 and three masters on PCI2
  - Support for accesses to all PCI address spaces
  - Support for parity
  - Selectable hardware-enforced coherency
  - Address translation units for address mapping between host and peripheral
  - Dual address cycle support as target
  - Internal configuration registers accessible from PCI

- Universal serial bus (USB) dual role controller
  - Supports USB on-the-go mode, which includes both device and host functionality
  - Complies with USB Specification Revision 2.0
  - Supports operation as a stand-alone USB device
    - Supports one upstream facing port
    - Supports six programmable USB endpoints
  - Supports operation as a stand-alone USB host controller
  - Supports USB root hub with one downstream-facing port
  - Enhanced host controller interface (EHCI) compatible
  - Supports high-speed (480 Mbit/s), full-speed (12 Mbit/s), and low-speed (1.5 Mbit/s) operations
  - Supports external PHY with UTMI, serial and UTMI+ low-pin interface (ULPI)

- USB multi-port host controller
  - Supports operation as a stand-alone USB host controller
  - Supports USB root hub with one or two downstream-facing ports
  - Enhanced host controller interface (EHCI) compatible
  - Complies with USB Specification Revision 2.0
  - Supports high-speed (480 Mbit/s), full-speed (12 Mbit/s), and low-speed (1.5 Mbit/s) operations
  - Supports a direct connection to a high-speed device without an external hub
  - Supports external PHY with serial and low-pin count (ULPI) interfaces

- Local bus controller (LBC)
  - Multiplexed 32-bit address and data operating at up to 133 MHz
  - Four chip selects support four external slaves
  - Up to eight-beat burst transfers
  - 32-, 16-, and 8-bit port sizes are controlled by an on-chip memory controller
  - Three protocol engines available on a per chip select basis:

    - General-purpose chip select machine (GPCM)
    - Three user programmable machines (UPMs)
    - Dedicated single data rate SDRAM controller
  - Parity support
  - Default boot ROM chip select with configurable bus width (8-, 16-, or 32-bit)
- Programmable interrupt controller (PIC)
  - Functional and programming compatibility with the MPC8260 interrupt controller
  - Support for 8 external and 34 internal discrete interrupt sources
  - Support for one external (optional) and seven internal machine check interrupt sources
  - Programmable highest priority request
  - Four groups of interrupts with programmable priority
  - External and internal interrupts directed to host processor
  - Redirects interrupts to external INTA signal when in core disable mode
  - Unique vector number for each interrupt source
- Dual I2C interfaces
  - Two-wire interface
  - Multiple-master support
  - Master or slave I2C mode support
  - On-chip digital filtering rejects spikes on the bus
  - System initialization data is optionally loaded from I2C-1 EPROM by boot sequencer embedded hardware
- DMA controller
  - Four independent virtual channels
  - Concurrent execution across multiple channels with programmable bandwidth control
  - All channels accessible by local core and remote PCI masters
  - Misaligned transfer capability
  - Data chaining and direct mode
  - Interrupt on completed segment and chain
- DUART
  - Two 4-wire interfaces (RxD, TxD, RTS, CTS)
  - Programming model compatible with the original 16450 UART and the PC16550D
- Serial peripheral interface (SPI)
  - Master or slave support
- General-purpose parallel I/O (GPIO)
  - 64 parallel I/O pins multiplexed on various chip interfaces

- System timers
  - Periodic interrupt timer
  - Real-time clock
  - Software watchdog timer
  - Eight general-purpose timers
- IEEE 1149.1 compliant JTAG boundary scan
- Integrated PCI bus and SDRAM clock generation

D *Freescale*'s Security Engines

According to the document "Network/Embedded Processors" [fre09b], released in 2nd quarter 2009, *Freescale* offers processors with integrated Security Engine (SEC) of following versions: SEC 1.0, SEC 1.2, SEC 2.0, SEC 2.1, SEC 2.2, SEC 2.4, SEC 3.0, SEC 3.1 and SEC 3.3.

All the security engines are designed to off-load computationally intensive security functions, such as authentication, and bulk encryption from the processor core of the System on a Chip (SoC). Except for SEC 1.2 and SEC 2.2 all security engines support also key generation and exchange.

The following Table D.1 shows protocols whose algorithms are supported by the security engines. The "Y" label identifies that the SEC is optimized to process all algorithms associated with the given protocol.

	IPsec	IKE	SSL/ TLS	iSCSI	SRTP	IEEE 802.11i	802.16 WiMAX	3G	A5/3 for GSM and EDGE	GEA3 for GPRS	802.1AE MACSec
SEC 1.0	Y	Y	Y	Y	Y	Y	.	.	.	.	.
SEC 1.2	Y	.	.	.	Y	Y	.	.	.	.	.
SEC 2.0	Y	Y	Y	Y	Y	Y	.	.	.	.	.
SEC 2.1	Y	Y	Y	Y	Y	Y	.	.	.	.	.
SEC 2.2	Y	.	Y	Y	Y	Y	.	.	.	.	.
SEC 2.4	Y	Y	Y	Y	Y	Y	.	.	.	.	.
SEC 3.0	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	.
SEC 3.1	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	.
SEC 3.3	Y	Y	Y	Y	Y	Y	Y	.	.	.	Y

Table D.1: SEC Versions vs. Support for Protocol's Algorithms

In the next sections the features of all SECs are listed. First, the properties are shown which are common to all SECs of a given main version, SEC 1.x, SEC 2.x and SEC 3.x and afterwards the additional properties of subversions are mentioned. To each SEC version it is also noted what processor types integrate it and from which reference manual the data was taken.

D.1 SEC 1.x

Common features:

- DEU: Data Encryption Standard execution unit
  - DES, 3DES
  - Two-key (K1, K2, K1) or three-key (K1, K2, K3)
  - ECB and CBC modes for both DES and 3DES
- AESU: Advanced Encryption Standard execution unit
  - Implements the Rijndael symmetric-key cipher
  - ECB, CBC, and counter modes
  - 128-, 192-, 256-bit key lengths
- MDEU: message digest execution unit
  - SHA with 160-bit or 256-bit message digest
  - MD5 with 128-bit message digest
  - HMAC with either algorithm

D.1.1 SEC 1.0

Processors: MPC 8248, MPC 8272
Data from MPC8272 Reference Manual

Properties as listed in SEC 1.x and following:

- PKEU: Public key execution unit that supports the following:
  - RSA and Diffie-Hellman
    - Programmable field size up to 2048 bits
  - Elliptic curve cryptography
    - F2m and F(p) modes
    - Programmable field size up to 511 bits
- AFEU: ARC four execution unit
  - Implements a stream cipher compatible with the RC4 algorithm
  - 40- to 128-bit programmable key
  - RNG: 1 Random number generator
- Master/slave logic, with DMA
  - 32-bit address/64-bit data
  - Up to 100-MHz operation
- Four crypto-channels, each supporting multi-command descriptor chains
  - Static and/or dynamic assignment of execution units through an integrated controller
  - Buffer size of 512 bytes per execution unit, with flow control for large data sizes

D.1.2 SEC 1.2

Processors: MPC 875, MPC 885
Data from MPC885 Reference Manual

Also called SEC Lite, derived from Motorola MPC185 "Talos" security processor, a member of the Smart Networks platform's S1 family of security processors developed for the commercial networking market.

Properties as listed in SEC 1.x and following:

- Crypto-channel supporting multi-command descriptor chains
- Integrated controller managing internal resources, and bus mastering
- Buffer size of 256 bytes for the DEU, AESU, and MDEU, with flow control for large data sizes

D.2 SEC 2.x

Common features:

- DEU: Data encryption standard execution unit
  - DES, 3DES
  - Two key (K1, K2, K1) or three key (K1, K2, K3)
  - ECB and CBC modes for both DES and 3DES
- AESU: Advanced encryption standard execution unit
  - Implements the Rinjdael symmetric key cipher
  - ECB, CBC, CCM, and counter modes
  - 128-, 192-, 256-bit key lengths
- MDEU: Message digest execution unit
  - SHA with 160- or 256-bit message digest
  - MD5 with 128-bit message digest
  - HMAC with either algorithm
- Master/slave logic, with DMA capability
  - Master interface allows multiple pipelined requests
  - 36-bit address/64-bit data
  - Up to 166-MHz operation
  - DMA blocks can be on any byte boundary
- Scatter/Gather capability

  - Gather capability enables the SEC 2.x to concatenate multiple segments of memory when reading input data
  - Similarly, scatter capability enables SEC 2.x to write to multiple segments of memory when writing output data

D.2.1 SEC 2.0

Processors: MPC 8343E, MPC 8347E, MPC 8349E, MPC 8541E, MPC 8555E
Data from MPC8555E Reference Manual

The SEC 2.0 is derived from integrated security cores found in other members of the PowerQUICC family, including SEC 1.0, the version implemented in the MPC8272.

Properties as listed in SEC 2.x and following:

- PKEU: Public key execution unit that supports the following:
  - RSA and Diffie-Hellman algorithms
    - Programmable field size up to 2048 bits
  - Elliptic curve cryptography
    - F2m and F(p) modes
    - Programmable field size up to 511 bits
- AFEU: ARC four execution unit
  - Implements a stream cipher compatible with the RC4 algorithm
  - 40- to 128-bit programmable key
- RNG: Random number generator
- Four channels, each supporting a queue of commands (descriptor pointers)
  - Dynamic assignment of crypto-execution units through an integrated controller
  - 256-byte buffer FIFOs on data input and output paths of each execution unit, with flow control for large data sizes

D.2.2 SEC 2.1

Processors: MPC 8543E, MPC 8545E, MPC 8547E, MPC 8548E, MPC 8544E, MPC 8567E, MPC 8568E
Data from MPC8548E Reference Manual

The SEC 2.1 is derived from integrated security cores found in other members of the PowerQUICC family, including the SEC 2.0, the version implemented in the MPC8555E and MPC8541E.

