UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

SOFTWARE ENGINEERING

Abdi-Hakim Yasin Ararse

COMPARISON OF THE PERFORMANCE OF 3G SECURITY ALGORITHMS IN THE NAS LAYER

Master's thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, (15th March 2013).

Supervisor D.Sc. (Econ.) Jouni Lampinen

Instructor M.Sc. (Tech.) Jani Kokkonen

ACKNOWLEDGEMENTS

First I would like to express my deepest appreciation to Professor Jouni Lampinen and Jani Kokkonen for their guidance and valuable advice throughout this Master’s thesis work.

I am also thankful to Lab. Engineer Jari Kallio for helping and providing the tools required for the test setup.

Thanks to my family for their endless encouragement. Special thanks to my loving wife Shamsa Mohamed, and my beloved children Nawar and Muhammed. Without their mental support this thesis work would not have been completed on time.

Abdi-Hakim Yasin Ararse

Helsinki, Finland, 15th of March, 2013

TABLE OF CONTENTS

CONCEPTS

Cellular Concept: the cellular concept is a system-level idea which calls for
a single high-power transmitter (large cell) to be replaced by many low-power transmitters (small cells), each providing coverage to only a small portion of the service area.

Domain: the highest-level group of physical entities. Reference points are defined between domains.

Stratum: the grouping of protocols related to one aspect of the services provided by one or several domains.

Stream Cipher: the stream concept is that Plaintext data are added bit by bit to random-looking mask data that are generated by the Cipher Key (CK) and a few other parameters.

Processor: this word is used in this thesis to denote the entire chip with all of the different functional hardware blocks, including all the cores on the chip.

A Cryptographic Accelerator: is a device that performs processor-intensive decrypting/encrypting while freeing the host CPU to perform other tasks.

Message authentication code: Is a short piece of information used to authenticate a message and to provide integrity and authenticity assurance on the message.

LIST OF SYMBOLS

=	The assignment operator.
$\oplus$	The bitwise exclusive -OR operation.
$\|\|$	The concatenation of the two operands.

ABBREVIATIONS

1G	1st Generation
2G	2nd Generations
3G	3rd Generations
3GPP	3rd Generations Partnership Project
cnMIPS	core networks standardized MIPS
AKA	Authentication and Key Agreement
AN	Access Network
AMPS	Advanced Mobile Phone Service
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CK	Cipher Key
CN	Core Network
CPU	Central Processing Unit
CDMA	Code Division Multiple Access
CS	Circuit Switched
DES	Data Encryption Standard
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
IETF	Internet Engineering Task Force
IK	Integrity Key
I/O	Input and Output
HE	Home Environment
HW	Hardware
HSPA	High Speed Downlink Packet Access

HSPA+	Evolved High Speed Packet Access
HSUPA	High Speed Uplink Packet Access
K	Key
KM	Key Modifier
OFB	Output Feedback
PDA	Personal Digital Cellular
PS	Packet Switched
PS	Padded String
LTE	Long-Term Evolution
MAC	Message Authentication Code
MIPS	Microprocessor without Interlocked Pipeline Stages
MM	Mobility Management
MME	Mobility Management Entity
NAS	Non-Access Stratum
NIST	National Institute of Standard and Technology
Node B	UMTS Base Station
RAN	Radio Access Network
RANAP	Radio Access Network Application Protocol
RNC	Radio Network Controller
RRC	Radio Resource Control
SAGE	Security Algorithms Group of Experts
SN	Serving Network
SoC	System-On-Chip
SW	Software
TDMA	Time Division Multiple Access

TR	Technical Report
TS	Technical Specification
UE	User Equipment
UEA	UMTS Encryption Algorithm
UIA	UMTS Integrity Algorithm
UMTS	Universal Mobile Telecommunication System
USA	United State of America
USIM	Universal Subscriber Identity Module
UTRAN	UMTS Terrestrial Radio Access Network
WCDMA	Wideband Code Division Multiple Access

TABLE OF FIGURES

LIST OF TABLE

UNIVERSITY OF VAASA	
Faculty of technology	
Author:	Abdi-Hakim Yasin Ararse
Topic of the Thesis:	**Comparison of the performance of 3G security algorithms in the NAS layer**
Supervisor:	Professor Jouni Lampinen
Instructor:	M.Sc.(Tech). Jani Kokkonen
Degree:	Master of Science in Technology
Major of Subject:	Software Engineering
Year of Entering the University:	2000
Year of Completing the Thesis:	2013 **Pages:** 101
Key words:	KASUMI, 3G, UMTS security, Multi-Core

ABSTRACT

Cryptographic functionality implementation approaches have evolved over time, first, for running security software on a general-purpose processor, second, employing a separate security co-processor ,and third, using built-in hardware acceleration for security that is a part of a multi-core CPU system. The aim of this study is to do performance tests in order to examine the boost provided by accelerating KASUMI cryptographic functions on a multi-core Cavium OCTEON processor over the same non-accelerating cryptographic algorithm implemented in software.

Analysis of the results shows that the KASUMI SW implementation is much slower than the KASUMI HW-based implementation and this difference increases gradually as the packet size is doubled. In detailed comparisons between the encryption and decryption functions, the result indicates that at a lower data rate, neither of the KASUMI implementations shows much difference between encryption or decryption processing, regardless of the increase in the number of data packets that are being processed.

When all the 16 cores of the OCTEAN processor are populated, as the number of core increases, the number of processing cycles decreases accordingly. Another observation was that when the number of cores in use exceeds 5 cores, it doesn’t make much difference to the number of decrease of processing cycles.

This work illustrates the potential that up to sixteen cnMIPS cores integrated into a single-chip OCTEON processor provides for HW- and SW-based KASUMI implementations.

VAASAN YLIOPISTO	
Teknillinen tiedekunta	
Tekija:	Abdi-Hakim Yasin Ararse
Diplomityön nimi:	**Comparison of the performance of 3G security algorithms in the NAS layer**
Valvojan nimi:	Professor Jouni Lampinen
Ohjaajan nimi:	M.Sc.(Tech) Jani Kokkonen
Tutkinto:	Diplomi-insinööri
Koulutusohjelma:	Tietotekniikka
Suunta:	Ohjelmistotekniikka
Opintojen aloitusvuosi:	2000
Diplomityön valmistusmisvuosi:	2013 **Sivumäärä:** 101
Key words:	KASUMI, 3G, UMTS-Tietoturva, moniydin

TIIVISTELMÄ

Salaustekniikoiden toteutustavat ovat kehittyneet ajan myötä, kun ensiksi yleiskäyttöisillä suorittimilla ajettavista tietoturvaohjelmista on siirrytty käyttämään erillistä apusuoritinta ja kolmanneksi on siirrytty käyttämään sisäänrakennettua laitteistokiihdytystä, joka on osana keskusyksikön moniydinsuoritinta tietoturvan takaamiseksi. Tämän tutkimuksen tavoitteena on tehdä suorituskykytestejä, joilla tutkitaan suorituskyvyn parannusta, joka saavutetaan KASUMI-salausalgoritmin kiihdytyksellä moniytimisellä Cavium OCTEON suorittimella suhteessa ohjelmistopohjaiseen, kiihdyttämättömään salausalgoritmiin.

Tulosten analyysi osoittaa, että KASUMI-algoritmin ohjelmistopohjainen toteutus on paljon hitaampi kuin laitteistopohjainen toteutus, ja että tämä ero kasvaa asteittain, kun paketin koko kaksinkertaistuu. Yksityiskohtaisissa vertailuissa salauksen koodauksen ja koodauksen purkamisen välillä osoittavat, ettei alemmalla datanopeudella kummassakaan KASUMI- toteutuksessa salauksen tai sen purkamisen käsittelyssä ole suuria eroja riippumatta lisääntyvän käsiteltävien datapakettien koosta.

Kun kaikki kuusitoista OCTEON-suorittimen ydintä on otettu käyttöön, ja kun ydinten määrä kasvaa, suoritusvaiheiden määrä vähenee vastaavasti. Toinen havainto oli, että kun käytössä olevien ytimien määrä ylittää viisi ydintä, ei sillä ole vaikutusta suoritusvaiheiden vähenemiseen.

Tämä tutkimus osoittaa ne mahdollisuudet, jotka jopa 16 cnMIPS ydintä integroituna yhden piirin OCTEON-suorittimeen tarjoaa ohjelmisto- ja laitteistopohjaisissa toteutuksissa.

1. INTRODUCTION

This chapter introduces the motivation for comparing KASUMI hardware accelerated cryptography with the software-based KASUMI cryptography implementation, which has become the most used option in a system-in-chip embedded system like the OCTEON processor that is studied in this thesis. Afterwards, an overview of the organization of the thesis will be given.

The Universal Mobile Telecommunication System (UMTS) is a so-called third-generation (3G) mobile radio system and it is a successor to second-generation (2G) systems such as the Global System for Mobile Communications (GSM) and General Packet Radio Service (GPRS). The current UMTS system is in Release 8 and is called Long-Term Evolution (LTE), which represents a flat architecture solution.

According to the 3GPP security specification (3GPP TS 33.102, 2006), the security architecture is made up of a set of security features and security mechanisms that meets one or several security requirements. The requirements for UMTS confidentiality and integrity algorithms are specified by 3GPP in the technical specification document (3GPP TS 33.105, 2007). Within the security architecture of the 3GPP system, there are two mandatory security algorithms, namely, $f8$ and $f9$ or the UMTS Encryption Algorithm (UEA1) and UMTS Integrity Algorithm (UIA1), respectively. The core functionalities of these algorithms are based on the KASUMI stream cipher.

For the confidentiality algorithm, the UMTS Encryption Algorithm (UEA1) is a stream cipher and it is used to encrypt and decrypt blocks of data under a confidentiality key (*CK*). The algorithm uses KASUMI in the form of Output Feedback (OFB) mode as a key stream generator. For the integrity algorithm, the UMTS Integrity Algorithm (UIA1*)* computes the Message Authentication Code (MAC*)* of a given input message under an integrity key (IK) and imposes no limitation on the length of the input message. The approach that has been adopted uses KASUMI as it is also used by the confidentiality algorithm (UEA1) in the form of Cipher Block Chaining (CBC-MAC) mode.

Cryptography is the art and science of encrypting and decrypting data in order to make it impossible for outside parties to access the data. Cryptographic functionality implementation approaches have evolved over time, first, with software running on general-purpose processors, second, employing a separate custom security co-processor, and third, using built-in hardware acceleration for security that is a part of a multi-core CPU system.

This thesis examines the use of built-in hardware acceleration for cryptographic functions in silicon, or, to give it another name, cryptographic functions in SoC. The SoC system consists of both a hardware component and a software component, which paves the way for the boundary between hardware and software being shrunk. Security algorithms are generally dependent on intensive computation and they require many bit-manipulating operations in order to transform back and forth between plaintext and ciphertext.

Software running on a general-purpose processor is often inefficient in performing such operations since the many instructions needed to implement cryptographic operations consume valuable CPU resources and thus affect the performance of the system. Hardware acceleration provides a better system performance implementation than the software.

This thesis work focuses on verifying the cryptographic performance boost that is provided by a Cavium OCTEON processor as an accelerator of KASUMI encryptions and decryptions via the software-based KASUMI implementation. KASUMI encryption and decryption are applied in the UMTS Non-Access Stratum (NAS) layer for ciphering radio link access which is specified in the 3GPP release 7 specifications.

The structure of this thesis is as follows. Chapter 2 presents the existing literature work in the field under study. Chapter 3 discusses the basics of performance analysis and how it is applied in this thesis. Chapter 4 develops the relevant test scenarios and test environment setups. Chapter 5 discusses the test execution and presents the test results. Results and analysis of the measurements can be found in Chapter 6, followed by overall conclusions in Chapter 7.

2. UMTS

2.1 Introduction

This chapter will first provide an insight into the history of mobile telephony systems, from the first generation of mobile systems to the third generation. After taking a closer look at the evolution of mobile systems, the second subchapter will highlight the basic principle of mobile security used in the third-generation system. At the end, a detailed description of the KASUMI cipher and its operations will be given.

2.2 The Evolution of Mobile Telephony Systems

The history of mobile telephony goes back to experiments with radio telephony in the USA in the 1920s (Agar 2005). There are three different generations as far as mobile telephony is concerned. The first mobile radio systems, the so-called first generation (1G) networks, were based on an analogue radio path but used digital switching technology. These mobile systems offered basic services for their users and the emphasis was on speech and service-related matter. The other characteristics of these early mobile systems were that networks were mainly national efforts and very often they were specified after the networks were established. For this reason, the 1G networks were incompatible with each other.

Later, the first real cellular systems were implemented, such as the analogue Advanced Mobile Phone Service (AMPS) system in the USA. For the first time, frequencies were reused, resulting in the interference inherent to cellular networks. Old analogue systems have a common limitation in terms of wide area coverage and frequency spectrum reuse in the systems as a result of the use of only one single radio transmitter (Walke, Seidenberg &Althoff 2003).

In May 1972 Bell Labs introduced and patented a cellular concept that laid the foundations for the second- and third-generation mobile radio systems. The cellular concept is simple: instead of a single base station providing coverage of as large an area as possible, each base station should only cover a small area. Further elaboration of the cellular concept is provided in (Macdonald 1979) and (Rappaport 2002).

As the need for mobile communication increased, the need for a more global mobile communication system also increased. The international specification bodies started to specify the **2G**; the so-called Second Generation (2G) network uses digital channels, resulting in more efficient use of the spectrum. Two main systems that established themselves in the USA – the Time Division Multiple Access (TDMA) systems IS-54 and IS-136 – are based on a time slot structure and are similar to the Global System for Mobile Communications (GSM) (Althoff 2003) .

Japan also developed its own standard: Personal Digital Cellular (PDC), which also uses the TDMA technology (3 time slots, 25-MHz channel bandwidth) and operates at 800 MHz and 1500 MHz.

The best-known Second Generation system is one that originated in Europe; the Global System for Mobile (GSM) Communications was designed in the late ’80s by the state-owned national telecommunication companies and harmonised for use throughout Europe. GSM also employs the TDMA technology and uses 8 time slots on a 200-kHz wide carrier frequency. GSM900 has a total of 124 frequency channels and GSM1800 has as many as 374.

The so-called Third-Generation (3G) mobile radio systems are based on the open interfaces of its successor, GSM. The third generation, **3G**, is expected to complete the globalisation process of mobile communication. UMTS (Universal Mobile Telecommunication System) is one 3G implementation. The current UMTS network has been upgraded to High-Speed Downlink Packet Access (HSPA) in order to increase the data rate and capacity for downlink packets and has been introduced as a 3GPP release 5 features, and High-Speed Uplink Packet Access (HSUPA) is introduced in 3GPP release 6 in order to boost uplink performance in a UMTS network.

The combination of HSDPA and HSUPA is often referred to as HSPA. However, even with these improvements and the introduction of HSPA, the evolution of UMTS has not reached its end. In a 3GPP release 7, HSPA+ has been introduced, with a significant enhancement and improvement to the performance of HSPA-based radio networks in terms of spectrum efficiency, peak data rate, and latency.

2.3 UMTS Network Architecture

There are different ways to visualise a UMTS network, depending on which angle you look from. One angle to look from is the functions of the network in terms of how the traffic is handled. Another approach is to study the functions of the network elements. In this thesis work, the network will be looked at from the point of view of both the physical and functional viewpoints.

The physical aspects are modeled using the domain concept and the functional aspects are modeled using the strata concept (3GPP TS 23.110, 2007). Figure 1 (below) illustrates the basic domains in a UMTS system.

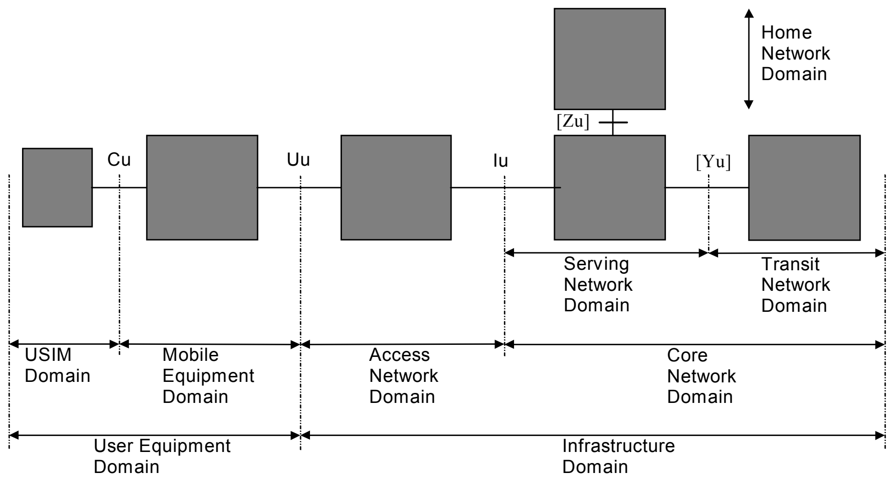


Figure 1. UMTS System

Cu	Reference point between USIM and UE
Iu	Reference point between Access and Serving Network domains
Uu	Reference point between User Equipment and Infrastructure domains, UMTS radio interface
[Yu]	Reference point between Serving and Transit Network domains
[Zu]	Reference point between Serving and Home Network domains

According to (3GPP TS 23.101, 2007), the basic UMTS architectural is split into a User Equipment domain and an Infrastructure domain. The main interest of this thesis work lies in the Infrastructure domain, which could be further split into an Access Network domain and a Core Network domain. From functionality point of view, UMTS network infrastructure is logically divided into Core Network (CN) and Access Network (AN) subsystems as specified in (3GPP TS 23.101, 2007) and (3GPP TS 23.110, 2007) respectively.