Properties as listed in SEC 2.x and following:

- PKEU: Public key execution unit that supports the following:

  - RSA (Rivest-Shamir-Adleman) and Diffie-Hellman algorithms
    - Programmable field size up to 2048 bits
  - Elliptic curve cryptography
    - F2m and F(p) modes
    - Programmable field size up to 511 bits

- AFEU: ARC Four execution unit
  - Implements a stream cipher compatible with the RC4 algorithm
  - 40- to 128-bit programmable key

- KEU: Kasumi execution unit
  - Implements the Kasumi cipher
  - Performs F8 encryption and F9 integrity checking as required for 3GPP
  - Additionally performs A5/3 and GEA-3 modes as used in GSM, EDGE, and GPRS

- MDEU: Message digest execution unit
  - SHA also with 160-bit, 224-bit, or 256-bit message digest
  - MD5 with 128-bit message digest
  - HMAC with either algorithm

- RNG: Random number generator

- XOR parity generation accelerator for RAID applications

- Four crypto-channels, each supporting a queue of commands (descriptor pointers)
  - Dynamic assignment of crypto-execution units via an integrated controller
  - 256-byte buffer FIFOs on data input and output paths of each execution unit, with flow control for large data sizes

D.2.3 SEC 2.2

Processors: MPC 8313E
Data from MPC8313E Reference Manual

The SEC 2.2 is derived from integrated security cores found in other members of the PowerQUICC family, including SEC 1.0, the version implemented in the MPC8272 and SEC 2.0, implemented on the MPC8555.

Properties as listed in SEC 2.x and following:

- MDEU: Message digest execution unit
  - SHA with 160-, 224-, or 256-bit message digest
  - MD5 with 128-bit message digest
  - HMAC with either algorithm

- One channel, supporting a queue of commands (descriptor pointers)
  - Dynamic assignment of execution units through an integrated controller
  - 256-byte buffer FIFOs on data input and output paths of each execution unit, with flow control for large data sizes. The input and output FIFOs are shared between AESU and DEU; MDEU has its own input FIFO.

D.2.4 SEC 2.4

Processors: MPC 8358E, MPC 8360E
Data from MPC8360E Reference Manual

The SEC 2.4 is derived from integrated security cores found in other members of the PowerQUICC family, including SEC 1.0, the version implemented in the MPC8272/MPC8248.

Properties as listed in SEC 2.x and following:

- PKEU: Public key execution unit that supports the following:
  - RSA and Diffie-Hellman algorithms
    - Programmable field size up to 2048 bits
  - Elliptic curve cryptography
    - F2m and F(p) modes
    - Programmable field size up to 511 bits
- AFEU: ARC four execution unit
  - Implements a stream cipher compatible with the RC4 algorithm
  - 40- to 128-bit programmable key
- MDEU: Message digest execution unit
  - SHA with 160-, 224-, or 256-bit message digest
  - MD5 with 128-bit message digest
  - HMAC with either algorithm
- RNG: Random number generator
- XOR parity generation accelerator for RAID applications
- Four channels, each supporting a queue of commands (descriptor pointers)
  - Dynamic assignment of crypto-execution units through an integrated controller
  - 256-byte buffer FIFOs on data input and output paths of each execution unit, with flow control for large data sizes

D.3 SEC 3.x

Common features:

- AESU: Advanced Encryption Standard Unit
  - Implements the Rinjdael symmetric key cipher
  - Modes providing data confidentiality: ECB, CBC, CCM, Counter, GCM, CBC-RBP, OFB-128, and CFB-128.
  - Modes providing data authentication: CCM, GCM, CMAC (OMAC1), and XCBC-MAC.
  - 128, 192, 256 bit key lengths (only 128 bit keys in XCBC-MAC)
  - ICV checking in CCM, GCM, CMAC (OMAC1), and XCBC-MAC mode
  - XOR operations on 2 - 6 sources for RAID
- CRCU: Cyclical Redundancy Check Unit
  - Implements CRC32C as required for iSCSI header and payload checksums, CRC32 as required for IEEE Std. 802 packets, and programmable CRC modes
  - ICV checking
- DEU: Data Encryption Standard Execution Unit
  - DES, 3DES
  - Two key (K1, K2, K1) or Three Key (K1, K2, K3)
  - ECB, CBC, CFB-64 and OFB-64 modes for both DES and 3DES
- MDEU: Message Digest Execution Unit
  - Implements SHA with 160-bit, 224-bit, 256-bit, 384-bit, and 512-bit message digest (as specified by the FIPS 180-2 standard)
  - Implements MD5 with 128-bit message digest (as specified by RFC 1321)
  - Implements HMAC computation with either message digest algorithm (as specified in RFC 2104 and FIPS-198)
  - Implements SSL MAC computation
  - ICV checking
- PKEU: Public Key Execution Unit
  - RSA and Diffie-Hellman
    - Programmable field size up to 4096 bits
  - Elliptic curve cryptography
    - F2m and Fp modes
    - Programmable field size up to 1023 bits
  - Run time equalization to protect against timing and power attacks

In addition to the execution units, SEC 3.x also includes:

- A context switching polychannel, permitting operation of up to four virtual channels, where each channel:
  - Supports a queue of commands (descriptor pointers) to be executed
  - Provides dynamic arbitration for needed crypto-execution units
  - Manages up to two execution units (one ciphering and one hashing), and configures for any required data transfers from one to another
  - Performs flow-control management of buffer FIFOs on the inputs and outputs of execution units
  - Supports scatter/gather of input and output data (where the term data is used loosely, and includes keys, context, ICV values, etc.), enabling concatenation of multiple segments of memory when reading or writing data
  - Masters data bursts on 32-byte boundaries to optimize bus throughput
- Master and slave interfaces, with DMA capability
  - 32- or 36-bit address/64-bit data
  - Master interface allows pipelined requests
  - DMA data blocks can start and end on any byte boundary

D.3.1 SEC 3.0

Processors: MPC 8377E, MPC 8378E, MPC 8379E, MPC 8535E, MPC 8536E, MPC 8572E
Data from MPC8379E Reference Manual

The SEC 3.0 is derived from integrated security cores found in other members of the PowerQUICC II and PowerQUICC III families.

The security engine includes eight different execution units (EUs). Where data flows in and out of an EU, each has buffer FIFOs of at least 256 bytes.

Properties as listed in SEC 3.x and following:

- AESU: Advanced Encryption Standard unit
  - Implements the Rijndael symmetric key cipher per U.S. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197.
- AFEU: ARC4 execution unit
  - Implements a stream cipher compatible with the RC4 algorithm
  - 8- to 128-bit programmable key
- KEU: Kasumi execution unit
  - Implements cipher and authentication modes f8 and f9 used in 3G, A5/3 for GSM and EDGE, and GEA3 for GPRS

  - 128-bit confidentiality key and 128-bit integrity key
  - ICV checking for f9
- RNGU: Random number generator unit
  - True Random Number Generator (TRNG)

D.3.2 SEC 3.1

Processor: MPC8569E
Data from MPC8569E Reference Manual

The SEC 3.1 is derived from integrated security cores found in other members of the PowerQUICC II and PowerQUICC III families.

The security engine includes eight different execution units (EUs). Where data flows in and out of an EU, each has buffer FIFOs of at least 256 bytes.

Properties as listed in SEC 3.x and following:

- AESU: Advanced Encryption Standard unit
  - Implements the Rijndael symmetric key cipher per U.S. National Institute of Standards and Technology (NIST), Federal Information Processing Standard (FIPS) 197.
- AFEU: ARC4 execution unit
  - Implements a stream cipher compatible with the RC4 algorithm
  - 8- to 128-bit programmable key
- KEU: Kasumi execution unit
  - Implements cipher and authentication modes f8 and f9 used in 3G, A5/3 for GSM and EDGE, and GEA3 for GPRS
  - 128-bit confidentiality key and 128-bit integrity key
  - ICV checking for f9
- STEU: SNOW3G execution unit
  - Implements cipher and authentication modes UEA2 (f8) and UIA2 (f9)
  - 128-bit confidentiality key and 128-bit integrity key
  - ICV checking for f9
- RNGU: Random number generator unit
  - Combines a True Random Number Generator (TRNG) and a deterministic Pseudo-RNG, based on SHA, as described in FIPS 186-2, Appendix 3.1.

In addition to the execution units, SEC 3.1 also includes:

- A context switching polychannel, permitting operation of up to four virtual channels, where each channel:
  - Can be configured for either host processor (Note: this feature follows from the fact that MPC8569E has four RISCs processors)

D.3.3 SEC 3.3

Processors: MPC 8314E, MPC 8315E, MPC 8321E, MPC 8323E
Data from MPC8315E Reference Manual

The SEC 3.3 is derived from integrated security cores found in other members of the PowerQUICC II Pro family.

The security engine includes six different execution units (EUs). Where data flows in and out of an EU, each has buffer FIFOs of at least 256 bytes.

Properties as listed in SEC 3.x and following:

- RNGU: Random Number Generator
  - Combines a True Random Number Generator (TRNG) and NIST-approved Pseudo-Random Number Generator (PRNG) (as described in Annex C of FIPS140-2 and ANSI X9.62).

E Securing Real-Time Multimedia Applications

This appendix lists commonly used security protocols that protect data transmitted through Internet and analyzes their suitability for securing real-time multimedia streams. As it will be explained, the Secure Real-Time Transport Protocol (SRTP) provides recently the best suitable protection for the multimedia data.

Utilizing Internet resources for voice communication is quite common and popular nowadays. The employed technology for voice data transmission works on IP based networks and is therefore called Voice over IP (VoIP). This way voice data share the same resources with other network traffic, but also the same security issues. A VoIP call can be more easily intercepted than a telephone call based on a circuit switching technology. Therefore it is important to provide some security services as e.g. confidentiality to the transmitted voice data in an open packet-based Internet.

Before different security protocols will be introduced and briefly analyzed it is noted that multimedia applications packetize a voice or video after digitalization mostly by the Real-Time Transport Protocol (RTP). This protocol creates messages from a stream of digitalized data and passes them through a socket to a connectionless unreliable UDP. A reliable TCP connection is not needed since real-time multimedia data as e.g. voice are loss-tolerant. Moreover, TCP could add unnecessary delays caused by retransmission of lost packets which could cause quality degradation of the delay-sensitive stream.

Different security approaches evolved to secure data traffic in the Internet but not all of them are suited for real-time multimedia applications. To the well known, and commonly used, security protocols belong IPsec on the internet layer, Transport Layer Security (TLS)/Secure Socket Layer (SSL) on the transport layer and SRTP with H.235 on the application layer. TLS/SSL is not suitable since it was designed for web browser to web server communication and is based on TCP. The next protocol H.235[1] was designed for multimedia applications but is "currently the less adopted solution in the context of voice communications over IP networks, due to its complexity, and time-consuming connection setup phase" [SGB08].

IPsec, a suite of protocols developed to provide network layer Virtual Private Network (VPN), can be used in general but it has some drawbacks which make it less suitable for media traffic. A component of IPsec, the Internet Key Exchange (IKE) protocol, establishes security associations,

[1] H.235 describes security services for H.323, a multitude of ITU-T multimedia communication standards.

associated with secret key exchange, only between two parties [Kau05, Section 1.1]. Therefore *no multicast* is supported. Moreover, "IKE is a very complex protocol and often the reason for *interoperability problems* between different IPsec implementations" [LIR02]. The third issue is that IPsec dramatically increases the packet size which in turn degrades the end-to-end bandwidth utilization, increases transmission delay and worsens CPU usage [BBR02]. According to [BBR02] from a 40 bytes long voice payload, which would make an 80 byte IP packet can become after IPsec transformation a packet with 134 bytes. Some solutions how to decrease the header overhead exist but they put additional load on processor and do not address interoperability or multicast issues. Therefore "IPSec VPN needs many enhancements to support security for VoIP" [DTB08].

"For real-time UDP traffic the main alternative is SRTP" [DTB08]. It supports media stream ciphering, caller authentication, data integrity check and replay protection and "it does not depend on any key management standard and has good results with low increase in packet size" [YAM09]. For key management e.g. the Multimedia Internet KEYing (MIKEY) could be chosen which supports multicast services such as peer-to-peer, simple one-to-many, decentralized small group of many-to-many and a large group of many-to-many with a centralized control unit [ACL+04, Section 2.1].

F Performance of Some Authenticated Encryption Modes

This appendix shows some results achieved by other people in the field.

The following two Tables F.1 and F.2 provide results obtained by David A. McGrew and John Viega which were published in their article [MV04].

Bytes	16	20	40	44	64	128	256	552	576	1024	1500	8192	IPI
GCM	64.0	71.1	91.4	93.9	102	114	120	124	124	126	127	128	77.7
CWC	10.7	13.1	23.7	25.6	34.1	53.9	75.9	97.0	98.0	109	115	125	35.3
OCB	5.82	7.19	13.6	14.8	20.5	35.3	55.4	79.6	80.8	96.4	105	123	22.8

Table F.1: Hardware performance in bits per clock cycle of various AES-128 modes of operation, with three significant digits, for a variety of packet sizes and the Internet Performance Index (IPI). ©[MV04]

Bytes	16	20	40	44	64	128	256	552	1024	1500	8192	IPI
GCM64K	**136**	**167**	**227**	**253**	**223**	**263**	**267**	**273**	266	266	258	**268**
GCM4K	116	140	190	207	192	213	229	237	239	247	240	240
GCM256	88.4	107	148	160	177	162	171	183	181	183	182	182
OCB	89.5	85.7	140	150	185	225	255	261	**273**	**275**	**282**	260
CWC	45.7	51.9	73.4	75.5	88.1	104	116	127	131	124	135	121
EAX	46.0	44.9	73.4	80.0	102	129	148	157	165	167	174	156
CCM	91.3	88.9	123	133	142	171	163	168	174	172	175	168
CBC-HMAC	6.3	8.0	15.2	16.6	23.4	39.0	64.5	96.0	117	129	156	88.6

Table F.2: Software performance in bits per kilocycle (or equivalently, megabits per second on a 1GHz Motorola G4 processor) to three significant digits, on various packet sizes, and the Internet Performance Index (IPI), for various AES-128 modes of operation and SHA1 for the HMAC mode. GCM256, GCM4K, and GCM64K refer to GCM with 256, 4K, and 64K byte table sizes, respectively. The highest entry in each column is highlighted. ©[MV04]

Algorithm	Through-put, MiB per Second	Cycles Per Byte	μ-seconds to Setup Key and IV	Cycles to Setup Key and IV
Encryption Only				
AES/CTR (128-bit key)	198	10.6	0.436	956
AES/CTR (192-bit key)	164	12.8	0.434	952
AES/CTR (256-bit key)	140	15.0	0.473	1038
AES/CBC (128-bit key)	148	14.1	0.358	785
AES/CBC (192-bit key)	129	16.3	0.367	805
AES/CBC (256-bit key)	113	18.5	0.411	902
Authenticated Encryption				
AES/GCM (64K tables)	148	14.1	10.557	23163
AES/GCM (2K tables)	130	16.1	1.471	3227
AES/CCM	84	24.8	0.559	1227

Table F.3: Software performance of some algorithms implemented in the Crypto++ Library. The tests were performed on an AMD Opteron 8354 4x2.2 GHz processor under Linux. ©www.cryptopp.com

The next Table F.3 provides benchmarking results from the website [2].

G *KMedia* Development

This appendix describes the recommended system for *KMedia*, then the *KMedia* development environment, analyzes also the Linux implementation of IPsec and shows another way of *KMedia*'s network functionality implementation.

G.1 System Requirements

The *KMedia* kernel module should be able to execute and provide its functionality on any system with the Linux kernel 2.6.27 and higher. Cryptographic algorithms must be either provided by the kernel or implemented in a cryptographic accelerator which is supported by *Talitos*.

It is important to note, that a bug in the Linux kernels up to version 2.6.30 causes that the software driver fails to transform correctly *KMedia* messages which have more than 4051 bytes. The bug was solved first in the kernel 2.6.30 and therefore kernels from that version must be used in order to get correct functionality also for longer messages.

G.2 Development Environment

Systems of the following configurations were employed for the development of *KMedia*.

- **Target System**
  - Processor Board: *Freescale*'s MPC8349E-MDS
  - Processor Core: e300c1 at 528 MHz, Revision 1.1, 32-bit Power Architecture, MPC8349E PowerQUICC II Pro family
  - DRAM: 256 MiB 64-bit DDR1 at 264 MHz data rate
  - Crypto Accelerator: SEC 2.0 at 88 MHz
  - Flash Memory: 8 MiB
  - Bootloader: U-Boot v2009.03
  - OS: Linux Kernel 2.6.29, 2.6.30
- **Host Development System**

- OS: Linux Kernel 2.6.24, 2.6.32
- Cross Development Toolchain: gcc 4.3.3 based

Appendix C gives a detailed functional overview of the utilized *Freescale* processor employed on the target system and Appendix D lists features provided by the various versions of *Freescale*'s cryptographic accelerators, of the used SEC 2.0 inclusive.

G.3 A Short Analysis of the Linux IPsec Implementation

In this section it is described how the Linux kernel implementation of ESP protocol, which belongs to the suite of IPsec protocols, handles a packet that must be either encrypted before transmition or decrypted after reception. First the transport mode is analyzed in which the IP layer extended with IPsec receives a segment from, or sends a segment to, transport layer. Afterwards, also the tunnel mode is briefly discussed in which case the IP layer exchanges packets with the underlying link layer. The mentioned functionality was observed in the Linux kernel version 2.6.29. Also the here listed functions correspond to this version of the kernel source.

When following the output flow, beginning with the function `udp_sendmsg()` in source code file `net/ipv4/udp.c`, it can be find out that routing is performed before UDP and IP header creation. During routing lookup it is not decided only on the next route for the packet but also what IPsec security policies should be applied, if any. In case no IPsec operations should take place the routing output is represented with a single entry describing whether the packet should be delivered locally, forwarded to a single host or to a group of hosts, etc. If the route should be secured by IPsec then a list of entry instances is created. Only the last item in the list carries the routing information, all others describe how the packet should be transformed according to the IPsec rules associated with the route. When a packet comes to the point when the rules should be applied, the `skb->dst->output(skb)` function, its UDP payload is already prefixed with UDP and IP header. Therefore if ESP protocol in transport mode is chosen for packet transformation the IP header must be moved to make place for ESP header and additionally also for the IV if it is required by the crypto algorithm that should be used. This IP header replacement is performed by the function `xfrm4_transport_output()`[1] in file `net/ipv4/xfrm4_mode_transport.c`. Afterwards the ESP header is created and the crypto transformations are performed according to the entry list created at routing lookup. Finally such packet, illustrated on the right side of Figure 2.1 on page 9, is sent to the link layer of the network stack.

If a packet secured by ESP in transport mode reaches its final host then the ESP protocol handler is invoked by the function `ip_local_deliver_finish()` which is good described in [Ben05, Chapter 24]. After the packet is decrypted by ESP the IP header is moved over the ESP header and IV, since they are not needed anymore, by function `xfrm4_transport_input()` to lie directly in front of the recently decrypted UDP datagram. Afterwards `ip_local_deliver_finish()` invokes the UDP handler which then continues with the packet processing.

In tunnel mode the security gateway finds out to which route the packets belong and whether they underlie some IPsec policies. The forwarding mechanism utilizes the same routing lookup

[1] Many programming constructs in the code which implements IPsec are prefixed with an "`xfrm_`" to denote "eXtensible FRaMework (XFRM)" on which IPsec implementation in the Linux kernel is based. It allows the IP packets transformation in the network stack.

as described for outgoing messages in transport mode. If packets should be cryptographically transformed a list of entries is created which describes what crypto operations in what order should be performed and only the last one defines the route. Afterwards the packets are accordingly modified and sent to their next destination. No header moves were encountered in case such packets must have been encrypted, but, interestingly, when they must have been decrypted the link layer header was copied to a new place in function `xfrm4_mode_tunnel_input()`.

G.4 Transparent *KMedia*

This section describes how *KMedia*'s network functionality could be implemented in order to provide transparent cryptographic services to hosts which want to communicate through a channel secured by *KMedia*.

When clients communicate with servers that are not in the local network they have to send the packets to the router which then forwards them to the next router on the path toward the wanted server. This way, in case no Network Address Translation (NAT) is performed, the destination and source IP addresses in the IP header do not change on the route to the destination. What changes are the Ethernet MAC addresses in the link layer header of the transmitted packet. The here presented concept requires only that clients would send the packets to *KMedia* instead of the router.

Figure G.1 illustrates how this variant could work on an example of a client `PC_A1` in local network `A` which transmits a message to the server `PC_B1` in local network `B`. First the client sends a frame of format `Frame_A` (depicted in the Figure) to *KMedia* residing in the same local network. After `KMedia_A` receives it through a "packet" socket[2] the original IP header, thus also the source and destination IP addresses, together with the UDP header and the UDP payload are encrypted. Such encrypted packet is transmitted afterwards in `Frame_B` from `KMedia_A` as a normal UDP payload to `KMedia_B` residing in the same local network as the destination server `PC_B1`. *KMedia* in the local network `B` receives such encrypted UDP payload, decrypts it to plaintext and sends through a "raw" socket[3] which prepends to the packet an appropriate link layer header and transmits such frame, `Frame_C` in the Figure, to the destination `PC_B1`.