Each subsystem can be further divided into separate technologies. For example, the RAN (Radio Access Network) is compromised of different air interface technologies, such as GERAN (GSM, EDGE, and Radio Access Network), UTRAN (UMTS Terrestrial Radio Access Network), and future solutions such as WLAN and 4G.

The core network is today clearly divided into two domains:

- the Circuit-Switched (CS) domain and
- the Packet-Switched (PS) domain.

A more detailed description of the UMTS architecture is provided in the 3GPP Network Architecture document (3GPP TS 23.002, 2007). The interest of this thesis work lies mainly in two interfaces, namely Iu and Iur. More information on UMTS interfaces can be found in specifications (3GPP TR 23.930, 1999) and (3GPP TS 25.420, 2007).

2.4 UMTS Radio Network

The UMTS Terrestrial Radio Access Network (UTRAN) consists of a set of Radio Network Subsystems (RNS) connected to the Core Network (CN) through the Iu interface. A RNS consists of a Radio Network Controller and or more Node Bs. A Node B is connected to the RNC through the Iub interface (3GPP TS 25.401, 2002).

The main task of the UTRAN is to create and maintain Radio Access Bearers (RAB) for communication between the UMTS User Equipment (UE) and core network (CN). With RAB the CN elements are given an illusion about a fixed communication path to the UE, thus releasing them from the need to take care of radio communication aspects. There is two open interfaces: Uu and Iu, which are used by UTRAN to talk with the UE and CN. Since there are packet switched and circuit switched domains for different services, Iu-interface was separated into the Iu-CS and Iu-PS interfaces. Detail description of UTRAN protocol architecture is specified in (3GPP TS 25.401, 2002) document.

The UTRAN architecture is shown following figure.

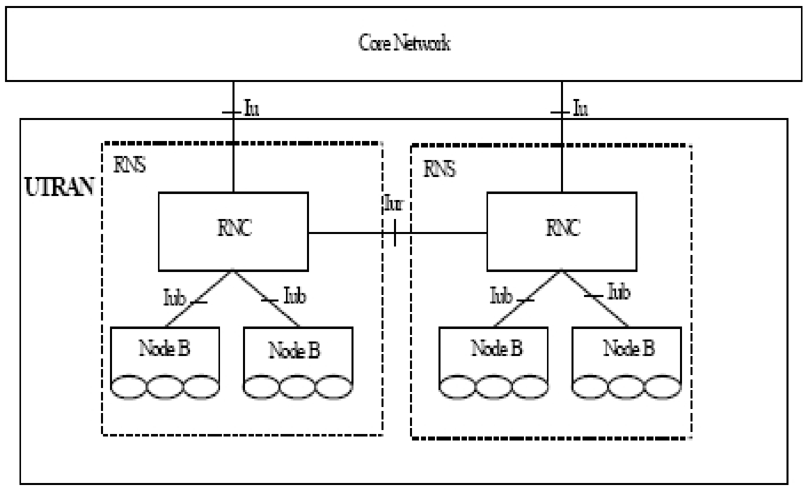


Figure 2. UTRAN, its networks and interfaces

The focus of this thesis work is on security parts of Non-Access Stratum (NAS) signaling between UE and CN which passes through UTRAN network. The overall protocol architecture of radio interface is layered into three protocol layers, namely, the physical layer (L1), the data link layer (L2) and network layer (L3) as described in (3GPP TS 25.301, 2007) . L3 provides Uu Stratum services and functions as whole and detail description of L3 protocol, Radio Resource Control (RRC) is given in (3GPP TS 25.331 2006).

2.4.1 Security Architecture UMTS

UMTS security builds on the security of GSM, inheriting the proven GSM security features. This maximizes the backward compatibility between GSM /UMTS and UMTS/LTE. UMTS also provides a solution to the weaknesses of GSM security and adds security features for new 3G radio access networks and services.

According to specifications, the security architecture is made up of a set of security features and security mechanisms (3GPP TS 33.102, 2006). A security feature is a service capability that meets one or several security requirements that are defined in (3GPP TS 21.133, 2002) and implement the security objectives and principles described in (3GPP TS 33.120, 2001).

A Security mechanism is an element that is used to realize a security feature and all the security features and security mechanism taken together to forms the security architecture (3GPP TS 33.102, 2006).

UMTS consists of five security feature groups:

I) Network Access Security provides users with secure access to UMTS services and protect against attacks on the radio access link.
II) Network Domain Security protects against attacks on the wireline network and allows nodes in the provider domain to exchange signaling data securely.
III) User Domain Security provides secure access to mobile stations.
IV) Application Domain Security allows the secure exchange of messages between applications in the user and in the provider domain.
V) Visibility and configurability of security allows the user to observe whether a security feature is currently in operation and if certain services depend on this security feature

Figure 3 shows the way security features are grouped together into five different sets of features, each one facing a specific threat and accomplishing certain security objectives.

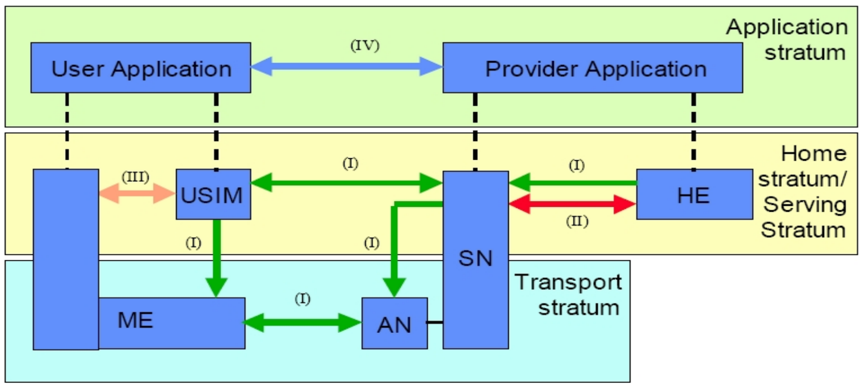


Figure 3. Overview of security architecture

The focus of this thesis work is on network access security (group **I**) and other UMTS security area are outside the scope of this thesis work.

2.4.2 Network Access Security to UMTS

Network Access Security features enables users to securely access services provided by the UMTS network. It consist a set of security features that provide users with secure access to UMTS services in which particular protect against on the radio access link. Network access security features can be further classified into the following categories, namely, user identity confidentiality, entity authentication and confidentiality, data integrity and mobile equipment identification as specified in (3GPP TS 33.102, 2006).

2.4.2.1 User Identity Confidentiality

The main objectives of the user identity confidentiality features are to provide mechanisms which prevent intruders from eavesdropping on the radio access link. It also makes sure that the user location confidentiality is secured, so the presence or the arrival of the user in a certain area cannot be determined by eavesdropping on the radio access. It also provides user un-traceability mechanism, so that an intruder cannot deduce whether different services are delivered to the same user by eavesdropping on the radio access link.

To achieve these objectives, the user is normally identified by a temporary identity by which he is known by the visited serving network, or by an encrypted permanently identity. To avoid user traceability, which may lead to the compromise of the user identity confidentiality, the user should not be identified for a long period by means of the same temporary or encrypted identity. To achieve these security features, in addition it is required that any signaling or user data that might reveal the user's identity is ciphered on the radio access link.

2.4.2.2 Mutual Entity Authentication

There are three entities involved in the authentication mechanism of the UMTS system, Home Environment (HE), Serving Network (SN) and the User Entity (UE), more specifically Universal Subscriber Identity Module (USIM). Authentication and Key Agreement (AKA) mechanism accomplishes this mutual authentication of the user and the network using a symmetric key (K) and derives the new cipher and integrity keys. Detail description of Authentication and Key Agreement (AKA) mechanisms are discussed in (Arkko & Haverinen 2006), (Kambourakis, Rouskas & Gritzalis 2004) and (Grecas, Maniatis & Vernieris 2003).

Once the user and the network have authenticated each other, they may begin secure communication. Encryption and decryption take place in the UE and in RNC on the network side, which means that Cipher Key (CK) has been transferred from the core network (CN) to the Radio Access Network (RAN). This is done in a specific Radio Access Network Application Protocol (RANAP) message, called the security mode command. After the RNC has obtained the CK, it can switch encryption on the sending a Radio Resource Control (RRC) security code command to the UE. The UMTS encryption mechanism is based on *a* steam cipher concept [see chapter concepts]. The core of encryption mechanism is the mask generation algorithm, which is denoted as function f8**.** The specification is available (3GPP TS 35.201, 2007) and it is based on a novel block cipher called KASUMI in Universal Mobile Telecommunication System (UMTS) and uses SNOW 3G in Long-Term Evolution (LTE).

2.4.2.3 Data Integrity

The purpose of integrity data protection is to provide a mechanism that the User Equipment (UE) and Serving Network (SN) can securely negotiate the integrity algorithm that they shall use subsequently in order to authenticate individual control messages. It also provides a mechanism that the User Equipment (UE) and Serving Network (SN) agree on an integrity key that they may use subsequently. Lastly, data integrity feature provide a mechanism that receiving entity (UE or SN) is able to verify that signaling data has not been modified in an authorized way since it was sent by the sending entity (UE or SN) and that data origin of the signaling data received is indeed the one claimed.

According to (Niemi & Nyberg 2006), this mechanism is important, since separate authentication procedures only give assurance of the identities of the communicating parties at the time of the authentication. A Message Authentication Code (MAC) function is applied to each individual signaling message at the RRC layer of UMTS Terrestrial Radio Access Network (UTRAN) protocol stack (3GPP TS 25.331 2006).

The integrity protection of signaling messages, between UE and RNC starts during the security mode set-up as soon as the integrity key and integrity protection algorithm is known. Integrity protection is based on a message authentication code concept [see in concept chapter] which is a one-way function controlled by the secret Integrity Key

(IK). The function is devoted by f9 and its output is MAC-I a 32 bit, random-looking bit string. The algorithm for integrity protection is based on the same core function as encryption.

This thesis work focus mainly on mechanisms used for confidentiality and data integrity security features, so mobile equipment identification and entity authentication related mechanisms are outside the scope of this thesis work.

2.4.3 Cryptographic Algorithms for UMTS

This section will highlight the cryptographic algorithms used in UMTS and explain their functionalities in detail. Further, this section presents the 3GPP UMTS security algorithm specifications and further descriptions of UMTS security functionalities provided by security experts, namely Valtteri Niemi and Kaisa Nyberg in their book: UMTS Security.

In cryptography, a block cipher is a symmetric key cipher which operates on fixed-length group of bits. The block cipher transforms a plaintext block of fixed lengths into a sequence of ciphertext blocks of equal length under the control of a secret key K**.** The operation of transforming a plaintext block into a ciphertext block is called encryption, and the operation of transforming a ciphertext block back to plaintext block is called decryption. A block cipher applies the encryption algorithms and key to an entire block of plaintext to obtain the ciphertext.

To encrypt messages longer than the block size or in another word in order to provide confidentiality for messages of arbitrary length, then a mode of operation is used. The four most widely known modes of operation were originally standardized by National Institute of Standard and Technology (NIST) for use with the Data Encryption Standard (DES) algorithm (NIST 1981).

A stream cipher is a cryptographic algorithm for encrypting plaintext similarly as to block ciphers but the plaintext is partitioned to sequence of blocks. The main difference between them is that in the block cipher, the current plaintext block is not taken as data input for cryptographic transformation. Instead, a string of bits, often called a “keystram” block, is generated independently of the current plaintext block and then combined with the plaintext using a simple operation, most commonly the ***XOR*** operation. Due to this functional difference, stream ciphers typically operate on plaintext blocks of shorten length, which can be just 1 bit or 1 byte.

The requirement for UMTS confidentiality and integrity algorithms are specified by 3GPP in the technical specification document (3GPP TS 33.105, 2007). Within the security architecture of the 3GPP system, there are standardized algorithms versions for UMTS confidentiality algorithm ***UEA1*** and integrity algorithm ***UIA1*** There are two versions for UMTS security algorithms: 1.0 and 1.1. Version 1.1 must be used.

In following sub-sections confidentiality and Integrity algorithms will be elaborated more as well as their structure and functionalities.

2.4.3.1 Confidentiality Algorithm

For data confidentiality of user data and signaling data, UMTS uses a cryptographic function called ($f8$). Confidentiality algorithm (UEA1) is a stream cipher is used to encrypt and decrypt blocks of data under a confidentiality key (CK). The block of data may be between 1 and 20, 000 bits long. The algorithm use KASUMI in a form of Output Feedback (OFB) mode as a key stream generator in UMTS and uses. The detail description of f8 algorithm is specified in (3GPP TS 35.201, 2007).

The 3GPP $f8$ stream cipher mode is not a standard stream cipher mode of operation of a block cipher. Examples of such standard modes are counter mode and OFB mode (NIST 1981). A counter mode keystream generator makes us of the generator that is updated for each new block and is taken as part of the input to the generator function. The $f8$ stream cipher mode can be seen as a combination of these two standard modes and makes use of prewhitening of feedback data (Niemi & Nyberg 2006).

In UMTS, The $f8$ algorithm makes use of the KASUMI key-dependent function, which operates on 64-bit data blocks and produces 64-bit blocks under control of a 128-bit key K. In LTE, $f8$ uses SNOW 3G as a keystream generator, which generates a sequence of 32-bit words under the control of a 128-bit key and a 128-bit initialization variable. The input parameters to $f8$ are the Cipher Key (CK), the time-dependent input (COUNT-C), the bearer identity (BEARER), the direction of the transmission (DIRECTION) and the length (LENGTH) of the plaintext.

The Cipher Key (CK) is renewed at every authentication process. *COUNT-C*, *BEREAR* and DIRECTION can be considered as initialization parameters as they are renewed for each keystream block. The time-dependent input COUNT-C also sent in cleartext and used as a synchronization parameter for the synchronous stream cipher. The input parameter LENGTH only affects the length of the keystream, not the actual bits in it. The exact structure of these parameters and further description of $f8$ algorithm operation it is specified in (3GPP TS 33.102, 2006) and in (3GPP TS 35.201, 2007).

Following figure illustrate the use of *ƒ*8 function to encrypt plaintext by applying a keystream using a bitwise XOR operation. The plaintext may be recovered by generating the same keysteam using the same input parameters and applying it to the ciphertext using a bitwise **XOR** operation.

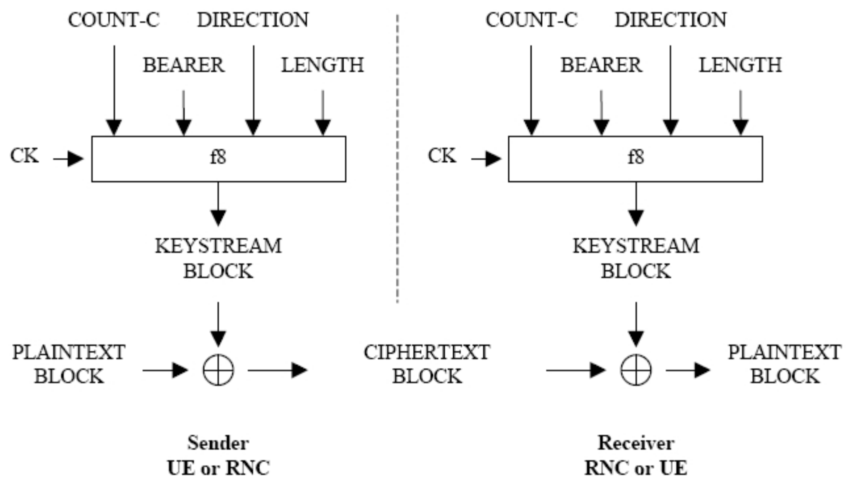


Figure 4. Ciphering user and signaling data transmitted over the radio access link.

The *ƒ*8 algorithm makes use of two 64-bit registers as described in (Niemi & Nyberg 2006): the static register A and the counter BLKCNT. Register A is initialized using the 64 bit initialization value:

IV = COUNT ||BEARER||DIRECTION||0…0 obtained as the concatenation of the 32-bit *COUNT*, 5-bit BEARER, 1-bit DIRECTION value and a string of 26 zero bits. The counter BLKCNT is set to 0.

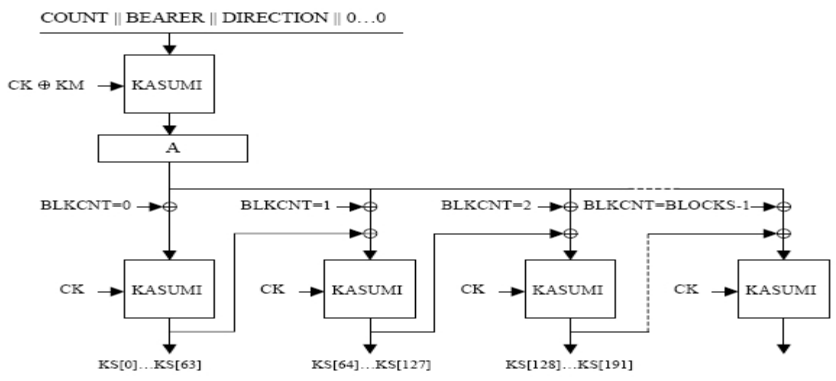


Figure 5. The *ƒ*8 stream cipher mode

The *ƒ*8 algorithm makes use of a Key Modifier (KM) constant that is equal to the octet 0 x 55 = 01010101 repeated 16 times. First, a single operation of KASUMI is applied to register A, using a modified version of the CK to compute the prewhiteninig value:

$$W = \text{KASUMICK} \oplus \text{KM(IV)} \tag{2.1}$$

which is stored in register A. Once the keystream generator has been initialized in this manner, it is ready to be used to generate keystream bits. The plaintext/ciphertext to be encrypted/decrypted consists of LENGTH bits, where LENGTH varies between 1 to 20,000 with granularity of 1 bit, while the keystream generator produces keystream bits in multiples of 64 bits. Between 0 and 63 of the least significant bits are discarded from the last block depending of the total number of bits required by LENGTH.