[2] packet_socket = socket(PF_PACKET, SOCK_DGRAM, int protocol); "Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level." [6]

[3] raw_socket = socket(PF_INET, SOCK_RAW, int protocol); "A raw socket receives or sends the raw datagram not including link level headers." [7]

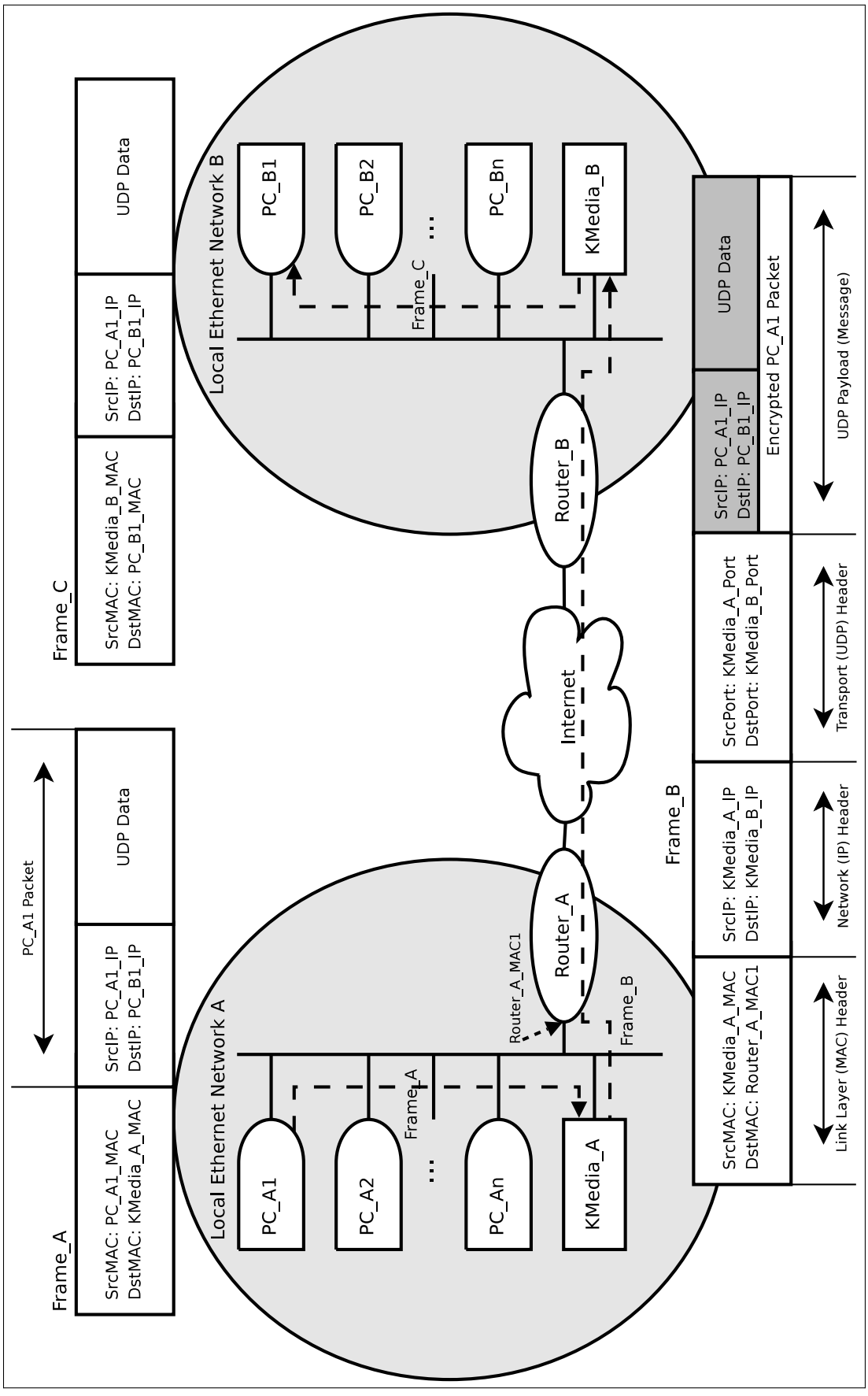

Figure G.1: Example of Data Flow in Network with *KMedia* Using Medium Access Control (MAC) Addressing (`Src` stands for "Source" and `Dst` for "Destination")

H Source Code and *KMedia* Webpage

For reasons of space, source code of the *KMedia* kernel module and of some benchmark tools presented in Chapter A is available only on the following website:

`http://www.ict.tuwien.ac.at/sysari/KMedia`.

The webpage provides, except for the source code, also all documents created during the work on the master thesis. This includes: the master thesis itself, presentation slides and a presentation poster. The slides and the poster summarize briefly the characteristics of the *KMedia* module, the testing method and the obtained results.

H.1 *KMedia*

`http://www.ict.tuwien.ac.at/sysari/KMedia/#KMediaKernelModule`

H.2 *KMedia Message Checker*

`http://www.ict.tuwien.ac.at/sysari/KMedia/#KMediaMessageChecker`

H.3 *KMedia Header Engine*

`http://www.ict.tuwien.ac.at/sysari/KMedia/#KMediaHeaderEngine`

H.4 *KMedia Statistics Reader*

`http://www.ict.tuwien.ac.at/sysari/KMedia/#KMediaStatisticsReader`

H.5 *KMedia CPU Top*

`http://www.ict.tuwien.ac.at/sysari/KMedia/#KMediaCPUTop`

H.6 *KMedia CPU Looper*

`http://www.ict.tuwien.ac.at/sysari/KMedia/#KMediaCPULooper`

I Test Environment

I.1 *KMedia* Test Message Sizes

The Table I.1 lists *KMedia* plaintext message sizes engaged in *KMedia*–DUT performance tests. It is described in Section 5.3.1. The row in gray indicates that the Maximum Transmission Unit (MTU) was reached, all messages above that size are fragmented into multiple frames.

Frame Size	Message Size	*KMedia* Payload	Min. Encryption Size
71	29	1	25
128	86	58	82
256	214	186	210
390	348	320	344
512	470	442	466
768	726	698	722
1024	982	954	978
1280	1238	1210	1234
1500	1458	1430	1454
1514	1472	1444	1468
-	1476	1448	1472
-	2048	2020	2044
-	3072	3044	3068
-	4096	4068	4092

Table I.1 – Continued on next page

Continued from previous page			
Frame Size	Message Size	*KMedia* Payload	Min. Encryption Size
-	6144	6116	6140
-	8192	8164	8188
-	12288	12260	12284
-	16384	16356	12380
-	24576	24548	24572
-	32768	32740	32764
-	49152	49124	49148
-	65403	65375	65399

Table I.1: *Plaintext* Message Sizes in Bytes

I.2 *KMedia* Extended Header Content Used in the Tests

The following Table I.2 shows configuration of *KMedia* header in test messages of different types as described in 5.3. Messages of *ciphertext* type have in their "Crypto Algorithm ID" field the ID of the algorithm which was used for their encryption and authentication and the remaining types have the ID set to zero in order to use default algorithm configured at DUT through *KMedia* sysfs interface, specified in Section 3.4.2. The not shown "Message Length" header field depends on the real size of the *KMedia* message and not on its type.

KMedia Header Field Name	Forward Message	Plaintext Message	Ciphertext Message
Message State	3	1	2
Address ID	2	2	2
Crypto Algorithm ID	0	0	Encryption Alg. ID

Table I.2: Content of *KMedia* Header Fields

In Example I.1 is shown what values were used in a configuration file for *KMedia Header Engine* tool in order to set the Extended Header of a *forward* message. The same IP addresses and port numbers were used also for each *plaintext* message. All parameters in the example are discussed in Section A.4.1.

```
#file name: forward.kmhdr
msg_state     3
crypto_alg_id 0
addr_id       2

source_host_ip          192.168.1.1
source_host_port        55556
source_crypto_ip        192.168.1.2
source_crypto_port      55557
destination_crypto_ip   192.168.1.1
destination_crypto_port 55552
destination_host_ip     192.168.1.1
destination_host_port   55552
```

Example I.1: Extended Header of a Forward Type Message

I.3 *Creator of File with KMedia Test-Payload* Source Code

This application in Example I.2 is described in Section 5.3.

```
1  /*============================================================================*
2   * PROJECT:      Master Thesis - KMEDIA                                       *
3   * UNIT:         Creator of File with KMedia Test-Payload                     *
4   * Workfile:     kmedia_filecreator.c                                         *
5   * Date:         12/2009                                                      *
6   * Version:      1.0                                                          *
7   * Description:  creates a file containing numbered octets                    *
8   *                00 01 02 03 ...  FD FE FF 00 01 02 ...                      *
9   * Author:       Ondrej Cevan, Frequentis, TU Vienna                          *
10  * Tutor:        H. Kronstorfer, Frequentis                                   *
11  * Substitute:   A. Reisenbauer, Frequentis                                   *
12  * Licence:      GNU GPL version 2 or later                                   *
13  *============================================================================*/
14 #include <stdio.h>
15 #include <sys/stat.h>
16 #include <fcntl.h>
17 #include <string.h>
18
19 int main()
20 {
21   FILE *fd;
22   int i;
23   char tmp[256];
24
25   fd = fopen("payload.100KiB", "w+");
26   if (fd == NULL)
27     return -1;
28
29   for (i = 0; i < 256; i++)
30     tmp[i] = i;
31
32   /*400 * 256B = 100KiB*/
33   for (i = 0; i < 400; i++)
34     fwrite(tmp, 1, 256, fd);
35
36   fflush(fd);
37   fclose(fd);
38
39   return 0;
40 }
```

Example I.2: *KMedia* Test-Payload Generator

I.4 Iptables Set Up

According to the manual pages of *Iptables* tool, which has its website at [29]:

> *Iptables* is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel. The NAT table is consulted when a packet that creates a new connection is encountered. It consists of three built-ins: PREROUTING (for altering packets as soon as they come in), OUTPUT (for altering locally-generated packets before routing), and POSTROUTING (for altering packets as they are about to go out).

The *KMedia*–DUT at address `192.168.1.2` was configured as shown in Example I.3. First the whole NAT table was erased, then a PREROUTING rule was set. The packets going from the address `192.168.1.1` to the device's port 55553 must have changed the destination address into `192.168.1.1:55553`. This way it was ensured that they will be routed to go back to `192.168.1.1`. Afterwards when their were just about to be sent to their new destination the POSTROUTING rule ensured a change of their source address, from the old `192.168.1.1` into `192.168.1.2`. Packets with these modifications look like they would be sent by an application residing on the device under test. Afterwards the command following POSTROUTING rule checks whether the table was set up correctly by printing out the NAT table. To activate IP forwarding capability of the device, as configured by the previous steps, the last command from Example I.3 must be also executed.

```
# ./iptables-static -t nat --flush
# ./iptables-static -A PREROUTING -t nat -p udp -s 192.168.1.1 -d 192.168.1.2 \
                    --dport 55553 -j DNAT --to-destination 192.168.1.1:55553
# ./iptables-static -A POSTROUTING -t nat -p udp -s 192.168.1.1 -d 192.168.1.1 \
                    -j SNAT --to-source 192.168.1.2
# ./iptables-static -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       udp  --  192.168.1.1          192.168.1.2      udp dpt:55553
                                                         to:192.168.1.1:55553
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       udp  --  192.168.1.1          192.168.1.1      to:192.168.1.2

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
#
# echo "1" > /proc/sys/net/ipv4/ip_forward
```

Example I.3: Iptables–NAT Set Up

Before using *Iptables* an effort was made to set up NAT through the tool *Iproute2*, "a collection of utilities for controlling TCP/IP networking and traffic control in Linux" [22]. Unfortunately the tool did not want to work on the device the way it was stated in *Iproute2* manual pages which was confirmed also by a post, with the date 21st August 2009, on the website [16] by the

statement: "(CONFIG_IP_ROUTE_NAT) controlled by ip rule/ip route has been deprecated for quite a long time now in 2.6 kernels". A post on the mailing list [24] had the same opinion. According to [16] the tool *Traffic Control*-`tc`, for manipulating traffic control settings, should be used instead, but it was decided to try the *Iptables* first, which worked happily.

J Results

This appendix is a collection of tables, filled with data gathered during *KMedia*–DUT performance measurements, and some figures which were generated according to the data in the tables. Some more figures created with this data can be found also in Chapter 6 which references this appendix.

Message size is always reported in Bytes B, throughput or data rate either in Messages Per Second (m/s) or Megabits Per Second (Mbit/s) units. A percentage term [%] denotes the processor utilization, with 100% being the maximum, and, lastly, all latencies in the tables are reported in nanoseconds [ns] (10^{-9} seconds). For place reasons sometimes only the units are shown at the top of the relevant column.

It is important to note, that *the encryption and decryption data rates expressed in Mbit/s cannot by directly compared.* The bits per second were computed by *Netperf* according to the sizes of messages that were transmitted to the DUT, independent from the sizes of messages that came back from DUT. Since encryption was tested by sending plaintext messages to DUT and decryption by sending ciphertext messages and messages in ciphertext are longer than in plaintext state, it follows that even if the same number of messages would be successfully forwarded by DUT the results would show more bits per second during the decryption measurement. For this reason, to compare decryption with encryption performance only the Messages Per Second (m/s) unit should be taken for consideration.

To save some place in the tables and in the figures following abbreviations were used: **TP** as "throughput" (the maximum data rate), **Msg** stands for "message", **enc** for "encryption", **dec** for "decryption" and finally **Tal** stands for the "*Talitos*" driver and the underlying SEC which it controls.

The following Table J.1 is a cross reference showing in what figures the data from the tables were used. To each table or figure label also a page number in the parenthesis is provided to find the relevant objects easily. Since some figures were generated according to data from more than one table they are mentioned multiple times.

Table(s)	Figure(s)	Notes to the Tables
J.2(p.151)	6.1(p.68), 6.2(p.69)	"-" denotes that the measurement was not performed.
J.3(p.152), J.4(p.153)	6.3(p.72), 6.4(p.73)	
J.5(p.154)	6.5(p.74), 6.8(p.79)	
J.6(p.155)	6.6(p.76), 6.7(p.77), 6.8(p.79)	"num test msgs" column denotes the number of messages incorporated in the 30 seconds lasting measurements. "sk2sk", "tfm" and "delta" (explained on the page 104 in Section A.5.3) were converted from clock cycles into nanoseconds.
J.7(p.156), J.8(p.157), J.9(p.158)	6.9(p.81), 6.10(p.82), J.1(p.159), 6.8(p.79)	"-" denotes that the measurement was not performed and "x" indicates insufficient system resources to obtain the data.

Table J.1: Relation of Tables and Figures

	NAT Forwarding				KMedia Forwarding			
Message Size[B]	TP[Mbit/s]	TP[m/s]	CPU-Looper[%]	CPU-Top[%]	TP[Mbit/s]	TP[m/s]	CPU-Looper[%]	CPU-Top[%]
29	10.17	44021	87.52	1.40	3.45	14859	99.66	99.30
86	26.25	38151	81.04	1.37	10.07	14628	99.59	98.88
214	61.81	36103	78.33	1.50	24.69	14422	99.50	99.38
348	79.70	28625	76.01	1.43	37.99	13644	99.69	99.28
470	84.59	22499	60.01	1.15	49.26	13101	99.61	96.93
726	88.93	15311	42.86	0.75	72.29	12446	99.64	99.25
982	90.91	11572	34.56	0.65	89.98	11453	99.71	97.48
1238	93.00	9390	29.74	0.52	93.13	9403	99.59	82.00
1458	94.46	8098	28.70	0.55	93.75	8037	96.59	76.39
1472	93.83	7968	27.12	0.52	93.91	7974	97.39	75.21
1476	-	-	-	-	80.11	6784	99.71	98.26
2048	-	-	-	-	92.29	5633	99.49	90.03
3072	-	-	-	-	92.49	3764	92.92	65.73
4096	-	-	-	-	93.44	2852	77.48	53.05
6144	-	-	-	-	93.50	1902	66.39	38.18
8192	-	-	-	-	94.22	1438	60.46	35.84
12288	-	-	-	-	94.03	957	55.65	34.31
16384	-	-	-	-	94.29	719	53.42	31.16
24576	-	-	-	-	94.52	481	51.61	31.64
32768	-	-	-	-	94.30	360	49.53	30.00
49152	-	-	-	-	94.41	240	47.26	31.06
65403	-	-	-	-	94.44	181	46.41	30.17

Table J.2: Throughput (TP) and CPU Load at NAT and *KMedia* Forwarding

SHA1-AES

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	10.15	5930	99.58	10.21	5964	93.03	12.39	5957	99.59	12.92	6210	97.58
1238	31.60	3191	99.57	50.20	5068	91.26	33.44	3255	99.56	52.55	5116	92.18
4096	40.34	1231	99.59	89.61	2735	92.76	41.89	1267	99.66	90.50	2737	92.07
32768	47.04	179	99.56	94.30	360	52.89	48.42	184	99.91	94.57	360	52.10

SHA1-DES3

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	8.12	4744	99.57	10.15	5931	92.40	9.14	4681	99.60	12.63	6471	98.22
1238	16.53	1668	99.55	51.06	5155	90.35	16.91	1666	99.55	51.62	5088	91.88
4096	18.47	563	99.54	89.13	2720	91.40	18.67	565	100	89.81	2722	91.61
32768	19.62	74	99.53	94.29	359	52.61	19.68	75	100	94.47	360	52.63

SHA256-AES

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	8.75	5114	99.58	11.01	6431	98.19	11.02	5139	99.59	15.26	7116	100
1238	27.87	2813	99.56	50.45	5093	92.47	29.27	2831	99.57	52.88	5116	93.38
4096	36.61	1117	99.58	89.66	2734	93.15	38.15	1151	99.67	91.13	2751	93.48
32768	42.90	163	99.56	94.30	359	52.87	44.49	169	100	94.59	360	52.24

SHA256-AES256

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	8.41	4913	99.58	10.81	6312	97.67	10.74	5011	99.58	15.26	7117	99.95
1238	25.55	2580	99.56	50.45	5093	92.09	27.33	2644	99.56	52.88	5115	93.71
4096	33.20	1013	99.57	89.61	2734	92.92	34.82	1051	99.53	91.09	2750	93.16
32768	38.69	147	99.56	94.28	359	52.57	40.20	153	100	92.42	359	52.46

SHA256-DES3

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	7.11	4155	99.57	10.03	5858	91.25	8.42	4175	99.57	13.04	6467	98.67
1238	15.28	1542	99.55	50.98	5147	90.28	15.87	1555	99.55	51.94	5087	92.10
4096	17.62	537	99.55	89.85	2742	91.62	17.92	542	99.55	90	2722	91.59
32768	18.87	71	99.55	94.30	359	52.66	18.98	72	100	94.50	360	52.66

Table J.3: Throughput (Mbit/s and m/s) and CPU Load (%) of Various Algorithms at Various Message Sizes in Bytes (B), Part1/2

MD5-AES

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	11.01	6432	99.58	11.09	6475	98.24	13.14	6466	99.59	14.45	7108	100
1238	37.04	3740	99.58	50.97	5145	92.69	39.13	3827	99.57	52.02	5087	93.47
4096	49.10	1498	99.59	89.84	2741	93.09	50.39	1526	99.59	90.8	2750	93.33
32768	58.84	224	99.56	94.29	359	52.54	60.62	231	99.57	94.55	360	52.37

MD5-DES3

Msg	enc SW			enc Tal			dec SW			dec Tal		
B	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%	Mbit/s	m/s	%
214	8.58	5009	99.59	10.21	5963	92.58	9.54	5011	99.57	13.56	7119	99.77
1238	17.87	1803	99.55	51.41	5190	90.77	18.25	1808	99.55	51.36	5086	91.92
4096	20.15	615	99.56	89.89	2743	91.60	20.15	611	99.55	89.66	2721	91.71
32768	21.43	81	100	94.29	359	52.55	21.45	81	100	94.46	360	52.49

Table J.4: Throughput (Mbit/s and m/s) and CPU Load (%) of Various Algorithms at Various Message Sizes in Bytes (B), Part 2/2