The number of required keystream bits is denoted by BLOCKS, whose value is determined by the value of the LENGTH parameter as follows: the value of LENGTH is divided by 64 and the result is rounded up to the nearest integer. The keystream blocks are denoted as KSB1, KSB2 ,…,KSBblocks. Set KSBo=0 and let *n* be an integer with $1 \leq n \leq$ BLOCKS, such that n = BLKCNT +1, and set:

$$\text{KSBn} = \text{KASUMICK}(\ W \oplus (n-1) \oplus \text{KSB}\ n-1) \tag{2.2}$$

Individual bits KS[0],KS[1],…….KS[LENGTH-1] of the keystream are extracted in turn from KSB1 to KSBblocks, with the most significant bit extracted first, by applying the following operation. For n = 1, …., BLOCKS and for each integer ***i***, with $\boldsymbol{0 \leq i \leq 63}$, set :

$$KS[((n-1) * 64) - i] = KSBn[i] \tag{2.3}$$

Encryption/decryption operations are identical and are carried out by the bitwise XOR of the input data using the generated keystream.

The main task with the design of *ƒ*8 was to make a steam cipher out of the block cipher KASUMI and there are several standard ways for this task, namely, cipher feedback (CFB), courter mode or output feedback (OFB) mode. For the function *ƒ*8 a combination of () and counter mode used for protection for KASUMI against chosen plaintext attack and protection against collision attacks (Günther, Howard & Niemi 2002).

To prevent chosen plaintext attacks the initialization vector (IV) is encrypted with different key CK′ = CK+KM, where KM is a key modifier. This initial encryption is also protection against collision attacks collision attacks as an attacker cannot freely choose the value which is XOR-ed with the block counter (Günther, Howard & Niemi 2002).

2.4.3.2 Integrity Algorithm

As specified in (3GPP TS 33.102, 2006), some Radio Resource Control (RRC), Mobility Management (MM) and Call Control (MM) signaling information elements are considered sensitive and must be integrity protected. Cryptographic integrity function (*f*9**)** shall be applied on certain signaling information elements transmitted between User Equipment (UE) and Serving Network (SN). For this task, UMTS Integrity Algorithm (UIA) has been specified and should be implemented in UE and in the RNC.

The UMTS Integrity Algorithm (UIA) shall be used with an Integrity Key (IK) to compute a message authentication code for a given message. At least the following signaling elements sent by the UE to the RNC should be protected:

- The UE capabilities, including authentication mechanism, ciphering algorithm and message authentication function capabilities.
- The security mode accept/reject message.
- The called party number in mobile originated call.
- Periodic message authentication messages.
- Various location updates, e.g. cell updates and URA updates

As for RNC concerned, at least following signaling message sent by RNC to the UE should be protected:

- The security mode command, including whether ciphering is enabled or not and the ciphering and integrity algorithm to be used.
- Periodic message authentication message

A cryptographic function (*ƒ*9) is used to protect data integrity and authenticate the data origin of signaling data at the RRC layer. A cryptographic message authentication algorithm generates a fixed length message authentication code (MAC) from a message of arbitrary length, under the control of the secret parameter key and a set of initialization values. The sender and receiver generate the MAC using same function.

3GPP integrity algorithm *ƒ*9 computes 32-bit Message Authentication Code (MAC*)* of a given input message under integrity key (IK) and imposes no limitation on the input message length. The approach adopted uses KASUMI as used by the confidentiality algorithm *ƒ*8 in form of Cipher Block Chaining (CBC-MAC) mode.

The input parameter to the integrity algorithm are the Integrity Key (IK), a time- and frame-dependent input (COUNT-I), a random value generated by the network side (FRESH), the direction bit (DIRECTION) and the signaling message (MESSAGE). The IK is cryptographic key that is newly generated at each authentication process. The COUNT-I, FRESH, and DIRECTION parameters can be considered as a set of initialization parameters that are renewed for each message to be authenticated. Based on these inputs parameters the user computes message authentication code for data integrity (MAC-I) using the UMTS Integrity Algorithm (UIA) *ƒ*9.

MAC-I is then appended to the message when sent over the radio access link. The receiver computes XMAC-I on the message received in the same way as the sender computed MAC-I on the message sent and verifies the data integrity of the message by comparing it the received MAC-I. The input parameter COUNT protects against replay during a connection. It is a value incremented at both sides of the radio access link every 10ms layer 1 frame. Its initial value is sent by the user to the network at connection set-up. The user stores the last used COUNT value from the previous connection and increments it by one. In this way the user is assured that no COUNT value is re-used (by the network) with the same integrity key.

The input parameter FRESH protects against replay of signaling message by the user. At connection set-up the network generates a random value FRESH and sent it to the user. The value FRESH is subsequently used by both the network and user throughout the during of a single connection. This mechanism assures the network that the user is not replaying and old MAC-Is.

Following figure is depicted derivation of MAC-I and/or XMAC-I on a signaling message:

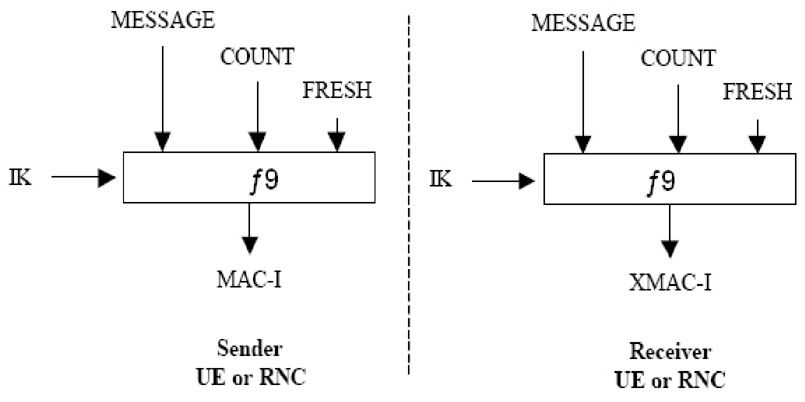


Figure 6. Derivation of MAC-I and/or XMAC-I o

The 3GPP standard *f*9 function (Niemi & Nyberg 2006) makes use of two 64-bit register***s*** A and B. The initial value for both registers is set equal to 0: A=0 and B = 0. The function also makes use of a constant value for a 128-bit Key Modifier (KM***)*** that is equal to 16 repetitions of the octet *0xAA = 10101010*. The inputs of the *f*9 function are as described above paragraphs, the value of all inputs are concatenated and then a single "1" bit is appended to this string, followed by between 0 and 63 "0" bits, so that the total length of the resulting string is an integer multiple of 64 bits. This string is called Padded String (PS) then:

```
PS = PSo||PS1||PS2|| . .|| PS BLOCKS -1                 (2.4)
```

This Padded String (PS***)*** is the data input to the Message Authentication Code (MAC) algorithm. It would also be possible to interpret the first block PS***o*** of PS as the initial value, since PSo = COUNT || FRESH, and this initial value would be different for each message. For each integer *n* with 0 ≤ n ≤ BLOCKS-1 the following operations are performed:

```
A = KASUMIIK(A⊕ PSn)                                     (2.5

B = B ⊕ A                                               (2.6)
```

Finally a further application of KASUMI is performed using a modified form of the Integrity Key (IK), as follows:

```
B = KASUMIIK ⊕ KM(B)                                    (2.7)
```

The output form KASUMI has 64 bits: MAC-I comprises the leftmost 32 bits of the result and the rightmost 32 bits are discarded.

Following figure show *f*9 integrity function

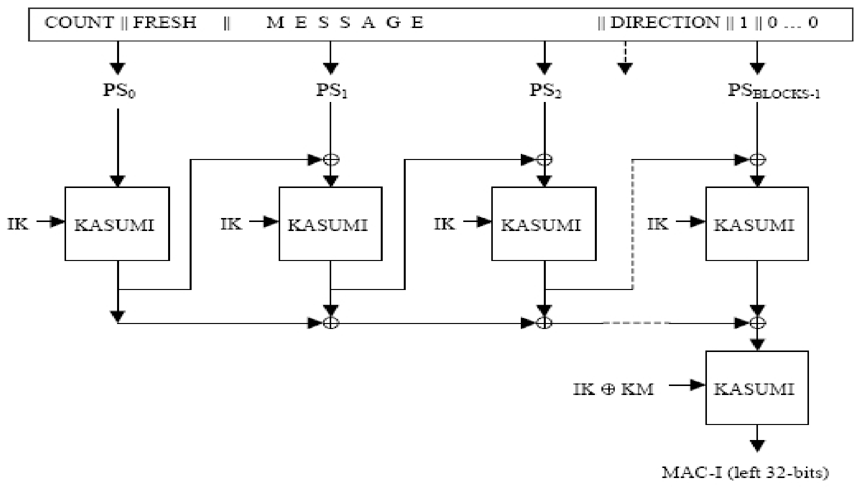

COUNT || FRESH || M E S S A G E || DIRECTION || 1 || 0 ... 0
PS0
PS1
PS2
PSBLOCKS-1
IK
KASUMI
IK ⊕ KM
MAC-I (left 32-bits)

Figure 7. The *f*9 integrity function

UIA negotiations, Integrity key lifetime, integrity protection procedures and local authentications are described in (3GPP TS 33.102, 2006) document.

2.5 KASUMI

This subchapter gives detail description of the KASUMI cipher. Most text of this subchapter is extracted from 3GPP specifications (3GPP TS 35.201, 2007) and (3GPP TS 35.202) handling of KASUMI and (Niemi & Nyberg 2006).

The KASUMI block cipher is used by the confidentiality algorithm (*f*8), and the integrity algorithm (*f*9) developed by Security Algorithms Group of Experts (SAGE) Technical Forum (TF) of 3GPP. As mentioned in subchapter 2.4, these algorithms were designed for specific use in the context of UMTS. They were designed for specific block cipher algorithm in mind, which was chosen as a starting point for what was going to be the kernel algorithm, a modified version of MISTY (Mitsure Matsui, 1997). In Parallel with the development of the *f*8 and *f*9 modes of operation, adjustments were also made to block cipher algorithm MISTY1 (Mitsure Matsui, 1997*)* and the final version of the block cipher algorithm is known as KASUMI—kasumi is Japanese for “hazy, dim blurred” (Niemi & Nyberg 2006).

2.5.1 Description of KASUMI

KASUMI (3GPP TS 33.102, 2006), is a 64-bit block cipher that has a key size of 128bits. KASUMI was designed as modification of MISTY1 (Mitsuru Matsui, 1997), optimized for implementation in hardware. Therefore, the most of the components of KASUMI are similar to the respective components of MISTY1 (Mitsuru Matsui, 1997).

KASUMI is a Feistel (3GPP TS 35.202, 2007) cipher with eight rounds. It operates on a 64-bit data block and uses a 128-bit key. The round function (or $fi()$ f-function) used in the ith round of the Feistel cipher is denoted by fi. The f-function has a 32-bit input and a 32-bit output. Each f-function of KASUMI is composed of two functions an FL-function and an FO-function. An FO-function is defined as a network that makes use of three applications of an FI-function.

An FI-function has a 16-bit input and a 16-bit output. Each FI-function comprises a network that makes use of two applications of a function S9 and two applications of a function S7. The functions S7 and S9 are also called “S-boxes of KASUMI”. In this manner KASUMI has similar three- layer nested structured of MISTY1 (Mitsuru Matsui, 1997) . In this manner KASUMI decomposes into a number of subfuntions (FL, FO and FI) that are used in conjunction with associated subkeys (KL, KO and KI). The outmost Feistel network comprises eight rounds, which are called in the specification outer rounds and numbered using index i, i = 1,2, ..., 8.

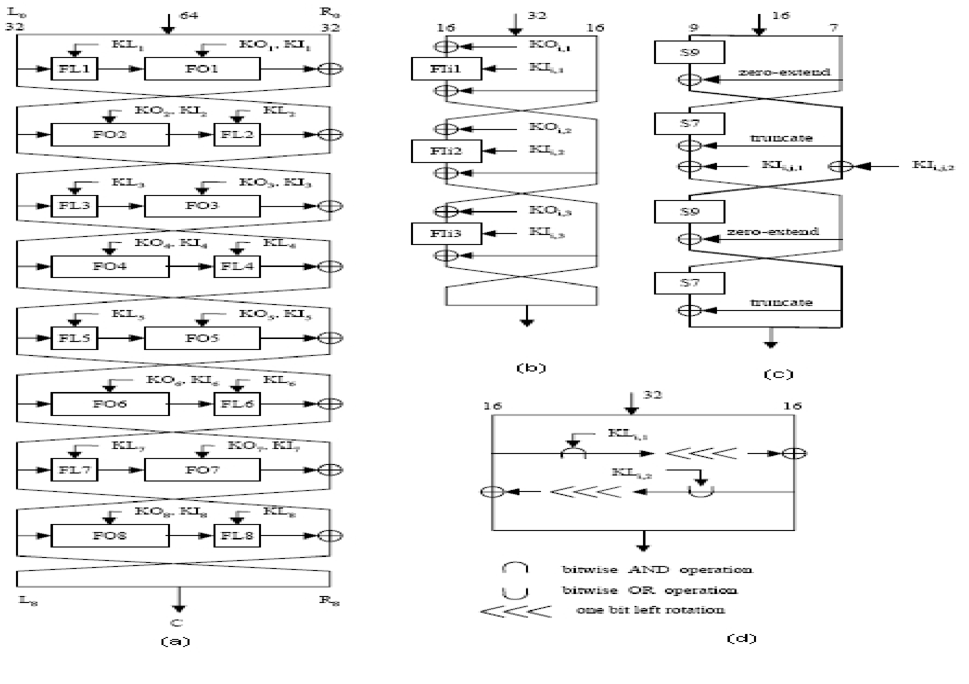


Figure 8. (a) KASUMI; (b) FOi function; (c) FIi,j function; and (d) FLi funtion

The FL-functions and *FO*-functions used at each round of the Feistel network are numbered accordingly (i.e FL_i, and FO_i are functions used at the ith round of the outer network). Function FL_i is used in conjunction with subkey KL_i, and function FO_i is used conjunction with two subkeys: KO_i and K_i.

The network formed by the eight FO-functions are called the inner networks and each one has three rounds indexed by *j, j=1, 2, 3*. Each round of an inner network makes use of a KO-key and an *FI*-function, the latter is used in conjunction with a KI-key. Consider the *i*th inner network FO_i. The KO-key, FI-function and the KI-key used at the *j*th round of FO_i are denoted as $KO_{i,j}$ $FI_{i,j}$ $KI_{i,j}$ respectively. In addition, the KI-key KIi,j splits into two halves $KI_{i,j\,1}$ and $KI_{i,j\,2}$.

2.5.2 KASUMI Components and Encryption Function

In *ƒ*8 and *ƒ*9 mode operation, the kernel function is only computed in one direction (Niemi & Nyberg, 2006). So even if the kernel function is a block cipher, the decryption transformation is never used. The purpose of the 3GPP is only the encryption function of KASUMI and that only has been defined. KASUMI operates on a 64-bit input (INPUT) using a 128-key (K) to produce 64-bit output (OUTPUT) as follows. INPUT is divided into two 32-bit strings L_0 and R_0, where:

$$\text{INPUT} = L0 \,||\, R0 \qquad (2.8)$$

Then for each integer ***i*** with $1 \leq i \leq 8$ the operation on the *i*th round of KASUMI is defined as :

$$Ri = Li\text{-}1, \quad Li = Ri\text{-}1 \oplus fi(Li\text{-}1, RKi) \qquad (2.9)$$

This constitute the *i*th round function of KASAUMI, where $L_{i-1} \,||\, R_{i-1}$ is the input data block, $L_i \,||\, R_i$ is the output data bock and RK_i is the *i*th round key, defined as a triplet of subkeys (KL_i, KO_i, KI_i). Subkeys are derived from the key K using the key-scheduling algorithm.

The output data block (OUTPUT) is defined as:

$$\text{OUTPUT} = L8 \,||\, R8 \qquad (2.10)$$

which is the data block offered at the end of the eighth round. In the specification of *ƒ*8 and *ƒ*9 this transformation is also denoted as

$$\text{OUTPUT} = \text{KASUMIK}\ [\text{INPUT}] \qquad (2.11)$$

Components of **KASUMI** and their functionalities have been defined in (3GPP TS 35.202, 2007) and it is highlighted here shortly.

2.5.2.1 *f* Functions

According to (Niemi & Nyberg 2006) each f-function f_i takes a 32-bit input I and returns a 32-bit output O under the control of round key RK_i , where the round key comprises the triplet ((KL_i, KO_i, KI_i). The f-function f_i itself is constructed from two subfunctions: an FL-function FL_i and an FO-function FO_i with associated subkeys KL_i (used with FL_i) and subkeys KO_i, and KI_i (used with FO_i).

The f-function f_i has two different forms depending on whether it is an even round or an odd round. For odd rounds I = 1, 3, 5 and 7.The f-function f_i is defined as:

$$f_i(I, RK_i) = FO_i (FL_i (I, KL_i,), KO_i, KL_i) \qquad (2.12)$$

And for even rounds ***i*** = 2,4,6 and 8, The f-function f_i is defined as:

$$f_i(I, RK_i) = FL_i (FO_i (I, KO_i, KI_i), KL_i \qquad (2.13)$$

i.e for odd rounds first the FL-function and then the FO-function is applied to the round data, while for even rounds the order of the functions is changed.

2.5.2.2 Function FL

The input function of FL_i comprises a 32-bit data input I and a 32-bit subkey KL_i. The subkey is split into two 16-bit subkeys, $KL_{i,1}$ and $KL_{i,2}$, where :

$$KL_i = KL_{i,1} \,||\, KL_{i,2} \qquad (2.14)$$

The input data I is split into two 16-bit halves, L and ***R***, where I = L ||R. The FL-funtions make use of the following simple operations:

ROL (D) the left circular rotations of a data block D by one bit

$D1 \cup D2$ the bitwise OR operation of two data blocks D1 and D2

$D1 \cap D2$ the bitwise AND operation of two data blocks D1 and D2

Then the 32-bit output value of the FL-function is defined as L || R || where:

$$L = L \oplus ROL(R' \cup KL_{i,2}) \qquad (2.15)$$

$$R = R \oplus ROL(L \cap KL_{i,1}) \qquad (2.16)$$

2.5.2.3 Function FO

The input to function FO_i comprises a 32-bit data input I and two sets of subkeys: a 48-bit, KO_i and 48-bit KI_i.The 32-bit data input is split into two halves, L_0 and R_0.where I = $L_0 \| R_0$, while the 48-bit are subdivided into three 16-bit subkeys, where:

$$KO_i = KO_{i,1} \| KO_{i,2} \| KO_{i,3} \text{ and } KI_i. = KI_{i,1} \| KI_{i,2} \| KI_{i,3} \quad (2.17)$$

For each integer j with $1 \le j \le 3$ the operation of the jth round of the function FO_i *i*s defined as:

$$R_j = FI_{i,j}(L_{j-1} \oplus KO_{i,j}, KI_{i,j}) \oplus R_{j-1} \quad (2.18)$$

$$L_j = R_{j-1} \quad (2.19)$$

Output from the FO_i function is defined as the 32-bit data block $L_3 \| R3$.

2.5.2.4 Function FI

The FI-function is depicted in **Figure 8**. The thick and thin lines in this diagram are used to emphasize the difference between the 9-bit and 7-bit data paths, respectively.

An FI-function $FI_{i,j}$ takes a 16-bit data input I and 16-bit subkey $KI_{i\,j}$. The input I is split into two unequal components, a 9-bit left half L_0 and a 7-bit right half R_0, where I = $L_0 \| R_0$, . Similarly, the key $KI_{i,j}$ is split into a 7-bit component $KI_{i,j,1}$ and a 9-bit component $KI_{i\,j,2}$, where and $KI_{i,j} = KI_{i,j,1} \| KI_{i,j,2}$. Each FI-function $FI_{i,j}$ uses two S-boxes: S7 which maps a 7-bit input to a 7-bit output and S9 which maps a 9-bit input to a 9-bit output. The FI-funcions also uses two additional functions, which are designated by ZE and TR. These simple functions are defined as following:

ZE (D) takes a 7-bit data string D and converts it to a 9-bit data string by appending two zero bits to the most significant end of D. TR *(*D*)* takes a 9-bit data string D and converts it to a 7-bit value by discarding the two most significant bits of D.

The function $FI_{i,j}$ is defined by the following series of operations:

$$L1 = R0 \qquad R1 = S9[L0] \oplus ZE\ (R0)$$

$$L2 = R1 \oplus KI_{i,j,2} \qquad R2 = S7[L1] \oplus TR\ (R1) \oplus KI_{i,j,1}$$

$$L3 = R2 \qquad R3 = S9[L2] \oplus ZE\ (R2)$$

$$L4 = S7\ [L3] \oplus TR(R3) \qquad R4 = R3 \quad (2.20)$$

The output of the FI_i, function is the 16-bit data block $L_4 \| R_4$,

2.5.2.5 S-boxes

The two S-boxes (S7 and S9) have been designed so that they may be easily implemented in combinational logic or by a look-up table. Both forms are given for each S-box. The input ***x*** comprises either seven or nine bits with a corresponding number of bits in the output ***y***. Therefore:

$$x = x8 \,||\, x7 \,||\, x6 \,||\, x5 \,||\, x4 \,||\, x3 \,||\, x2 \,||\, x1 \,||\, x0 \,||\, \text{and} \quad (2.21)$$

$$y = y8 \,||\, y7 \,||\, y6 \,||\, y5 \,||\, y4 \,||\, y3 \,||\, y2 \,||\, y1 \,||\, y0 \,|| \quad (2.22)$$

Where the x_8, y_8 and x_7, y_7 bits only apply to S9 and the x_0, y_0 bits are the least significant bits. Gate logical operations of S7 and S9 are described in detail (3GPP TS 35.202, 2007).

2.5.3 Key Schedule

KASUMI has a 128-bit key K. Each round of KASUMI uses 128 bits of key that are derived from K. Before the round keys can be calculated two arrays of 16-bit values K_j and K'_j(j = 1, …,8) are derived in following manner. The first array K_1, K_2, ….K_8 is derived by subdivision of K into eight 16-bit sub-blocks such that:

$$K = K1 \,||\, K2 \,||\, K3 \,||\, K4 \,||\, K5 \,||\, K6 \,||\, K7 \,||\, K8 \quad (2.23)$$

The second array $K`_1$, $K`_2$, …,$K`_8$ is derived from the first array by adding an array of 16-bit constants C_j as follows:

$$K`j = K`j \oplus Cj \quad (2.24)$$

Where the constants Cj are given in following table 1:

Table 1. Constants Cj

C1	0x0123
C2	0x4567
C3	0x89AB
C4	0xCDEF
C5	0xFEDC
C6	0xBA98
C7	0x7654
C8	0x3210

Then the subkeys (KL, KO and KI) are derived as defined by the following Table 2.

Table 2. Subkeys (KL, KO, and KI)

	1	2	3	4	5	6	7	8
$KL_{i,1}$	K1<<<1	K2<<<1	K3<<<1	K4<<<1	K5<<<1	K6<<<1	K7<<<1	K8<<<1
$KL_{i,2}$	K3′	K4′	K5′	K6′	K7′	K8′	K1′	K2′
$KO_{i,1}$	K2<<<5	K3<<<5	K4<<<5	K5<<<5	K6<<<5	K7<<<5	K8<<<5	K1<<<5
$KO_{i,2}$	K6<<<8	K7<<<8	K8<<<8	K1<<<8	K2<<<8	K3<<<8	K4<<<8	K5<<<8
$KO_{i,3}$	K7<<<13	K8<<<13	K1<<<13	K2<<<13	K3<<<13	K4<<<13	K5<<<13	K6<<<13
$KI_{i,1}$	K5′	K6′	K7′	K8′	K1′	K2′	K3′	K4′
$KI_{i,2}$	K4′	K5′	K6′	K7′	K8′	K1′	K2′	K3′
$KI_{i,3}$	K8′	K1′	K2′	K3′	K4′	K5′	K6′	K7′

2.5.4 KASUMI Operations

Since KASUMI is a Feistel cipher with eight rounds, it operates in recursive structure manner in three round functions, namely, FO(), FI() and FL().Both functions of FO() and FI() has a subcomponents which are also have a Feistel-like structure. KASUMI encrypt 64 bits blocks by using a Feistel network of eight rounds. In each round, it alternate between the left and right halves of the state. In first level, one half enters the out round function F*O()* for processing. In second level the FO() function´s output is then XOR’ed to the opposite half of the state and then operation continues with next round. As explained earlier, FO() function itself consists of three round of inner round function FI() and it happens in similar Feister network recursive manner with half the block size. In third level of recursion *FI()* itself consists of four rounds of nonlinear S-Box transformations arranged in a Feiste structure. Following figure illustrate the KASUMI operations detail structures.

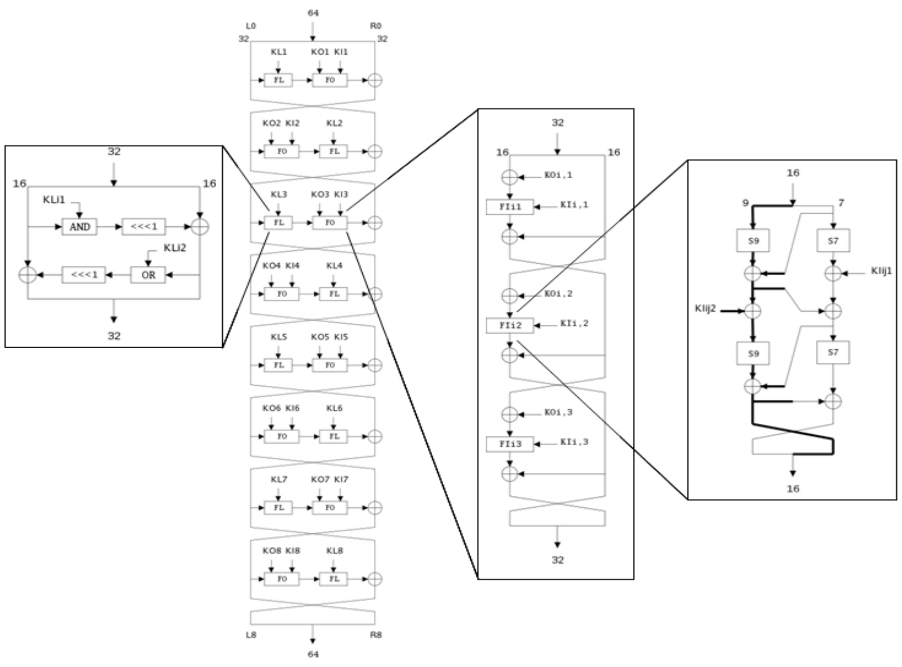


Figure 9. KASUMI operations

3. THEORETICAL ANALYSIS OF CRYPTOGRAPHIC PERFORMANCE

In this chapter, thesis will highlight the benchmarking process of analyzing cryptographic performance in order to understand the characteristics of the multi-core processor being measured. This chapter also provides a short description of the analysis of the potential variables that affect the cryptographic performance of a multi-core processor when measurements are made on the driver level and explains how these variables manifest themselves in the measured performances of the processor that is used. This chapter also introduces a theoretical hypothesis about the implementations of the KASUMI algorithm in both software and hardware. In this way, the predicted can be compared against measured results.

Cryptographic algorithms are used in modern communication systems and have become very important components for ensuring data security. Traditionally, cryptographic algorithms were implemented either by software on a general-purpose processor or incorporated dedicated security hardware in the form of a specialized processor with cryptographic features. In this latter option, the cryptographic functions took place in an off-chip accelerator approach, which means adding another chip or daughter chip and writing low-level code for accessing data between the CPU and this external dedicated chip.

Since cryptographic algorithms are computationally demanding, software-based implementations are very slow or their power consumptions are high, and thus, hardware-based implementations have been preferred, but even this hardware-based cryptographic implementations option also had its drawbacks; for instance, in order to have access to CPU functionalities, low-level code has to be written for transferring the data forth and back between the CPUs, system bus, memory, and other subsystems, which could have a huge effect on the overall performance of the systems.

The present embedded system development is moving on to the next step of cryptographic algorithm implementations. Hardware-based implementations are moved on-chip or System-on-Chip (SoC), which means that mainstream CPUs incorporate cryptographic function hardware acceleration. These hardware accelerations were developed to offload the heavy processing load from the core CPU.

These new SoC systems consist of an embedded CPU and an on-chip bus, a DMA controller, memories, and other subsystem modules, such as dedicated crypto coprocessors used for the accelerations of the cryptographic algorithms. The performance measurements are made for cryptographic algorithms on different levels, on the driver level, the application level, or the protocol stack level. This thesis makes the performance measurements on the driver level. According to (Freescale Semiconductor 2008), there are variables which may influence the results of the measurements.

As the demand for faster processing has increased over the years, the idea of multi-core has been born; in this two or more cores are combined into a single Integrated Circuit (IC) package and in this way, the multi-core technology takes advantage of parallel processing rather than increasing the frequency of the clock rate to achieve higher performance.

Verifying the cryptographic performance claims of multi-core vendors can be a difficult task since there can be a difference between the theoretical cryptographic performance of the processor and its performance in a given application. Some of the results of the performance are inherited from its architecture, while others are due to the performance of the application itself, the protocol stacks of the software, or the underlying OS.

According to (Waters & Stammberg 2009), there are many cryptographic acceleration implementations but there are two common cryptographic acceleration implementation architectures, namely flow-through and look-aside, which most accelerator implementations are based on. The following variables that have been identified in above mentioned reference have an influence on the performance measurements of the cryptographic implementations and would explain shortly as following.

3.1 Influential Variables for Cryptographic Performance Measurements

In this subchapter, 4 variables that could affect the performance measurements of different cryptographic-based implementations are discussed. Although a detailed description of how each parameter affects the results lies outside the scope of this thesis work, they are defined here to raise awareness of their presence in the measurements.

3.1.1 Application or Protocol Stack Software Overheads

Application and protocol stack overheads are the instructions a processor must execute in order to determine what sort of crypto processing is required. Application/protocol stack overheads are typically the greatest source of performance degradation and often the most shocking to users as they transition from a non-secure version of a given protocol to the secured versions.

The reason for this degradation is that security protocols are stateful, so as the cryptographic keys that are used to encrypt and/or decrypt or authenticate the data have lifetimes, which can be measured in terms of the number of bytes encrypted,the number of seconds since the key was first used, the number of seconds since the key was last used, or all of the above. Security protocols also add a header field to packets, allowing the packet to be forwarded to the other end of the security tunnel without revealing information about the parties to the communication.

3.1.2 Software Overheads

Crypto API, application stacks, and driver overhead affect the interface between crypto hardware and application software stacks. Efficient interaction between OS and crypto accelerators is crucial in order to minimize their effects. With small packet sizes, software overheads are more evident with a large number of packets processed, but as the packet size increases, all software overheads become less important and the row performance of the look-aside accelerator becomes more critical.

3.1.3 Bus Bandwidth

The amount of memory bandwidth consumed during look-aside crypto processing is significantly greater than what is consumed during a plaintext operation. Whether the data to be cryptographically processed are a large file or a small packet, the data must originally be moved from the network or peripheral interface to the system memory, moved from the system memory to the accelerator and back, and then moved from the system memory to a network or peripheral interfaces.

In addition to this data movement, the security context, such as the crypto key, must also be fetched. There is also additional memory bandwidth consumption associated with additional look-ups, instruction fetches, or architecturally specific reads or writes. Crypto performance can be constrained if there is not enough bus bandwidth to keep the look-aside accelerator fully utilized. The bus bottleneck could be between the SOC and its system memory.

3.1.4 Data Size

Is the test encrypting of a small or large chunk of data matter? The smaller the size of the chunk of data, the more the results will be influenced by the memory latency of the accelerator, the descriptor size, and other "non-data" context it must fetch to perform the operation, and the accuracy of the timer.

3.1.5 Iteration

A good way to include accelerator DMA overheads and memory latency in a small amount of data while reducing timer resolution as a variable is to construct the test in such a way that the timer starts before iteration 1 and stops after the nth iteration, where n is a fairly large number.

4. EXPERIMENTAL SET-UP AND TOOLS

This chapter will describe both the OCTEON HW and SW architectures, the setting up of the test environment, and the tools used; the software under test is also described briefly in this chapter. The experimental tests that are studied in this thesis were performed at the Nokia Siemens Networks laboratory in Espoo, in Finland.

KASUMI cipher implementations have been tested in several other studies (Jääskeläinen 2003; Tomás Balderas-Contretras 2004; Tomás, Balderas-Contretras, and René, Cumplido 2005 and H.Kim, Choi, M.Kim and H.Ryu 2002). But none of those involved Cavium OCTEON Multicore processors. The OCTEON Multicore processors family contains a wide range of different product options in which some of the chips will contain a hardware accelerator for security algorithms like KASUMI. According to (Cavium Networks, 2009), a Cavium HW-based security accelerator should provide a substantial performance boost compared to a purely SW-based KASUMI implementation. The intention of the experimental KASUMI tests in this thesis is to verify those claims by comparing SW- and HW-based KASUMI implementations to each other.

4.1 OCTEON Hardware Architecture

This section describes the hardware architecture that was used for running the experimental tests. The first subchapter first gives a high-level overview of the OCTEON processor, while the next subchapter describes its architecture in greater detail. This thesis does not intend to describe each and every fact about the OCTEON but just cover it on a high level and touch only on those issues that are germane to this study. For in-depth descriptions of the OCTEON hardware, refer to the OCTEON User's Manual (Cavium Networks, 2009f).

Many embedded applications require a higher performance response time for real-time systems. This demand makes it difficult for a single CPU to satisfy this ever-increasing hunger for high-performance applications, so the use of multiple CPUs has become very popular among embedded system manufacturers.

A multi-CPU system is not the same as a multi-core system; the multi-core processor is a processing system that comprises two or more cores (CPUs) which are integrated in a single chip. Historically, more processing power was achieved by increasing the speed of the processor, but current multi-core processors could deliver better performance by forming multiple logical units into a core one single chip.

4.1.1 Cavium Networks MIPS *(cnMIPS)*

MIPS (*M*icroprocessor without *I*nter locked *P*ipeline *S*tages) is a RISC (Reduced Instruction Set Computer) processor architecture developed by MIPS Computer System Inc. The early MIPS architecture was 32-bit implementation and the latest MIPS64 is based on 64-bit wide registers and data path implementation.

The Cavium Networks OCTEON product family architecture is based on officially licensed MIPS Inc. (MIPS Technology 2001:7-15) technology and uses the multi-core MIPS64 v2 processor instruction set supporting both 32-bit and 64-bit processing. Cavium Networks has added some custom instructions to accelerate common networking operations, such as a bit test branch instructions, security and packet processing instructions or bit-field insert/extract forming the cnMIPS core.

These processors are software-compatible processors, with one to sixteen cnMIPS cores on a single chip and OCTEON II Internet application Processor (IAP) scale from 1 to 32 cores according to (Cavium Networks, 2009).The cnMIPS cores is Cavium Networks´s custom implementation of the MIPS64 release 2 instruction set.