Msg	KMedia Forwarding			enc SW SHA1-AES			enc Tal SHA1-AES			enc SW SHA256-DES3			enc Tal SHA256-DES3		
B	Mbit/s	m/s	CPU%	Mbit/s	m/s	CPU%	Mbit/s	m/s	CPU%	Mbit/s	m/s	CPU%	Mbit/s	m/s	CPU%
29	3.45	14859	99.66	1.66	7169	99.59	1.71	7375	99.71	1.39	5993	99.47	1.74	7485	100.00
86	10.07	14628	99.59	4.57	6643	99.58	4.97	7217	99.51	3.63	5276	99.40	5.08	7380	99.92
214	24.69	14422	99.50	10.15	5930	99.58	10.21	5964	93.03	7.11	4155	99.57	10.03	5858	91.25
348	37.99	13644	99.69	14.76	5301	99.54	16.14	5798	92.56	9.46	3396	99.57	16.02	5753	91.71
470	49.26	13101	99.61	18.47	4913	99.58	21.50	5719	92.17	10.98	2921	99.56	21.65	5756	91.74
726	72.29	12446	99.64	23.96	4125	99.58	31.48	5420	91.69	13.07	2250	99.56	31.63	5445	91.29
982	89.98	11453	99.71	28.12	3579	99.58	41.47	5279	92.72	14.38	1830	99.55	41.68	5305	91.17
1238	93.13	9403	99.59	31.60	3191	99.57	50.20	5068	91.26	15.28	1542	99.55	50.98	5147	90.28
1458	93.75	8037	96.59	32.36	2774	99.56	54.16	4643	91.59	15.61	1338	99.54	53.47	4584	88.08
1472	93.91	7974	97.39	32.49	2759	99.56	54.65	4640	91.94	15.61	1325	99.54	54.05	4590	88.07
2048	92.29	5633	99.49	33.89	2069	99.59	59.92	3657	92.09	16.17	987	99.57	59.46	3629	92.25
3072	92.49	3764	92.92	37.55	1528	99.58	74.13	3016	89.96	16.94	689	99.54	73.64	2996	90.01
4096	93.44	2852	77.48	40.34	1231	99.59	89.61	2735	92.76	17.62	537	99.55	89.85	2742	91.62
6144	93.50	1902	66.39	42.84	872	99.57	93.17	1896	82.53	18.12	368	99.55	93.15	1895	82.86
8192	94.22	1438	60.46	44.28	676	99.57	93.72	1430	72.52	18.38	280	99.55	93.68	1429	72.28
12288	94.03	957	55.65	45.49	463	99.57	93.86	955	64.46	18.62	189	99.54	93.74	953	64.70
16384	94.29	719	53.42	46.01	351	99.57	93.95	717	60.37	18.72	142	99.54	93.95	716	60.43
24576	94.52	481	51.61	46.73	238	99.56	94.29	480	55.33	18.85	95	99.54	94.29	479	54.80
32768	94.30	360	49.53	47.04	179	99.56	94.30	360	52.89	18.87	71	99.55	94.30	359	52.66
49152	94.41	240	47.26	47.28	120	99.53	94.41	240	49.92	18.95	48	99.54	94.41	240	49.38
65403	94.44	181	46.41	47.57	91	99.59	94.44	181	48.08	19.03	36	99.55	94.44	180	48.36

Table J.5: Throughput (Mbit/s and m/s) and CPU Load (%) of SHA1-AES and SHA256-DES3

	SHA1-AES Encryption						SHA1-AES Decryption						
Msg	SW			Talitos			SW			Talitos			num
B	sk2sk	tfm	delta	sk2sk	tfm	delta	sk2sk	tfm	delta	sk2sk	tfm	delta	test msgs
29	51019	46970	4050	50756	46316	4441	51031	46777	4254	53516	49113	4403	155079
86	60698	56236	4462	50840	46208	4632	60719	56178	4541	52862	48226	4636	104555
214	76619	72126	4493	50956	46319	4637	75836	71317	4519	52933	48303	4630	84159
348	92217	87866	4351	51180	46637	4543	91032	86714	4319	53641	49107	4534	103591
470	108220	103708	4512	52485	47853	4632	106268	101706	4562	54033	49406	4628	76775
726	140055	135538	4518	56410	51760	4650	137099	132699	4400	56805	52181	4624	74595
982	172277	167817	4460	60307	55659	4648	167882	163473	4409	60274	55635	4639	73300
1238	217999	213442	4557	64354	59713	4642	204926	200341	4586	64284	59652	4633	75702
1472	235857	231462	4395	68563	64138	4425	230576	225799	4777	68069	63529	4539	63797
2048	338766	334407	4359	79008	74393	4615	315737	311512	4225	78834	74314	4520	59952
3072	474397	469811	4586	95491	90703	4788	453696	448481	5215	95724	90857	4866	42849
4096	620312	615467	4845	112175	107249	4925	594764	589562	5201	112421	107463	4958	33338
6144	907594	902311	5282	145302	140114	5188	862245	856730	5516	144760	139603	5157	25003
8192	1187276	1181475	5801	178180	172605	5575	1126543	1120165	6378	177795	172349	5445	19362
12288	1730074	1724044	6030	242701	236959	5742	1659106	1651961	7145	243119	237524	5596	13338
16384	2290190	2283126	7065	306299	300339	5960	2194637	2187990	6646	306864	301182	5682	10173
24576	3436435	3430048	6387	433352	427192	6160	3296086	3289230	6856	433727	427956	5772	7589
32768	4555341	4548408	6932	560711	554443	6268	4409861	4402775	7086	561006	555192	5814	5219
49152	6800793	6793022	7771	814990	808549	6441	6561586	6554588	6999	815646	809652	5994	3531
65403	8878827	8871687	7140	1066835	1060513	6322	8592505	8585285	7218	1066110	1059984	6126	2501

Table J.6: Encryption and Decryption Latencies in **Nanoseconds** of SW– and Talitos–Driver for SHA1-AES Transform

	29 Byte Message			86 Byte Message			214 Byte Message			348 Byte Message		
Mbit/s	KMedia	SW	Talitos	KMedia	SW	Talitos	KMedia	SW	Talitos	KMedia	SW	Talitos
0.1	5.49	8.44	7.62	2.18	3.35	2.87	0.91	1.49	1.22	0.57	0.98	0.76
0.2	10.66	16.90	14.49	4.18	6.52	5.48	1.76	2.90	2.30	1.08	1.86	1.41
0.3	15.59	25.01	21.33	5.88	9.41	7.95	2.63	4.34	3.44	1.61	2.80	2.10
0.4	20.46	33.07	28.44	7.55	12.29	10.17	3.50	5.77	4.57	2.15	3.76	2.81
0.8	39.58	64.46	54.77	14.20	23.59	19.37	6.22	10.78	8.36	4.17	7.36	5.48
1.2	58.64	92.80	80.20	20.65	34.88	28.51	8.91	15.74	12.15	5.88	10.74	7.85
2.4	96.93	x	x	40.45	68.76	56.12	16.96	30.53	23.39	10.88	20.70	14.83
4.8	x	x	x	78.41	x	99.39	33.17	60.25	45.89	20.91	40.56	28.89
9.6	x	x	x	99.51	x	x	64.51	99.45	87.49	40.94	78.29	56.80
14.4	x	x	x	x	x	x	92.65	x	x	60.53	99.43	83.50
19.2	x	x	x	x	x	x	x	x	x	78.45	x	x
25	x	x	x	x	x	x	x	x	x	x	x	x
30	x	x	x	x	x	x	x	x	x	x	x	x
35	x	x	x	x	x	x	x	x	x	x	x	x
40	x	x	x	x	x	x	x	x	x	x	x	x
45	x	x	x	x	x	x	x	x	x	x	x	x
50	x	x	x	x	x	x	x	x	x	x	x	x
60	x	x	x	x	x	x	x	x	x	x	x	x
70	x	x	x	x	x	x	x	x	x	x	x	x
80	x	x	x	x	x	x	x	x	x	x	x	x
90	x	x	x	x	x	x	x	x	x	x	x	x

Table J.7: CPU Load in [%] During *KMedia* Forwarding and SHA1-AES Encryption at Various Data Rates (Mbit/s) and Message Sizes Using SW– or *Talitos*–Driver, Part 1/3

	470 Byte Message			726 Byte Message			982 Byte Message			1238 Byte Message		
Mbit/s	KMedia	SW	Talitos	KMedia	SW	Talitos	KMedia	SW	Talitos	KMedia	SW	Talitos
0.1	0.43	0.79	0.56	0.27	0.56	0.36	0.21	0.45	0.27	0.17	0.40	0.22
0.2	0.83	1.53	1.11	0.55	1.10	0.72	0.42	0.91	0.55	0.33	0.79	0.44
0.3	1.23	2.27	1.62	0.82	1.65	1.08	0.63	1.37	0.82	0.50	1.17	0.66
0.4	1.61	2.97	2.12	1.09	2.20	1.44	0.83	1.82	1.11	0.66	1.56	0.88
0.8	3.23	5.99	4.27	2.17	4.38	2.86	1.65	3.63	2.19	1.32	3.12	1.74
1.2	4.58	8.67	6.12	3.24	6.57	4.26	2.47	5.43	3.25	1.97	4.67	2.61
2.4	4.00	16.71	11.54	5.94	12.57	8.00	4.70	10.63	6.29	3.94	9.37	5.23
4.8	16.00	32.68	22.17	11.07	24.27	15.07	8.57	20.29	11.81	7.13	17.97	9.64
9.6	30.99	63.82	43.26	21.21	47.61	29.39	16.26	39.74	22.56	13.47	35.14	18.48
14.4	46.06	93.83	63.67	31.39	70.94	43.39	23.88	59.22	33.09	19.71	52.34	27.11
19.2	59.10	x	82.93	41.46	94.40	57.28	31.70	77.86	43.99	26.04	69.35	36.02
25	75.96	x	x	53.01	x	73.80	40.88	98.92	56.07	33.49	90.64	46.64
30	x	x	x	62.35	x	88.83	48.51	x	67.47	40.02	99.65	54.93
35	x	x	x	71.99	x	x	55.37	x	78.94	46.24	x	64.29
40	x	x	x	x	x	x	63.82	x	91.34	52.10	x	74.43
45	x	x	x	x	x	x	73.20	x	x	58.28	x	84.73
50	x	x	x	x	x	x	x	x	x	67.26	x	91.81
60	x	x	x	x	x	x	x	x	x	80.63	x	x
70	x	x	x	x	x	x	x	x	x	x	x	x
80	x	x	x	x	x	x	x	x	x	x	x	x
90	x	x	x	x	x	x	x	x	x	x	x	x

Table J.8: CPU Load in [%] During *KMedia* Forwarding and SHA1-AES Encryption at Various Data Rates (Mbit/s) and Message Sizes Using SW– or *Talitos*–Driver, Part 2/3