4.1.1.1 OCTEON Product Overview

The OCTEON product family includes OCTEON, OCTEON Plus and OCTEON II multi-core MIPS64 processors. All OCTEON processors are software compatible and supported by industry-standard software tool-chains and operating systems. All OCTEON products share the same architecture and more detail description of OCTEON products are shown in Cavium Networks Product Table (Cavium Networks, 2009c).

In this thesis work, OCTEON Plus CN58XX product family is used for running on the experimental tests that are used for performance tests on KASUMI cipher implementations. This processor has a sixteen cnMIPS cores at up to 750MHz speed on a single chip. The CN58XX processor is provides security hardware acceleration which implements the latest set of algorithms including KASUMI cipher algorithms.

4.1.1.2 OCTEON Architecture Overview

According to (Cavium Networks, 2009d) All OCTEON products share the same OCTEON architecture. Individual OCTEON products vary and scale based on the number of integrated cnMIPS cores, the frequency that the cores and the internal interconnects run at, the type and the number of I/O interfaces integrated, and the type and the number of application acceleration hardware units integrated.

According to (Cavium Networks, 2009e) key OCTEON features which most OCTEON models provide are:

- Up to 32 cores, up to 1.5 GHz: The OCTEON family of multicore processors supports up to 32 cnMIPS with a speed from 300MHz to 1.5GHz.
- Hardware Acceleration Units: Multiple hardware acceleration units are integrated into each processor. These acceleration units include; Packet-managment accelerators, Security acceleratiors, Application accelerators and specialized accelerators.
- Dedicated DMA Engines: Dedicated DMA Engines are provided for each hardware unit which accesses memory.
- High-Speed Interconnects: The hardware units and the cores ae connected by high-speed interconnects. These interconnects run at the same frequency as the cores. Each interconnect is a collection of multiple buses with extensive pipelining and sophisticated hardware arbitration logic.
- Industry-Standard Tools chains and Operating Systems: Industry-standard tool chains (GCC, GDB) and operating systems (including SMP Linux) has been modified to utilize the OCTEON processor´s multiple cores, hardware accelerators and Cavium Networks –specific instructions.
- Packet Management Acceleration: Packet receive/transmit is automated by software-configurable packet management accelerators.
- TCP/UDP acceleration
- Per-Core Security Hardware Acceleration. Common security algorithms including KASUMI are accelerated by optional per-core Security Engines.

The architectural definition of the CN58XX product line provides a good perspective for illustrating the OCTEON architecture. Following figure is OCTEON Plus CN58XX block diagram.

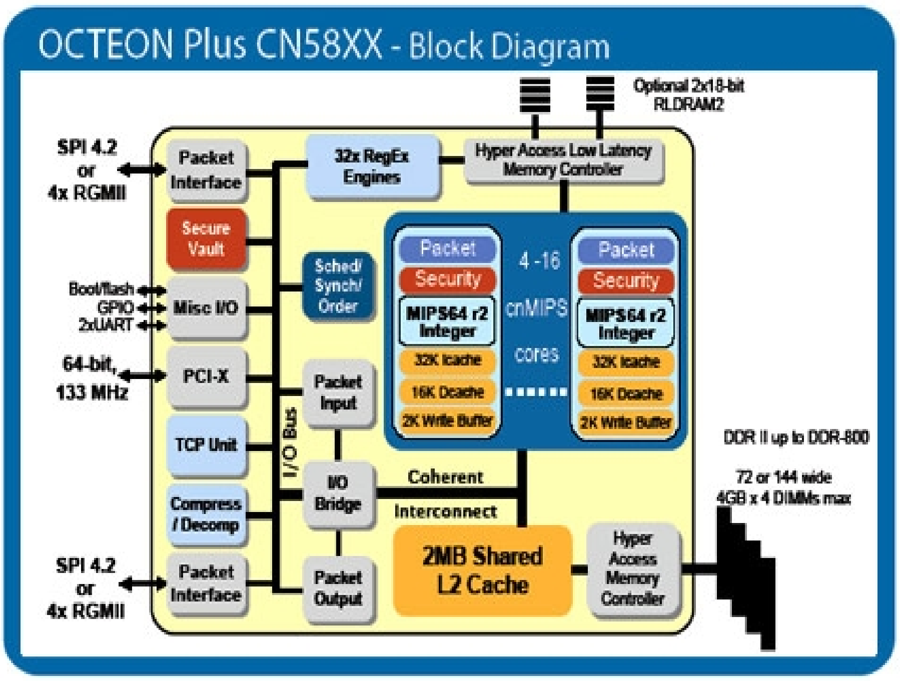


Figure 10. Cavium Networks MIPS Architecture (cnMIPS)

Although every units of OCTEON hardware is essential to work of this thesis work, in this subchapter, will be highlighted only some of the hardware –acceleration units that their functionalities are very essential to this thesis work.

As described earlier, Hardware-Acceleration units consists of packet-management accelerators, security accelerators, application accelerators and specialized accelerators. Packet management accelerators and security accelerators has huge effect on work and the experimental test results analysis of this thesis work, so their functionalities would be described them in detail following paragraphs.

Packet Management Accelerators in its part consists of Schedule/Synchronization and Order (SSO) unit that manages packet scheduling and ordering, Free Pool Allocator (FPA) unit that handles pools of free buffers including Packet Data buffers, and Input Packet Data (IPD) unit which manages packet input. Packet Input Processor (PIP) unit which works closely with IPD a handles with input packets, Packet Output Unit (PKO) manages packet output.

PIP (Packet Input Packet) and IPD (Input Packet Data) units work together to receive the packet and perform early processing on it. Each packet is represented as “work” for the cores to do and is represented by a Work Queue Entry (WQE) data structure which contains the tag type, tag value QoS value, group and pointer to Packet Data Buffer. These units are responsible for many layer-2 through layer-7 processing requirements such as exception checking and TCP/UDP checksum verification.

Schedule/Synchronization and Oder (SSO) is responsible for scheduling the work to cores and also maintaining the packet ingress order. Cores may request work from SSO either asynchronously (the core continue to do other work while the instruction completes) or synchronously (the core waits for instruction to complete). In another word, SSO unit provides Scheduling, Synchronization and Ordering services.

Packet Output (PKO) in its turn is responsible for packet transmission. When the packet processing is complete, the core notifies the PKO that the packet is ready for transmission and PKO manages the transmission priority.

Security accelerators consists Random Number Generator (RNC) unit, Key Unit which provides and manages secure on chip memory which can be used to store a hardware key and can be reset using an external pin. Per-Core Security or Security Engine, this is special coprocessor used for accelerating security applications and hash generation and there is one Security-Coprocessor per core.

Once the core issue an instruction to the coprocessor, the core can continue to do other work while the coprocessor complete the instruction, or the core can wait for the coprocessor to complete the task. Following figure illustrate the communication between core and its co-processor.

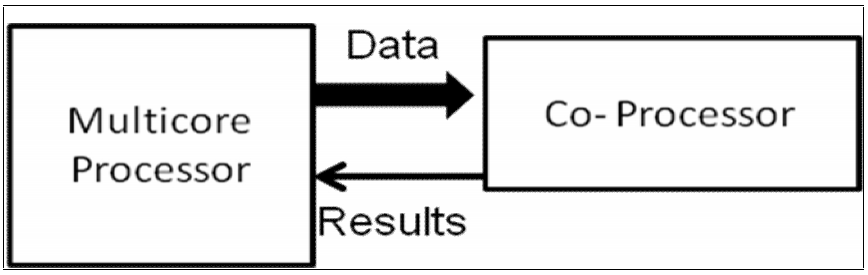


Figure 11. Communication between core and its co-processor

More detail descriptions of these units and how they functions is handled in (Cavium Networks, 2009e).

4.1.1.3 Other Units and Functionalities

According to (Cavium Networks 2009c, 2009d) there are also other import units beside those been described in above paragraphs, such as On-Chip Interconnects which joins the different integrated units together. The OCTEON on-chip interconnect architecture involves a network of Coherent Memory Buses (CMB), and another network of interconnects connecting the I/O sub-system (IOB) together. Both of these networks are implemented based on multiple split-transaction and highly pipelined buses. These two networks, CMB, and IOB, are connected together through a high performance I/O bridge.

The CMB network connects all the cnMIPS cores, L2 cache controller, memory controller, and I/O bridge together while the I/O sub-system (IOB) includes the controllers for the various I/O interfaces, all the application specific offload and acceleration engines, and the I/O bridge. The OCTEON architecture offers as well a cache hierarchy which includes split instruction and data L1 caches on each cnMIPS core, and a large L2 cache, up to 2MB, that is shared by all cnMIPS cores and the I/O subsystem.

The OCTEON architecture includes also, a large variety of application specific offload and acceleration engines. Some of these acceleration features are implemented as individual functional units that are shared among all the cnMIPS cores, while some of these acceleration features are integrated into each of the cnMIPS cores. The decision of stand-alone functional units against integration into cnMIPS core is optimized for each of the application specific acceleration features in the OCTEON architecture.

4.2. OCTEON Software

This subchapter highlights the OCTEON processor´s software overview specially software architecture and other software utilities that is used to access the hardware services. The OCTEON Software is a Software Development Kit parts that is intended to provide the tool, drivers Application Protocol Interface (APIs), Libraries and other utilities that is needed to access the underlying HW chipset solution.

It also plays crucial role for providing services to higher level applications that run on the OCTEON processors. Following figure shows high level overview of software suits that this test system uses.

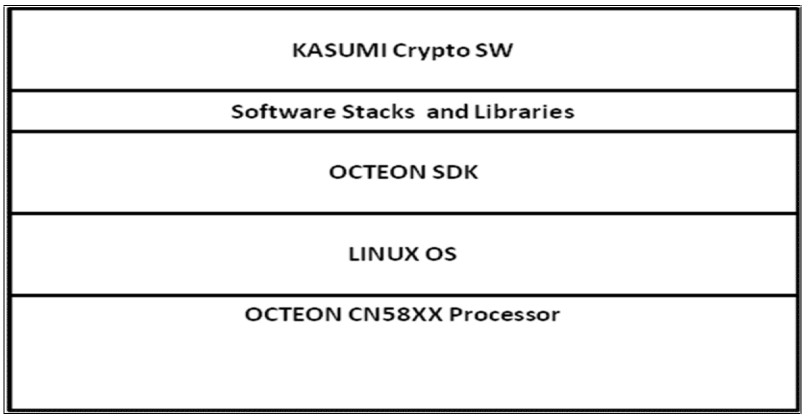


Figure 12. High level test system architecture

The KASUMI Crypto software that is to be tested runs on OCTEON Simple Executive to leverage OCTEON per-core security acceleration to achieve maximum performance possible.

4.2.1 Simple Executive API

In this thesis work KASUMI test are performed by using Octeon Simple Executive architecture instead of Linux based application. The reason for this choice is the services that normal Operating Systems provide to applications such as interrupts, context switches, kernel address space copy in/copy out with system calls causes some overhead to pay. The Simple Executive provides a Hardware Abstraction Layer (HAL) in the form of an Application Programming Interface (API) to the underlying hardware units. This API is a very thin layer of simple functions which access the Central Process Unit (CPU) registers. It also provides some convenience routines for block initialization. The API can be used from both kernel and user mode.

In another word, Cavium OCTEON Simple Executive (SE) means that the executable SW will run directly on top of OCTEON HW without Operating System (OS) e.g Linux.

Following figure is show the role of the Simple Executive API:

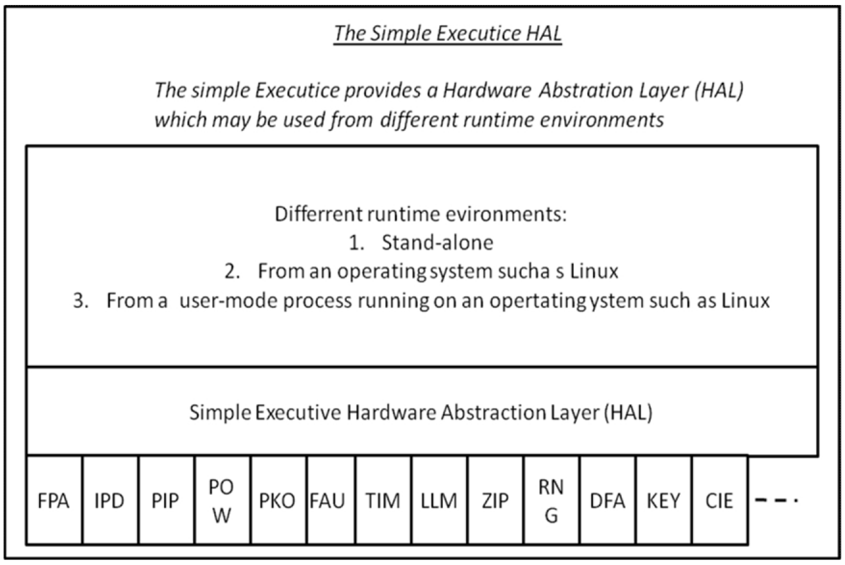

The Simple Executice HAL
The simple Executice provides a Hardware Abstration Layer (HAL) which may be used from different runtime environments
Differrent runtime evironments:
1. Stand-alone
2. From an operating system sucha s Linux
3. From a user-mode process running on an opertating ystem such as Linux
Simple Executive Hardware Abstraction Layer (HAL)
FPA
IPD
PIP
POW
PKO
FAU
TIM
LLM
ZIP
RNG
DFA
KEY
CIE

Figure 12. Simple Executive API

The Simple Executive API is used to access the hardware units:

- Basic units; FPA, IPD, PIP, SSO and PKO
- Intermediate units: FAU and TIM
- Advanced units: LLM, ZIP, RNG, DFA, KEY, CIU, etc.

Simple Executive API also includes functions and macros for:

- System memory allocation (bootmem)
- Synchronization between cores
- spinlocks
- Reader-Writer locks
- Atomic set, add, compare and store operations
- Barrier functions

Simple Executive functions and macros may be used either to create stand-alone Simple Executive application, or may be called from drivers or applications running on an operating system kernel such as Linux. For instance, after the Linux kernel is booted, a

Cavium Networks Ethernet driver may be started. This driver uses the Simple Executive API to configure the OCTEON hardware. Simple Executive User-Mode applications may also be started from Linux. Both 32-bit and 64-bit modes are supported, although 64-bit mode should be used whenever possible due to better overall performance.

Following figure shows using Simple Executive API from different Runtime Environment:

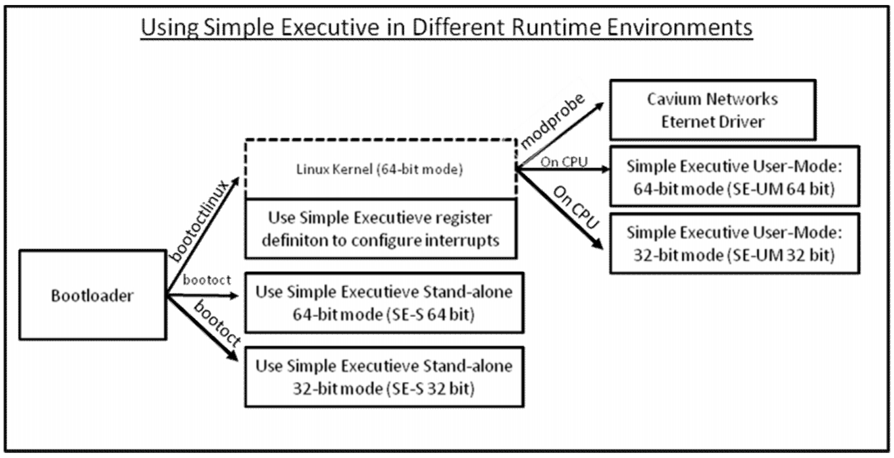

Using Simple Executive in Different Runtime Environments
Bootloader
bootoctlinux
bootoct
bootoct
Linux Kernel (64-bit mode)
Use Simple Executieve register definiton to configure interrupts
Use Simple Executieve Stand-alone 64-bit mode (SE-S 64 bit)
Use Simple Executieve Stand-alone 32-bit mode (SE-S 32 bit)
modprobe
On CPU
On CPU
Cavium Networks Eternet Driver
Simple Executive User-Mode: 64-bit mode (SE-UM 64 bit)
Simple Executive User-Mode: 32-bit mode (SE-UM 32 bit)

Figure 13. Simple Executive API runtime environments

4.2.2 Runtime Environment Choices for cnMIPS Cores

There are several choices for runtime environment. The tree supplied by Cavium Networks is Simple Executive stand-alone mode, Linux and the hardware simulator:

1. When running Simple Executive on multiple cores, the same FLF file is usually run on all of the cores. These cores are all started from one load command.

2. When running Linux on multiple cores (SMP), there is one kernel running, not one kernel per core. Linux applications are schedule to run on different cores.

3. The third runtime environment supplied by Cavium Networks is the Hardware Simulator. The simulator is useful when actual hardware is not available and it also very useful for performance tuning. Performance tuning is most easily done using the tool Viewzilla. This tool analyzes the output of the simulator, so making sure the code will run on the simulator as well as on actual hardware is recommended for performance critical application.

In addition to the three runtime environments supplied by Cavium Networks that is described above, several open-source and proprietary operating systems are available. In this thesis work, the Simple Executive running as Linux Stand-alone (SE-S) application is used, so thesis considerate to explain in detail how it works.

4.2.2.1 Simple Executive

Simple Executive provides an API to the hardware units. Simple Executive may run Stand-alone (SE-S) application, or as user-mode (SE-U) application on an operating system such as Linux, in another word, Linux kernel and applications may both make Simple Executive API calls. When Simple Executive calls are made from Linux user space, the process is referred to as a Simple Executive User-Mode application (SE-UM).

For instance, when run as a user-mode application, different application start code (*main()*) is called and there other minor porting items to consider. The following figure shows a representation of a core running Simple Executive in Stand-alone (SE-S) mode. One core runs a Simple Executive Stand Alone (SE-S) Application.

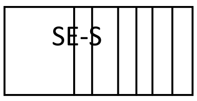


Figure 14. Simple Executive Stand-alone Application (SE-S)

Simple Executive calls may be made from kernel mode. For example, the Cavium Networks Ethernet driver, which runs on Linux, makes Simple Executive calls:

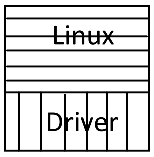


Figure 15. Simple Executive calls from kernel Mode

The following figure shows a representation of a core running Simple Executive as a User-mode application.

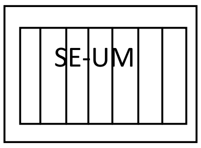


Figure 16. Simple Executive User-Mode (SE-UM) application

To use all available memory, SE-UM applications should be compiled for 64-bit mode. 32-bit mode is sometimes used, but can only access a limited amount of physical memory. SE-S is very fast compared to SE-UM. There no context switches, and all the memory is mapped for fast access. To get maximum performance from OCTEON: Whenever possible, design the application to use a 64-bit Simple Executive application.

4.2.2.2 Application Configurations

There are number of possible way to do the multi-core configuration, but one of the easiest configurations to implement is Simple Executive Stand-alone application configurations. Following figure shows example of 8-core simple system running a Simple Executive Stand-alone application uses 1 ELF file, 8 instance of SE-S.

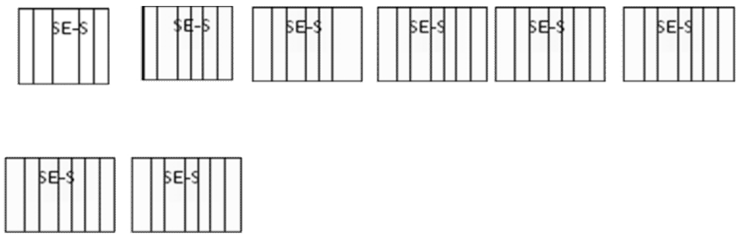


Figure 17. 8-core simple system running a SE-S application

4.2.3 Other Software Issue

In this subchapter, thesis will go through some other steps and issue that are related to complying and running the software under the test.

4.2.3.1 Application Entry Point and Start up Code

An application may be compiled as either SE-S or SE-UM application without modification. The code executed when application is started is not the same: when SE-S is the built target. The makefile *$OCTEON_ROOT/executive/cvmx.mk* is responsible for making this change.

4.2.3.2 Booting SE-S Application

When the application is executed and SE-S object is compiled, then to boot Simple Executive application *bootoct* bootloader command is used. This command is used to boot application on one or more cores. Following figure shows cores running Simple Executive Stand-alone in a Load Set.

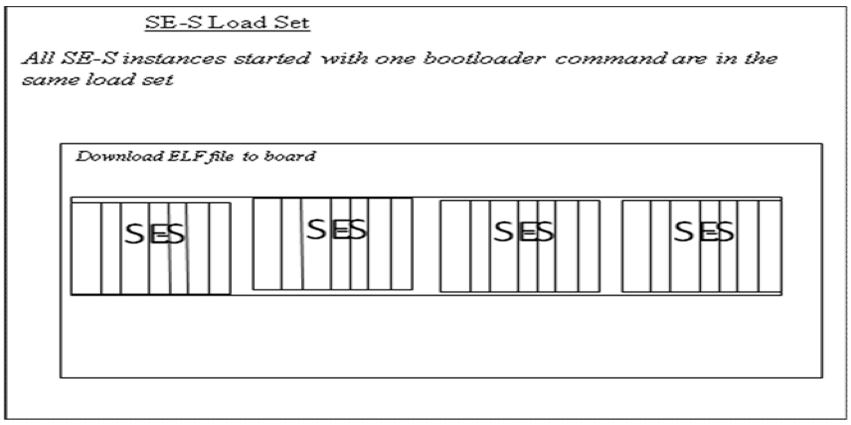

SE-S Load Set
All SE-S instances started with one bootloader command are in the same load set
Download ELF file to board
SES
SES
SES
SES

Figure 18. Simple Executive Stand-alone Mode (SE-S) application

Following figure is shown in detail how booting SE-S applications with Bootoct command works:

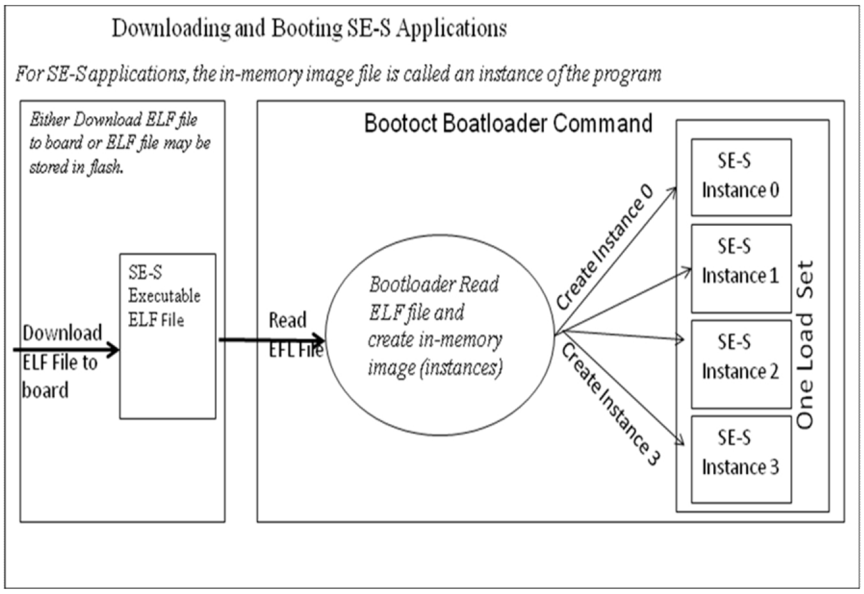

Downloading and Booting SE-S Applications
For SE-S applications, the in-memory image file is called an instance of the program
Either Download ELF file to board or ELF file may be stored in flash.
Download ELF File to board
SE-S Executable ELF File
Read EFL File
Bootoct Boatloader Command
Bootloader Read ELF file and create in-memory image (instances)
Create Instance 0
Create Instance 3
SE-S Instance 0
SE-S Instance 1
SE-S Instance 2
SE-S Instance 3
One Load Set

Figure 19. Downloading and Booting SE-S Applications

4.2.4 Software Architecture

In most software design paradigm that intended to run embedded systems follows the separation of control and data planes in order to simplify the software architecture design. In cnMIPS software helps too to separate two basic types of processing: normal packet processing (fast path), and exception processing (slow path). Depending on the number of cores available, different configuration of cores devoted to either fast path or slow path processing can be used to optimize throughput.

4.2.4.1 Control-plane versus Data-plane applications

Typical telecom networking SE-S application functions are dived into two categories: control-plane (slow path), and data-path (fast path). In OCTEON processor the control-plane usually handles exceptions while the data-plane handles normal packet processing. Software application used in this thesis work falls into fast path category.

According to (MIPS Technology 2001) SE-S applications may be used for both control-plane and data-plane. SE-S applications provide the lowest overhead and highest potential for scaling. Next best solution (a typical solution) is SE-UM for control plane and SE-S for data-plane.

Following figure illustrate SE-S application divisions and only one core does the initialization routine.

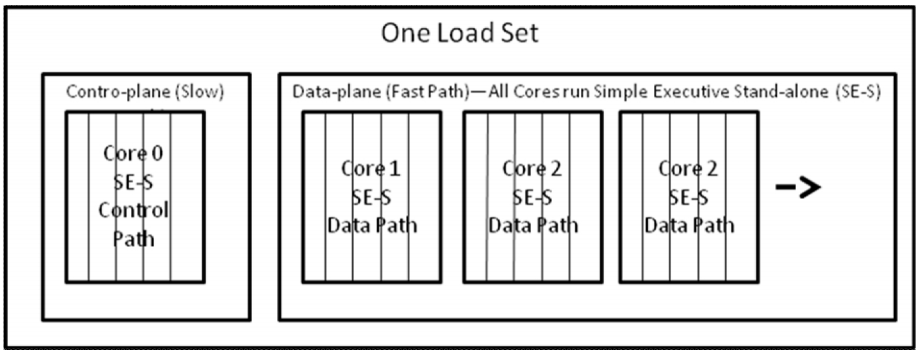


Figure 20. SE-S used both Control plane and Data plane

4.2.4.2 Event-driven Loop (polling) versus Interrupt-driven loop

There are two different models for receiving packet to process: an even-driven loop (polling) or an interrupt-driven loop.

An even-driven loop looks like:

```
while ( there is work to do )
{
        do the work
}
```

The event-driven loop is a higher performance processing architecture than the interrupt-driven loop. In an event-driven loop, when the core is ready for work and work is available, it gets the work; when there is no work, the core loops looking for work to do. When using an interrupt-driven loop, there may be a delay between work available and the process being notified. The SSO interrupts are configured based either on a time counter or the quantity of work available for a particular group. Instead of looping looking for work, the interrupt-handler thread exits, then is called again when the interrupt occurs. This not only can result in work being processed less quickly, but also results in more context switches, costing unnecessary system overhead.

4.2.4.3 Packet Processing Steps in OCTEON

Performance analysis of KASUMI cipher implementations is handled in chapter 6, in order to do that performance analysis, thesis considers that it is very essential to understand how packet data is processed inside the processor. Packet processing functionalities could be grouped into three different block parts that participates packet processing, namely, Packet Input, SSO and Core Processing and Packet Output.

Packet Input block part consists of two blocks: Packet Input Processor (PIP) and Input Packet Data (IPD) which work closely together. This block parts receives and processes data from the Simplified Packet Interface (SPI) which is a generic representation of the receive (RX) and the Transit (TX) functions of any of the several interfaces that OCTEON processor can be used to receive the data.

Packet is processes inside Packet Input Block as following:

1. After the interface RX port receives the packet and checks it for error, it passes the packet to the Input Packet Data (IPD) unit. The IPD shares the data with the Packet Input Processor (PIP).

2. After Packet Input Processor (PIP) performs the packet parsing, including any check configured by software, it computes the data needed by the Input Packet Data (IPD) for the Work Queue Entry (WQE) fields (work flow and QoS). Work Queue Entry (WQE) is a data structure which contains the tag value, QoS value, group and pointer tot eh Packet Data Buffer.

3. If Input Packet Data (IPD) does not drop the packet, it allocates a WQE Buffer and Packet Data buffer from the Free Pool Allocator (FPA) unit. The FPA manages the free buffers.

4. The Input Packet Data (IPD) write the WQE fields to the WQE Buffer and writes the packet data to the Packet Data Buffer in L2/DRAM.

5. The IPD performs the *add_work* operation to add the WQE Pointer to the appropriate QoS queue in the Schedule Synchronization Order (SSO) unit

Packet processing inside the SSO and Core Processing blocks in its turn happens as following:

1. The core performs the *get_work* operation to get a new WQE pointer from the SSO. The WQE contains the Packet Data Buffer pointer.

2. The core processes the packet data, reading and writing the packet data in L2/DRAM

3. After processing the packet data, the core sends the Packet Data Buffer pointer and the data offset to the appropriate Packet Output Queue in the Packet Output (PKO) unit. The queue´s configuration specifies the output port and packet priority.

4. The core frees the WQE Buffer back to the Free Pool Allocator (FPA).

Packet is processes inside Packet Output block as following:

1. The Packet Output (PKO) DMA read data from Packet Data Buffer in L2(DRAM) into its internal memory.

2. The Packet Output (PKO) unit optionally adds the TCP or UDP Checksum, then sends the packet data from its internal memory to the Output port. The interface TX port will then transmit the packet. The PKO optionally notifies the core that the packet was sent.

3. The Packet Output (PKO) free the Packet Data Buffer back to the Free Pool Allocator (FPA) unit.

4.2.4.4 Using Work Groups in Packet Processing

Software applications manipulate or configures the packet processing services provided underlying OCTEON HW. In order to understand that, following paragraphs describes in detail how data packets are processed inside the packet processing blocks of OCTEON.

The Work Queue Entry (WQE) data structure contains a field “Grp” which stands for Group or Work Group. This group number is set by the PIP/IPD unit based on the setting of its configuration register when packet is received.

When a core performs *get_work* operation, the request goes to SSO scheduler which maintains per-core group mask. This group mask has one bit set for each group the core will accept work from. For stance, when the scheduler receives the *get_work* request, it will schedule the highest priority WQE which is based on its group that schedulable to the core. Cores may also be configured such that it may accept work from a limit set of input queues. Following figure illustrate cores and group works

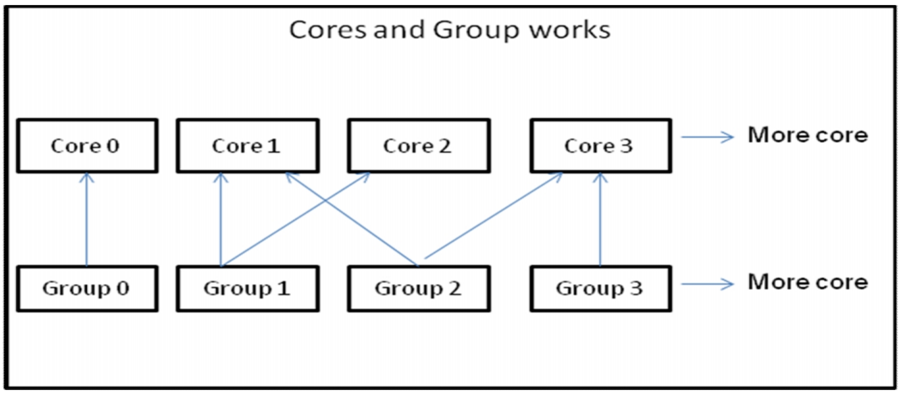


Figure 21. Core may accept work from any and all groups

The SSO scheduler maintains a per-core group mask and this group mask has one bit set for each group the core will accept work from. By using the Simple Executive functions it is set this group mask which then modifies the per-core SSO registers.Once the core performs the *get_work* operation, it works only with a group number corresponding to a bit set in the core’s group mask.

4.2.4.5 Passing Work from one Core to another Core

According to (Cavium Networks, 2009i) document the SSO scheduling functions is specific to Cavium Networks and divides packet processing into different phases. This thesis work interested in only for stance packet data processing from one core to another core.

First, ones the packet is received in Packet Input (PKI) block, it performs basic checking like header checks and flow classification and stores the data packet into

L2/DRAM. Then Packet Input (PKI) block creates data structure which contains the information needed by the SSO to manage the scheduling, synchronization and ordering the packet. After that PKI submits the packet information to the SSO.

The SSO will put the packet information into the selected QoS Input Queue. There is a Packet Data Buffer which contains the received packet data. The Input Packet Data (IPD) allocates the buffer from the Free Pool Allocator (FPA) which manages the free buffers. The IPD then copies the packet data into the buffer.

The Quality of Service (QoS) is a number (0-7) which represent the priority of the packet. When packet is received, the PIP/IPD computes the QoS number for the packet and save the value in the Work Queue Entry (WQE). The PIP/IPD fill the WQE fields and then sends the WQE pointer to the SSO using the *add_work* operations.

The SSO has 8 QoS *Input Queues* (0-7) one per QoS value. When a new WQE is added to the SSO, the WQE goes onto the Input Queue which matches its QoS value. When the WQEs are added to the SSO´s Input Queue, the Next Pointer is used to link them into list.

The SSO contains internal memory. Part of the internal memory has been used to create a limit number of Work Descriptors (WD). Each Work Descriptor contains the key information needed by the SSO to schedule the work on a core and to keep the packet in the correct order. The key fields in the Work Descriptors are WQE pointer, Tag value, Tag Type (TT), QoS and group (Grp). Tag Type (TT) could be either one of : ORDERED, ATOMIC or NULL.

Multiple packet form the same flow (collection of packet that shares the same tag types and tag value) with an ORDERED *tag type* can be processed in parallel by multiple cores. The SSO caches the head of each QoS queue in internal memory, one Work Desciptor per WQE. The portion of the QoS queue which is in internal memory is referred to as the *Cached Input Queue.*

Inside the SSO, there is one *Core State Descriptor* data structure for each core. The states of Core states are either *Scheduled* or *Unscheduled.* A Work Descriptor that has been assigned to a core is considered to be *scheduled*. The core may perform an operation to deschedule the Work Descriptor, so that the Work Descriptor is no longer assigned to the core.

When core performs a successful *get_work* operation, a Work Descriptor is removed from the Cached Input Queue and assigned to the core. A pointer to the assigned Work

Descriptor is stored in the Core State Descriptor. The Work Descriptor contains a pointer to the WQE. The *get_work* operation returns the WQE pointer to the core.

Once a Work Descriptor has been scheduled on a core, it is considered to be in-flight until it is discarded or switched to NULL. Descheduled Work Descriptors are also considered to be *in-flight* since processing on the associated WQE has started but has not completed.

In-Flight Queues are internal to the SSO and maintained by the SSO. This queue is very important for stance to maintaining packet order. When Work Descriptor is scheduled on core, it put onto the *In-flight* Queue which corresponds to its tag tuple, so there is one *In-flight* Queue per unique tag tuple.

Allowing multiple cores to work on packets from the same flow allows scaling where more cores working on packets in the same flow results in faster packet throughput

The following steps are used to pass work from one core to another core:

1. The *swtag_deschead()* operation descheduled the work from the core. The work remains the In-Flight Queue so that ordering properties are maintained.