	1472 Byte Message			4096 Byte Message			32768 Byte Message		
Mbit/s	KMedia	SW	Talitos	KMedia	SW	Talitos	KMedia	SW	Talitos
0.1	0.15	0.37	0.20	-	-	-	-	-	-
0.2	0.29	0.75	0.40	-	-	-	-	-	-
0.3	0.44	1.12	0.60	-	-	-	-	-	-
0.4	0.58	1.49	0.81	-	-	-	-	-	-
0.8	1.15	2.99	1.61	-	-	-	-	-	-
1.2	1.71	4.47	2.40	1.06	3.22	1.28	0.61	2.58	0.65
2.4	3.39	8.89	4.78	-	-	-	-	-	-
4.8	6.26	17.19	8.97	4.16	12.87	5.12	2.46	10.33	2.59
9.6	11.69	33.56	17.11	8.11	25.42	10.03	4.91	20.65	5.18
14.4	17.07	49.92	25.04	11.79	37.77	14.58	7.40	30.98	7.76
19.2	22.55	66.36	33.31	15.47	50.08	19.22	9.81	41.29	10.34
25	29.11	86.38	43.15	19.90	65.03	24.85	12.78	53.74	13.47
30	34.81	98.69	51.31	23.69	78.45	29.46	15.33	64.56	16.15
35	40.24	x	60.10	27.58	92.59	34.34	17.89	75.85	18.83
40	45.34	x	68.36	31.24	99.58	39.10	20.45	86.47	21.43
45	50.53	x	78.24	35.04	x	44.34	23.04	96.97	24.10
50	56.33	x	87.70	38.53	x	48.04	25.36	x	26.98
60	68.36	x	x	46.14	x	60.29	30.25	x	32.15
70	x	x	x	54.88	x	72.48	35.13	x	38.92
80	x	x	x	65.27	x	83.21	41.56	x	44.19
90	x	x	x	74.35	x	x	46.99	x	50.24

Table J.9: CPU Load in [%] During *KMedia* Forwarding and SHA1-AES Encryption at Various Data Rates (Mbit/s) and Message Sizes Using SW– or *Talitos*–Driver, Part 3/3

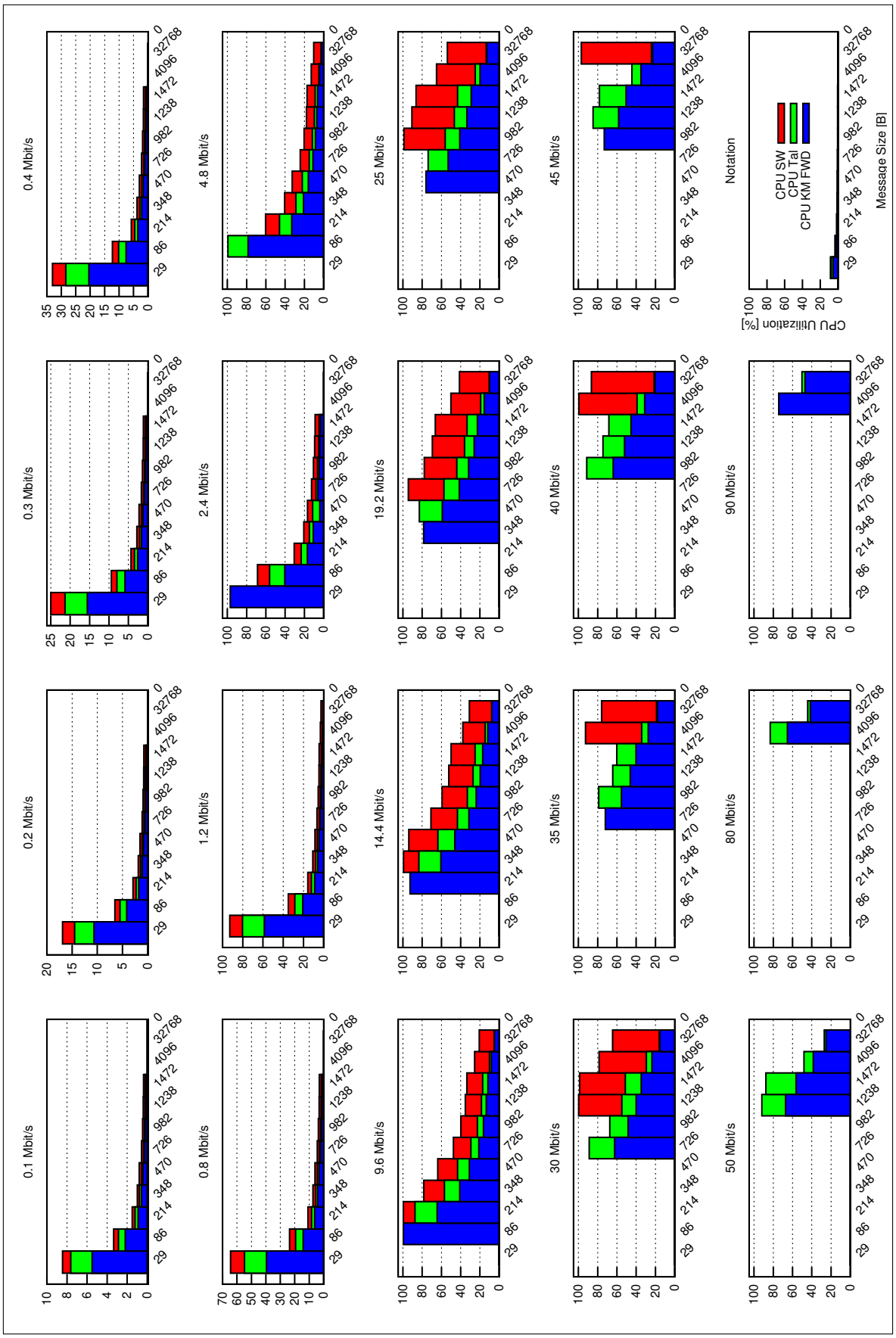

Figure J.1: Influence of Message Size on the CPU Load at Constant Data Rate (Measured at *KMedia* Forwarding and SHA1-AES Encryption with SW– and *Talitos*–Driver)

Literature References

[AAV06] Antonio Vincenzo Taddeo, Alberto Ferrante, and Vincenzo Piuri. Scheduling Small Packets in IPSec-based Systems. In *CCNC 2006*, Las Vegas, NV, USA, 8 January 2006. IEEE.

[ACL+04] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman. MIKEY: Multimedia Internet KEYing. RFC 3830, Internet Engineering Task Force, August 2004. Available at `http://www.rfc-editor.org/rfc/rfc3830.txt`.

[AVJ05] Alberto Ferrante, Vincenzo Piuri, and Jeff Owen. IPSec Hardware Resource Requirements Evaluation. In *NGI 2005*, Rome, Italy, 18 April 2005. EuroNGI.

[AWK09] Andre L. Alexander, Alexander L. Wijesinha, and Ramesh Karne. An evaluation of secure real-time transport protocol (srtp) performance for voip. volume 0, pages 95–101, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[BBR02] Roberto Barbieri, Danilo Bruschi, and Emilia Rosti. Voice over ipsec: Analysis and solutions. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 261, Washington, DC, USA, 2002. IEEE Computer Society.

[Ben05] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2005.

[BGV97] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Sha: a design for parallel architectures? In *EUROCRYPT'97: Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, pages 348–362, Berlin, Heidelberg, 1997. Springer-Verlag.

[Bla05] John Black. Authenticated encryption. In *Encyclopedia of Cryptography and Security*. Springer, 2005.

[BM99] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544, Internet Engineering Task Force, March 1999. Available at `http://tools.ietf.org/html/rfc2544`.

[BMN+04] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711, Internet Engineering Task Force, March 2004. Available at `http://www.rfc-editor.org/rfc/rfc3711.txt`.

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008. Springer-Verlag New York, Inc.

[Bra89] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, Internet Engineering Task Force, October 1989. Available at `http://www.rfc-editor.org/rfc/rfc1122.txt`.

[Bra91] S. Bradner. Benchmarking Terminology for Network Interconnection Devices. RFC 1242, Internet Engineering Task Force, 1991. Available at `http://tools.ietf.org/html/rfc1242`.

[CFP04] Fabien Castanier, Alberto Ferrante, and Vincenzo Piuri. A packet scheduling algorithm for ipsec multi-accelerator based systems. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference*, pages 387–397, Washington, DC, USA, 2004. IEEE Computer Society.

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 453–474, London, UK, 2001. Springer-Verlag.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition.* O'Reilly & Associates, 2005. The book is available at `http://lwn.net/Kernel/LDD3`.

[DTB08] Wafaa Bou Diab, Samir Tohme, and Carole Bassil. Vpn analysis and new perspective for securing voice over vpn networks. In *ICNS '08: Proceedings of the Fourth International Conference on Networking and Services*, pages 73–78, Washington, DC, USA, 2008. IEEE Computer Society.

[FGK03] S. Frankel, R. Glenn, and S. Kelly. The AES-CBC Cipher Algorithm and Its Use with IPsec. RFC 3602, Internet Engineering Task Force, September 2003. Available at `http://www.rfc-editor.org/rfc/rfc3602.txt`.

[FPC05] Alberto Ferrante, Vincenzo Piuri, and Fabien Castanier. A qos-enabled packet scheduling algorithm for ipsec multi-accelerator based systems. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 221–229, New York, NY, USA, 2005. ACM.

[Fre05a] Freescale Semiconductor. *HW Getting Started Guide–MPC8349E MDS Processor Board*, eighth edition, May 2005. Available at `http://www.freescale.com`. Document Number: MPC8349EMDSUMAD.

[Fre05b] Freescale Semiconductor. *MPC8349E PowerQUICC II Pro Integrated Host Processor Family Reference Manual*, first edition, August 2005. Available at `http://www.freescale.com`. Document Number: MPC8349ERM.

[fre08] Understanding Cryptographic Performance. White Paper Rev. 3, Freescale Semiconductor, August 2008. Available at `http://www.freescale.com`. Document Number: CRYPTOWP.

[fre09a] MPC8349E PowerQUICC II Pro Integrated Host Processor Hardware Specifications. Technical Data Rev. 11, Freescale Semiconductor, February 2009. Available at `http://www.freescale.com`. Document Number: MPC8349EEC.

[fre09b] Network/Embedded Processors. Product Listing Rev. 0, Freescale Semiconductor, April 2009. Available at `http://www.freescale.com`. Document Number: SG1007Q22009.

[FS03] Niels Ferguson and Bruce Schneier. A Cryptographic Evaluation of IPsec. Technical report, Counterpane Internet Security, Inc, December 2003. Available at `http://www.schneier.com/paper-ipsec.html`.