2. The corresponding Work Descriptor (WD) is unscheduled from the core and its state is set to Descheduled.

3. Once the WD is the dead of its In-Flight Queue, a pointer to it is stored in the Descheduled Now-Ready List (DS-Now_Ready_list). The WD can now be scheduled to a new core. (There is one DS-Now-Ready List per group. These lists contain only pointers to WDs which are ready to be rescheduled because each is the head of its In-Flight Queue).

4. A new core will receive the now-ready WD when the core perform the *get_work()* operation and the SSO schedules now-ready WD to the core.

The DS-Now-Ready List has a higher priority than the QoS queue, which allows now-ready in-flight work to complete prior to new work.

4.2.4.6 Pipelined versus Run-to-Completing Software Architecture

OCTEON supports traditional pipeline, run-to-completion, and modified pipeline architectures.

Software Architecture supported includes:

1. Run-to-completion: In run-to-completion architecture
2. Architure, each core performs all the functions, and the packet stays on the same core as it moves through the series of functions. This option implements a static hard-coded core allocation.
3. Traditional pipeline: In Traditional pipeline architecture, each core handles one function and the packet moves through the pipeline, changing cores as needed to pass through the series of functions. The stages of the pipeline are bound to specific cores. On OCTEON, when each core completes its part of the processing, it changes the packet's work group to a new value, and performs the *swtag_desched()* operation to send the packet to the next core in the piple. The next core receives the packet when it performs the *get_work ()* operation.
4. Modified pipeline: On OCTEON, because there is no limitation on code size, a modification on the traditional pipeline architecture can be used. A modified pipeline is one when any core can process any stage of the pipeline: the stages are not bound to specific cores. This modified architecture provides better load-balancing and scaling capabilities than traditional pipelining.

In this thesis work, run to complete option is used for running the experimental test since one core or number of cores does all the work and there is no other instructions or tasks that are needed to be done outside the original core (s) that are handling the test.

4.3 Software under the Test

This subchapter describes the software under the test. The software that is under the experimental test in this thesis work is 3GPP KASUMI software implementation in C programming. The source code (C code) of the KASUMI implementation is shown in the annex 2 part of (3GPP TS 35.202, 2007). First, the code could not directly be ported to OCTEON environment, so some adaption fixes has be done to the original code in order to run on OCTEON processor. Secondly, original code is not optimized, so Cavium Networks done some optimizations for OCTEON platform.

Software consists of function calls that implements *FI(), FO()* and *FL()* functionalities as well as function that implements the main KASUMI algorithm including building the key schedules. In order to see where needs yet more optimizations, it is outside the scope of this thesis work.

5. TEST EXECUTION AND RESULT COLLECTION

This chapter provides a detailed description of the experimental test cases that are used to evaluate the performance of the KASUMI cipher implementations. In the first phase, the evaluation of the performance is based on a set of data messages that are sent between two cores but the actual processing is performed in one core. In the second phase of the evaluation, the processing of the data packets is shared between more than one cores.

First, the configuration files that are used for configuring the test cases are defined, then the execution of the test cases follows, and last, the test results are collected.

5.1 Test Scenarios Selection

KASUMI cipher is used to encrypt and decrypt the sensitive messages between the user and the network and it is implemented in the NAS layer of the User Equipment (UE) and Radio Network Centre (RNC). The lengths of 3G NAS messages are specified in (3GPP TS 23.930, 2010) and the transaction speed depends on the capacity of the product. In this thesis, several different test cases are conducted because of the theoretical verification of the processor and not the KASUMI algorithm itself.

Test case traffic rates were selected to verify the expectations about the 3GPP reference results. Each test case has subcases. The number of samples that are tested will be from 2 to 4 samples in order to collect enough data for the analysis. In order to verify how KASUMI implementation behaves with small amount of data, this thesis work opted to use 40-byte and 88-byte message lengths. For a long message length, this thesis work opted to use 500 bytes and 1000 bytes.

To test the transmission data block sizes and how they affect the KASUMI implementations, this thesis opts to start with traffic formed of small data blocks, such as 100 and 500 IPv4 packets. For traffic formed of larger blocks from 1000 up to 8000 IP are used. Following table is shown number of cases selected for one core implementation testing.

Table 3. Test cases for one core

TEST CASES	KASUMI HW	KASUMI SW	Short Bits	Long Bits
Ciphering and Deciphering	X	X	x	x
Only Ciphering	X	X	x	x
Only Deciphering	X	X	x	x
Key Generation Effects	-	-	-	-
KASUMI cipher mode effects	-	-	-	-
Multicore Ciphering and Deciphering	X	X	x	x

5.2 Running the Test Cases

In order to run the test cases, the following steps must be followed.

1. The application will be downloaded from the development host computer to the development target computer via Ethernet, using the ssh utility.

   ssh yasinara@10.144.19.19

   then the compiled SE-S application will be loaded:

   scp yasinara@10.144.19.19:~

2. After it is downloaded, the application is booted by typing the bootoct command in the target console

   $yasinara@10.144.19.19: oct58sdk

   $yasinara@oct58sdk: oct-pci-reset

3. Connect to the target console

Minicom will provide a connection to the target console and after you are connected, you should see the bootloader prompt.

4. Set the object file name in the bootloader in order to select a correct object instance name from the load set of the core.

5. Run the test application that you have already built by using the Boot script:

   $yasinara@oct58adk: ./BOOT

5.3 Hardware-based KASUMI Acceleration Case Simulation

In this test case, the data under test are sent in plain text form from one core to another core, which accelerates the HW (Hardware) encryption and decryption functionalities. In order to capture the actual processing time for acceleration, the transfer time from one core to another core is subtracted from the processing time. The acceleration time is measured in terms of the clock ticks of the OCTEON processor.

5.3.1 Case A: KASUMI HW Encryption and Decryption of packets of 40 bytes

Data packets with a length of 40 bytes are sent from one core to another core, which accelerates the data received by performing encryption and decryption. In order to see how different data blocks affect this acceleration, about 8 different data blocks are used in this test case, starting from 1000 and going up to 8000.

The table below shows the test results when data packets with a length of 40 bytes are sent from one core to another core.

Table 4. KASUMI HW Enc and Dec on 40 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW (ticks) Tics for data transfer	Kasumi HW (ticks) Tics for processing packets	Actual Kasumi HW (ticks) Processing time – Transfer time
1000	40	320000	219635	1438347	1218712
2000	40	640000	439121	2876510	2437389
3000	40	960000	660700	4317096	3656396
4000	40	1280000	870790	5772137	4901347
5000	40	1600000	1031082	7241771	6210689
6000	40	1920000	1193866	8784992	7591126
7000	40	2240000	3126625	10381396	7254771
8000	40	2560000	3286899	11950643	8663744

5.3.2 Case B: KASUMI HW Encryption and Decryption of packets of 88 bytes

Data packets with a length of 88 bytes are sent from one core to another core, which accelerates the data received by performing encryption and decryption. 8 different data blocks are used in this test case, starting from 1000 and going up to 8000, as in the previous case (a). The following table below shows the results of this test case.

Table 5.KASUMI HW Enc and Dec on 88 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW (ticks) Tics for data transfer	Kasumi HW (ticks) Tics for processing packets	Actual Kasumi HW (ticks) Processing time – Transfer time
1000	88	704000	219559	2722153	2502594
2000	88	1408000	439080	5444199	5005119
3000	88	2112000	660219	8168705	7508486
4000	88	2816000	849970	10904975	10055005
5000	88	3520000	1010063	13694086	12684023
6000	88	4224000	2941114	16553936	13612822
7000	88	4928000	3101505	19447713	16346208
8000	88	5632000	3260885	22240587	18979702

The figure below shows the test results from both test cases, (a) and (b), depicted side by side in order to make it possible to see the difference between them clearly.

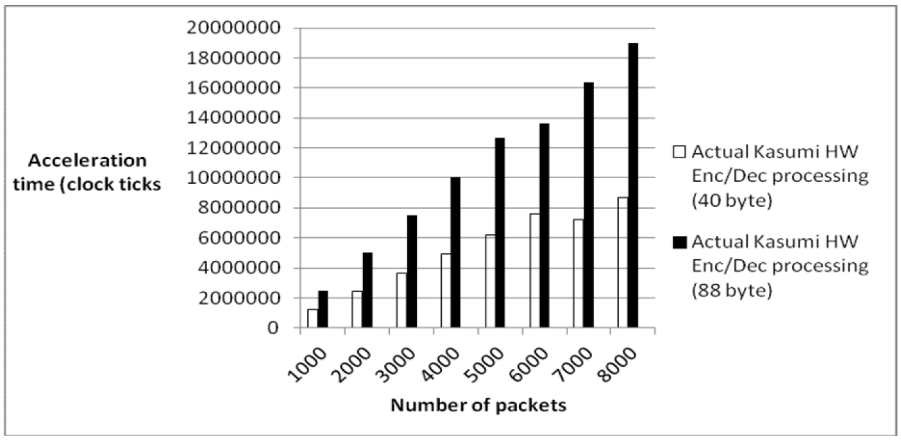


Figure 22. KASUMI HW Enc and Dec on 40 and 88 bytes packets

5.3.3 Case C: KASUMI HW Encryption and Decryption of packets of 500 bytes

In this test case, the size of the data packet increased to 500 bytes and the data blocks used in this case are 1000, 5000, and 8000. The table below shows the results of this experiment.

Table 6. KASIMI HW Enc and Dec on 500 bytes packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW (ticks) Tics for data transfer	Kasumi HW (ticks) Tics for processing packets	Actual Kasumi HW (ticks) Processing time – Transfer time
1000	500	4000000	5068439	13894593	13674397
5000	500	20000000	2790077	71088453	68298376
8000	500	32000000	220196	113788809	108720370

5.3.4 Case D: KASUMI HW Encryption /Decryption of packets of 1000 bytes

In this case, the data size is increased to 1000 bytes and the data blocks that are used in this case are 1000, 5000, and 8000, as in the previous case (c). The table below shows the results of this experiment.

Table 7. KASUMI HW Enc and Dec on 1000 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW (ticks) Tics for data transfer	Kasumi HW (ticks) Tics for processing packets	Actual Kasumi HW (ticks) Processing time – Transfer time
1000	1000	64000000	6892905	27587723	27587722
5000	1000	40000000	4618690	139659480	135040790
8000	1000	8000000	224199	2234777755	216584850

Following figure shows the results of both the above cases (c and d), which are depicted graphically side by side in order to make it possible to see the behaviours of the KASUMI implementations with larger data size.

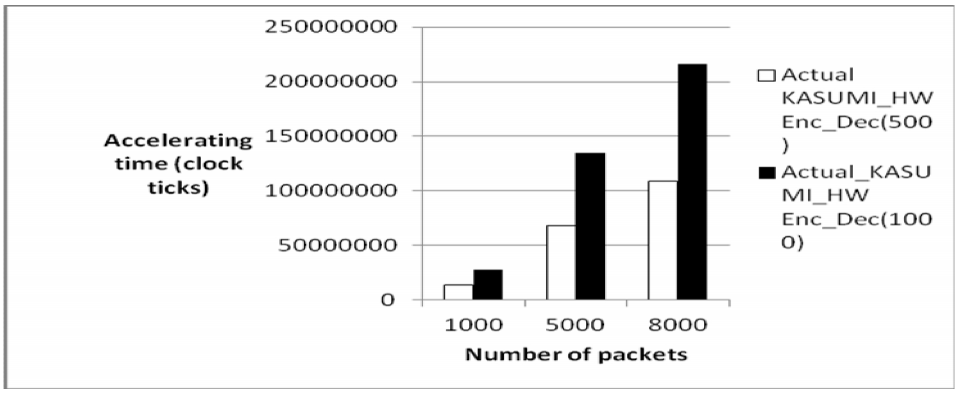


Figure 23. KASUMI HW Enc and Dec on 500 and 1000 byte packets

5.4 Software-based KASUMI Acceleration Case Simulation

In this subchapter, software-based based KASUMI implementation is executed in order to see closely, how encryption and decryption functions behave in different data lengths and data block packets. Similar tests were executed in previous subchapter (5.3) and thesis study will be carried out for close comparison between them in the analysis phase.

5.4.1 Case A: KASUMI SW Encryption and Decryption on 40 bytes packets

Data length of 40 byte is sent from one core to another core which accelerates received data by doing encryption and decryption. 8 different data blocks are used in this test case starting from 1000 to 8000. Following table shows the results of this experiment.

Number of Packets	Data length (byte)	Data size (bits)	Kasumi SW (ticks) Tics for data transfer	Kasumi SW (ticks) Tics for processing packets	Actual Kasumi SW (ticks) Processing time – Transfer time
1000	40	320000	219667	69210416	8370019
2000	40	640000	439167	60543156	17181882
3000	40	960000	660012	51835722	25113403
4000	40	1280000	870630	43080866	33513866
5000	40	1600000	1031142	34384496	42049724
6000	40	1920000	1193420	25773415	50642302
7000	40	2240000	3126027	17181882	57417129
8000	40	2560000	3285855	8589686	65924561

Table 8. KASUMI SW Enc and Dec on 40 byte packets

5.4.2 Case B: KASUMI SW Enc and Dec on 88 bytes packets

Data length of 88 bytes is sent from one core to another core which accelerates received data by doing encryption and decryption. 8 different data blocks are used in this test case starting from 1000 to 8000. Following table shows the result of KASUMI SW based implementation execution on 88 bytes data size.

Table 9. KASUMI SW Enc and Dec on 88 byte packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW (ticks) Tics for data transfer	Kasumi SW (ticks) Tics for processing packets	Actual Kasumi SW (ticks) Processing time – Transfer time
1000	88	704000	219539	16964627	16745088
2000	88	1408000	439118	33928772	33489654
3000	88	2112000	660020	50885730	50225710
4000	88	2816000	849864	67883728	67033864
5000	88	3520000	1010042	84962136	83952094
6000	88	4224000	2941342	102100446	99159104
7000	88	4928000	3101304	119189380	116088076
8000	88	5632000	3261291	136237677	132976386

Following figure shows the result of SW based KASUMI implementation on 40 bytes and 88 bytes data lengths depicted side by side in order to make it possible to see the difference between them very clearly.

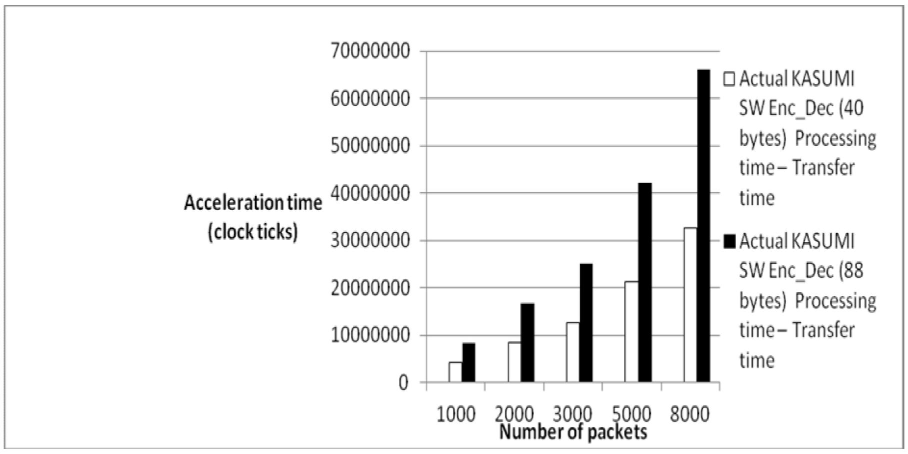


Figure 24. KASUMI SW Enc and D ec on 40 and 88 bytes packets

5.4.3 Case C: KASUMI SW Encryption and Decryption on 500 bytes packets

Data length of 500 bytes is sent from one core to another core which accelerates received data by doing encryption and decryption. 3 different data blocks are used in this test case starting from 1000 to 8000. Following table shows the result of KASUMI SW based implementation execution on 500 byte data length.

Table 10. KASUMI SW based implementation on 500 byte packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Dec (ticks) Tics for data transfer	Kasumi SW Dec (ticks) Tics for processing packets	Actual Kasumi SW Dec (ticks) Processing time – Transfer time
1000	500	4000000	219931	89638692	89418761
5000	500	20000000	2788516	449786178	446997662
8000	500	32000000	5068623	719635546	714566923

5.4.4 Case D: KASUMI SW Encryption and Decryption on 1000 bytes packets

Data length of 1000 bytes is sent from one core to another core which accelerates received data by doing encryption and decryption. 3 different data blocks are used in this test case starting from 1000 to 8000. Following table shows the result of KASUMI SW based implementation execution on 1000 data lengths.

Table 11. KASUMI SW Enc and Dec on 1000 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi SW (ticks) Tics for data transfer	Kasumi SW (ticks) Tics for processing packets	Actual Kasumi SW (ticks) Processing time – Transfer time
1000	1000	8000000	224329	1415238530	176330442
5000	1000	40000000	4611966	884500916	879888950
8000	1000	64000000	6889716	1415238530	1408348814

5.5 Key Generation/Expansion Effects on Test Cases

In this thesis work, KASUMI key setup implication is not tested due to the overall effects on the different implementations option. Smaller increase of clock cycles for different key setup doesn't have much number of cycles per word used for different key stream generation. For that reason key generation related test cases are left out from this thesis work.

5.6 Multi-core Test Cases

In this subchapter, test cases are configured differently than cases that have been handled in previous subchapter (5.4). Instead of using one core, following test cases will be used more than one core that is processing the data under the test simultaneously. In these test cases, all 16 cores of the OCTEAN processor are populated for running on both shorter and higher data sizes.