[Jon07] Rick Jones. *Care and Feeding of Netperf (Versions 2.4.3 and Later).* Hewlett-Packard Company, 2007. Available at `http://www.netperf.org/svn/netperf2/tags/netperf-2.4.5/doc/netperf.pdf`.

[Kau05] C. Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC 4306, Internet Engineering Task Force, December 2005. Available at `http://www.rfc-editor.org/rfc/rfc4306.txt`.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Internet Engineering Task Force, February 1997. Available at `http://www.rfc-editor.org/rfc/rfc2104.txt`.

[Ken05] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, Internet Engineering Task Force, December 2005. Available at `http://www.rfc-editor.org/rfc/rfc4303.txt`.

[KH09] M. Kaeo and T. Van Herck. Methodology for Benchmarking IPsec Devices. Internet-Draft draft-ietf-bmwg-ipsec-meth-05, Internet Engineering Task Force, July 2009. Work in progress.

[KMS95] P. Karn, P. Metzger, and W. Simpson. The ESP Triple DES Transform. RFC 1851, Internet Engineering Task Force, September 1995. Available at `http://www.rfc-editor.org/rfc/rfc1851.txt`.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, Internet Engineering Task Force, December 2005. Available at `http://www.rfc-editor.org/rfc/rfc4301.txt`.

[KVW04] Tadayoshi Kohno, John Viega, and Doug Whiting. Cwc: A high-performance conventional authenticated encryption mode. In *Proceedings of FSE 2004, LNCS 3017*, pages 408–426. Springer-Verlag, 2004.

[LIR02] L. Lo Iacono and C. Ruland. Confidential multimedia communication in ip networks. In *ICCS '02: Proceedings of the The 8th International Conference on Communication Systems*, pages 516–523, Washington, DC, USA, 2002. IEEE Computer Society.

[Lov05] Robert Love. *Linux Kernel Development, Second Edition.* Sams Publishing, 2005.

[LP06] Todd Lumpkin and Kim Phillips. Linux, IPsec, and Crypto Hardware Acceleration. Technical report, Freescale Semiconductor, 2006. Available at `http://www.openswan.org/docs/local/rfcs/references.txt`.

[Man98] R. Mandeville. Benchmarking Terminology for LAN Switching Devices. RFC 2285, Internet Engineering Task Force, February 1998. Available at `http://tools.ietf.org/html/rfc2285`.

[Man07] V. Manral. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4835, Internet Engineering Task Force, April 2007. Available at `http://www.rfc-editor.org/rfc/rfc4835.txt`.

[McG09] D. McGrew. Authenticated Encryption with AES-CBC and HMAC-SHA1 (and other generic combinations of ciphers and MACs). Internet-Draft draft-mcgrew-aead-aes-cbc-hmac-sha1-01, Internet Engineering Task Force, March 2009. Work in progress.

[MD98] C. Madson and N. Doraswamy. The ESP DES-CBC Cipher Algorithm With Explicit IV. RFC 2405, Internet Engineering Task Force, November 1998. Available at `http://www.rfc-editor.org/rfc/rfc2405.txt`.

[MIK02] Stefan Miltchev, Sotiris Ioannidis, and Angelos D. Keromytis. A study of the relative costs of network security protocols. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–48, 2002.

[MV04] David A. McGrew and John Viega. The security and performance of the galois/-counter mode (gcm) of operation. In *In INDOCRYPT, volume 3348 of LNCS*, pages 343–355. Springer, 2004.

[NIS01] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001. Available at `csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[oST08] National Institute of Standards and Technology. FIPS 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS). Technical report, October 2008. Available at `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[PA98] R. Pereira and R. Adams. The ESP CBC-Mode Cipher Algorithms. RFC 2451, Internet Engineering Task Force, November 1998. Available at `http://www.rfc-editor.org/rfc/rfc2451.txt`.

[Pos81] J. Postel. Internet Protocol. RFC 0791, Internet Engineering Task Force, September 1981. Available at `http://www.rfc-editor.org/rfc/rfc791.txt`.

[PY05] Kenneth G. Paterson and Arnold K.L. Yau. Cryptography in theory and practice: The case of encryption in ipsec. Cryptology ePrint Archive, Report 2005/416, 2005. Available at `http://eprint.iacr.org/2005/416`.

[Rie10] Gerhard Rieger. *socat 1.7.1 – Manual Page*. dest-unreach, January 2010. Available at `http://www.dest-unreach.org/socat/doc/socat.html`.

[SGB08] S. Spinsante, E. Gambi, and E. Bottegoni. Security solutions in voip applications: State of the art and impact on quality. In *ISCE 2008. IEEE International Symposium on Consumer Electronics*. IEEE Computer Society, 2008.

[Shi00] R. Shirey. Internet Security Glossary. RFC 2828, Internet Engineering Task Force, May 2000. Available at `http://www.rfc-editor.org/rfc/rfc2828.txt`.

[SLW07] Marc Stevens, Arjen Lenstra, and Benne Weger. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In *EUROCRYPT '07: Proceedings of the 26th annual international conference on Advances in Cryptology*, pages 1–22, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sta04] William Stallings. *Data and Computer Communication, Seventh Edition.* Pearson Publication, 2004.

[Til05] Henk C.A. van Tilborg. *Encyclopedia of Cryptography and Security.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[TRC08] Janar Thoguluva, Anand Raghunathan, and Srimat T. Chakradhar. Efficient software architecture for ipsec acceleration using a programmable security processor. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 1148–1153, New York, NY, USA, 2008. ACM.

[YAM09] Chan Yeob Yeun and Salman Mohammed Al-Marzouqi. Practical implementations for securing voip enabled mobile devices. volume 0, pages 409–414, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

Web References

[1] Cevan O. *KMedia Website*, 2010. http://www.ict.tuwien.ac.at/sysari/KMedia.

[2] Crypto++ Library. *Crypto++ 5.6.0 Benchmarks*, 2010. http://www.cryptopp.com/benchmarks.html.

[3] DENX. *Root Filesystem for PowerPC 82xx*, 2002. ftp://ftp.denx.de/pub/LinuxPPC/usr/src/SELF/images/ppc_82xx/pRamdisk.

[4] Denys Vlasenko. *Busybox Homepage*, 2010. http://busybox.net.

[5] die.net. *Linux man pages*, 2010. http://linux.die.net/man.

[6] die.net. *packet(7) - Linux man page*, 2010. http://linux.die.net/man/7/packet.

[7] die.net. *raw(7) - Linux man page*, 2010. http://linux.die.net/man/7/raw.

[8] FH München. *Internet-Technologie*, 2004. http://www.netzmafia.de/skripten/server.

[9] The Free Software Foundation (FSF). *FSF Homepage*, 2010. http://www.fsf.org.

[10] Freescale. *SEC 2.x and SEC 3.x Security Core Device Driver Package*, 2009. http://www.freescale.com, Keyword: SEC23DRVRS.

[11] Freescale Semiconductor. *Freescale – Homepage*, 2009. www.freescale.com.

[12] Freeswan. *FreeS/WAN documentation*, 2003. http://www.freeswan.org/freeswan_trees/freeswan-2.05/doc/toc.html.

[13] G. Waters and K. Stammberger. *Understanding Crypto Performance in Embedded Systems: Part 2*, 2009. http://www.embedded.com/design/218400877.

[14] Gerhard Rieger. *Socat – Homepage*, 2009. www.dest-unreach.org/socat.

[15] Google. *Google Search Engine*, 2010. http://www.google.com.

[16] groupsrv.com. *Linux Forum–NAT using Iproute2*, 2009. http://www.groupsrv.com/linux/about156023.html.

[17] Hewlett-Packard Company. *Netperf – Homepage*, 2009. www.netperf.org.

[18] Hobbit hobbit@avian.org. *Netcat – README file*, 2009. http://nc110.sourceforge.net.

[19] IKEv2 Project. *Linux implementation of IPsec*, 2010. `http://ikev2.zemris.fer.hr/docs/linux/index.shtml`.

[20] kernel.org. *Linux Kernel Crypto on Git (Version Control System)*, 2010. `http://git.kernel.org/?p=linux/kernel/git/herbert/cryptodev-2.6.git`.

[21] linux-crypto@vger.kernel.org mailing list. *e-mail with Kim Phillips from Freescale*, 2009. `http://article.gmane.org/gmane.linux.kernel.cryptoapi/3472`.

[22] The Linux Foundation. *Iproute2 Homepage*, 2010. `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`.

[23] LXR Community. *LXR — the Linux Cross Reference*, 2010. `http://lxr.linux.no`.

[24] Mail. *Mailinglist–IP_ROUTE_NAT is broken*, 2004. `http://mailman.ds9a.nl/pipermail/lartc/2004q3/013674.html`.

[25] Mike Muuss. *Test TCP (TTCP)*, 2010. `http://ftp.arl.mil/~mike/ttcp.html`.

[26] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)–Home Page Archive*, 2000. `http://csrc.nist.gov/archive/aes/index.html`.

[27] National Institute of Standards and Technology. *NIST Comments on Cryptanalytic Attacks on SHA-1*, 2006. `http://csrc.nist.gov/groups/ST/hash/statement.html`.

[28] National Security Agency (NSA). *Security Enhanced Linux Mailing List: sock_create_kern()*, 2004. `http://www.nsa.gov/research/selinux/list-archive/0405/6969.shtml`.

[29] Netfilter Project Group. *Iptables Homepage*, 2010. `http://www.netfilter.org/projects/iptables`.

[30] NIST. *Block Cipher Modes*, 2010. `http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html`.

[31] Openswan. *IPsec related RFC's and drafts*, 2010. `http://www.openswan.org/docs/local/rfcs/references.txt`.

[32] Pascal Rigaux. *Hexadecimal Viewer and Editor*, 2009. `http://merd.sourceforge.net/pixel/hexedit.html`.

[33] Sourceforge. *Procps - The /proc file system utilities*, 2009. `http://procps.sourceforge.net`.

[34] Sourceforge. *OCF-Linux*, 2010. `http://ocf-linux.sourceforge.net`.

[35] Steve Friedl's Unixwiz.net Tech Tips. *An Illustrated Guide to IPsec*, 2010. `http://unixwiz.net/techtips/iguide-ipsec.html`.

[36] Steve Kemp- Debian Administration. *Use And Abuse of Pipes With Audio Data*, 2009. `www.debian-administration.org/articles/145`.

[37] Wikipedia. *Block cipher modes of operation*, 2010. `http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation`.