5.6.1 Shorter Data Size for Multi-core Test Cases

Following test cases that are designed for both HW and SW based KASUMI implementations; data size of 64 bytes is used for running more than one core for accelerating data under the test. In this test configuration, maximum data size of 96 bytes is possible due to FPA allocation setting. Since test is using more than one core, packet data are put into one queue, so different cores could fetch from it. The size of allocated FPA is to be handled for using more than one core.

5.6.1.1 KASUMI HW Encryption and Decryption

In this test case, all 16 core of OCTEON HW processor is accelerated for KASUMI encryption and decryption in order to see time used for processing cycles when used more than on core. Following figure shows the results of this experiment.

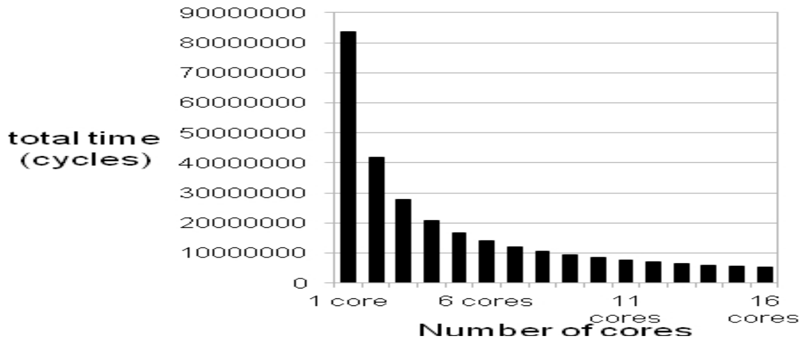


Figure 25. Multicore for KASUMI HW Enc and Dec on 64 byte packets

As shown in above figure, as number of core increases, processing cycles decrease accordingly. Another observation that this test case could be made from, is that when number of cores in used exceeds 5, then, it doesn't make much difference the decrease of processing cycles. Following figure shows increase of processing time linearly as number of core in use for HW acceleration increases.

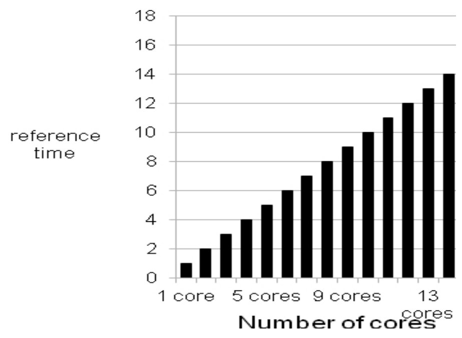


Figure 26. Reference time for KASUMI HW Enc and Dec on 64 byte packets

5.6.1.2 KASUMI SW Encryption and Decryption

The configuration of this test case is similar as in previous case but KASUMI software implementation is executed instead of KASUMI HW implementation for more than one core using 64 byte data size, following figure is shown processing cycles.

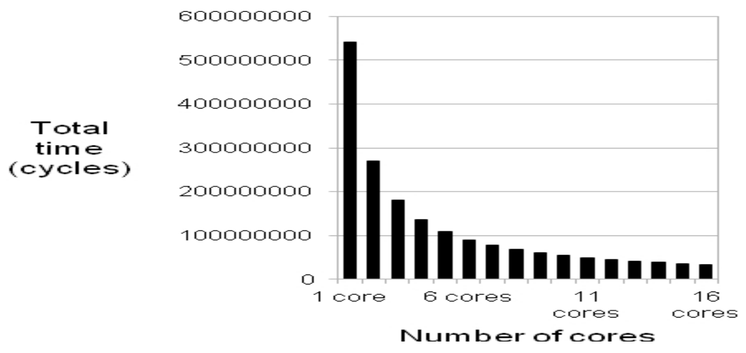


Figure 27. Multicore for KASUMI SW Enc and Dec on 64 byte packets

As shown in above figure, as number of core increases, processing cycles decrease accordingly. Another observation that this test case could be made from, is that when number of cores in used exceeds 5 as in the case KASUMI HW, then, it doesn't make much difference the decrease of processing cycles. Following figure shows increase of processing time linearly as number of core in use for SW implementation increases.

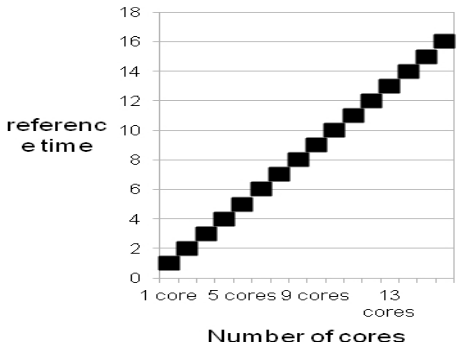


Figure 28. Reference time for KASUMI SW Enc and Dec on 64 byte packets

5.6.2 Higher Data Size for Multi-core Test Cases

Following test cases are intended to a measurement of the higher data size effects on both HW and SW based KASUMI implementations; data size of 512 bytes and 16 000 data blocks are used for running more than one core for accelerating data under the test.

5.6.2.1 KASUMI HW Encryption and Decryption

In this test case, all 16 core of OCTEON HW processor is accelerated for KASUMI encryption and decryption using on 512 byte for 16 000 data blocks in order to see time used for processing cycles when used more than on core. Following figure shows the results of this experiment:

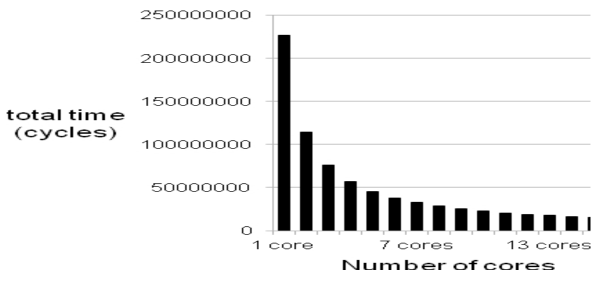


Figure 29. Multicore for KASUMI HW Enc and Dec on 512 byte packets

As shown in above figure, as number of core increases, processing cycles decrease. Another observation that this test case could be made from, is that when number of cores in used exceeds 5, then, it doesn’t make much difference the decrease of processing cycles as in the case of smaller data size. Following figure shows increase of processing time linearly as number of core in use for HW acceleration increases.

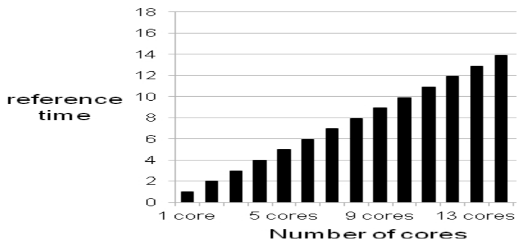


Figure 30. Reference time for KASUMI HW Enc/Dec on 512 byte packets

5.6.2.2 KASUMI SW Encryption and Decryption

The configuration of this test case is similar as in previous case but KASUMI software implementation is executed instead of KASUMI HW implementation for more than one core using 512 byte data size of 16 000 data blocks. Following figure is shown processing cycles.

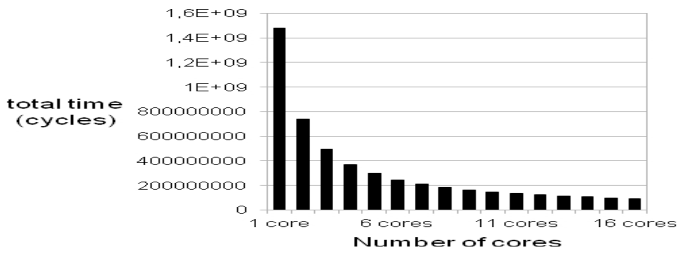


Figure 31. Multicore for KASUMI SW Enc/Dec on 512 byte packets

As shown in above figure, as number of core increases, processing cycles decrease but not after the fifth core used. Another observation is that in first three cores, number of cycles for processing is very high and that number decreases until fifth core is applied.

Following figure shows increase of processing time linearly as number of core in use for SW implementation increases:

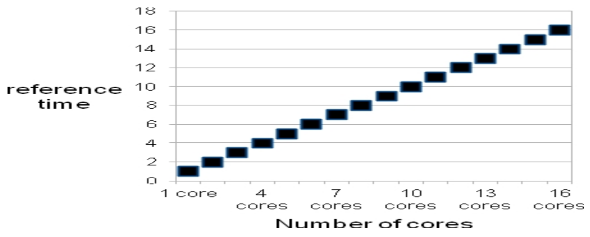


Figure 32. Reference time for KASUMI SW Enc/Dec on 512 byte packets

6. RESULT COMPARISONS AND DETAIL ANALYSIS

In this chapter, comparisons will be made between the test results. In each comparison, this thesis will also provide an explanation of why these results are as they are; in short, this thesis will also, if possible, perform another comparison between these experimental test results and the 3GPP reference results, as well as the results of other processor manufacturers, where possible.

6.1 Comparisons between KASUMI HW and KASUMI SW

These comparisons are divided into three main categories, namely, lower data comparison between two implementations with one core, a higher data rate with multi-core, and, last, a comparison with other implementations.

6.1.1 Lower Data Rate with One Core

In this subchapter, this thesis will carry out deep comparisons of how different implementations behave in data blocks of different sizes with smaller amount of data and applying only one core of the OCTEON processor; for that reason the following cases are checked closely.

6.1.1.1 KASUMI HW/SW Enc & Dec Comparisons with 40 bytes

In this comparison case, the basic KASUMI HW implementation is measured against the KASUMI SW implementation; both implementations executed encryption and decryption activities on the data under test. Different packet numbers are used for chunks of data with a size of 40 bytes. Performance is measured in terms of the number of clock scales taken per implementation to encrypt and decrypt the data. The figure below shows the result of the experiment:

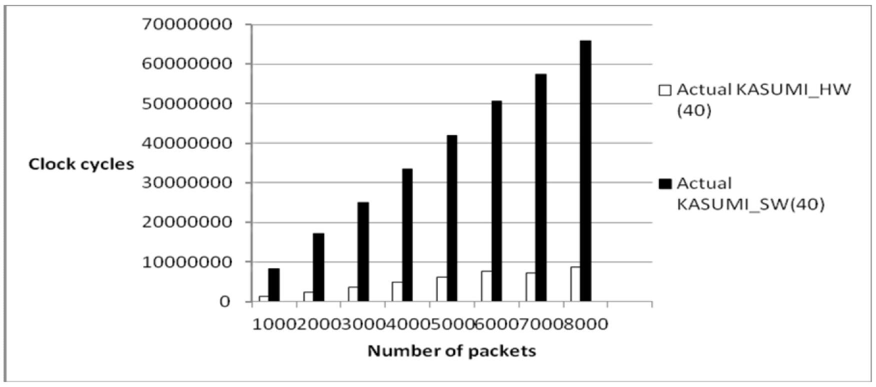


Figure 33. Number of cyles for HW/SW Enc/Dec on 40 byte packets

As shown in the figure above, the KASUMI SW implementation is much slower than the KASUMI HW implementation and this difference increases gradually as the packet block is doubled.

6.1.1.2 KASUMI HW/SW Enc and Dec Comparisons with 40 bytes

In this case, this thesis would like to look very carefully at the efficiency provided by different KASUMI implementations using a similar lower data rate to the one used in the previous case. In this case, the study aims to see the number of cycles per bit when different numbers of packets are used, in between1000 to 8000 data packet blocks, as shown in the table below.

Table 12. Number of Cycles per bit on 40 byte packets

Number of packets	KASUMI_HW	KASUMI_SW
	Cycles per bit	Cycles per bit
1000	3.808475	26.1563094
2000	3.80842031	26.8466906
3000	3.80874583	26.1597948
4000	3.82917734	26.1827078
5000	3.88168063	26.2810775
6000	3.95371146	26.376199
7000	3.23873705	25.6326469
8000	3.384275	25.7517816

The results of the experiment are shown in the figure below, which shows that an increase in the numbers of packets processed will not make any significant change to the processing cycles accomplished per bit. Both implementations show the same behavior in this matter. The KASUMI SW implementation in this test sitting took more than 7 times longer to process the same numbers of packets than the KASUMI HW acceleration did.

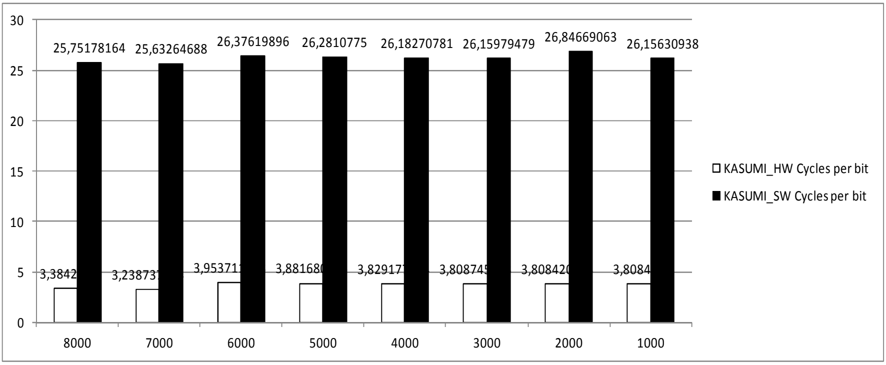


Figure 34. Number of cyles per bit on 40 byte packets

6.1.1.3 KASUMI SW Enc/Dec vs. only SW Enc/SW Dec for 40 bytes

In this test case, the relative efficiency of the KASUMI SW implementations is measured carefully in order to see how different functionalities affect the overall processing of KASUMI SW acceleration. Encryption and Decryption are measured against only either the encryption or decryption functions. The table below shows the test case results.

Table 13. KASUMI SW Enc/Dec Vs. Only Enc or Dec on 40 bytes

Number of packets	Actual KASUMI_SW Enc_Dec(40)	Actual KASUMI_SW only Enc(40)	Actual KASUMI_SW only Dec(40)
1000	8370019	4198057	4198070
2000	17181882	8395955	8396028
3000	25113403	12594960	12594962
5000	42049724	21180930	21180592
8000	65924561	32556081	32556190

As the figures below show, there is no significant difference between the encryption and decryption processing of KASUMI SW implementations with smaller amount of data size in all the different packet blocks.

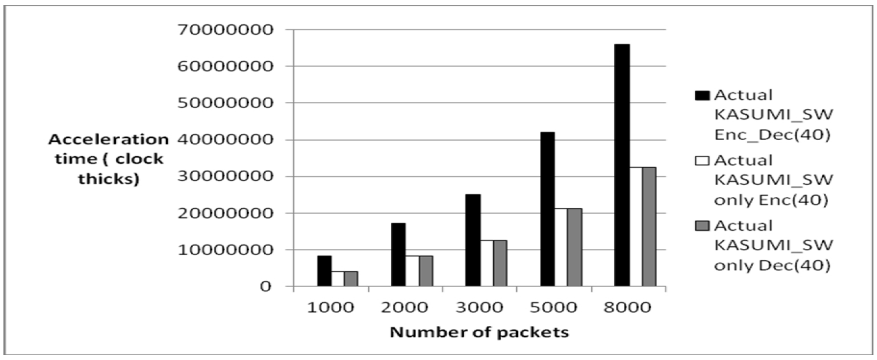


Figure 35. Comparison between KASUMI SW implementations

6.1.1.4 KASUMI HW Enc/Dec vs. only HW Enc & HW Dec on 40 bytes

In this test case, the relative efficiency of the HW implementations is measured in order to see how different functionalities affect the overall processing of KASUMI HW acceleration. Encryption and Decryption are measured against only either encryption or decryption. The table below shows the test case results.

Table 14. KASUMI HW Enc/Dec vs. Enc or Dec on 40 bytes

Number of packets	Actual KASUMI_HW Enc_Dec(40)	Actual KASUMI_HW only Dec(40)	Actual KASUMI_HW only Enc(40)
1000	1218712	624002	624005
2000	2437389	1247957	1248061
3000	3656396	1872193	1872460
5000	6210689	3184285	3184161
8000	8663744	3687658	3691216

As shown in the figure below, there is not much difference in the encryption or decryption activities of the KASUMI HW implementation with smaller amount of data, regardless of an increase in the size of the data blocks that are involved in the processes.

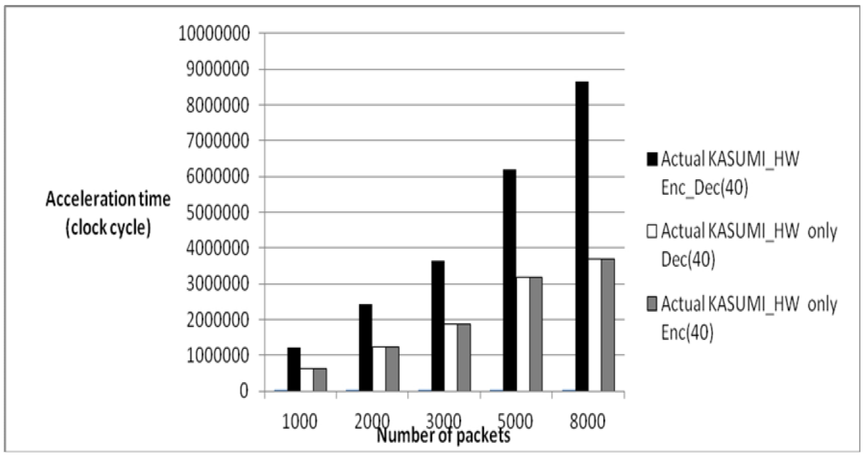


Figure 36. Comparison between KASUMI HW implementations

6.1.1.5 KASUMI HW and SW Enc/Dec for 1000 bytes

In this test case, thesis study would like to test theoretically the effect that an increase in the size of the chunks of data to 1000 bytes has on both implementations and how this larger amount of data size could affect the data-processing effectiveness of both KASUMI implementations. The figure below shows the results of the experiment.

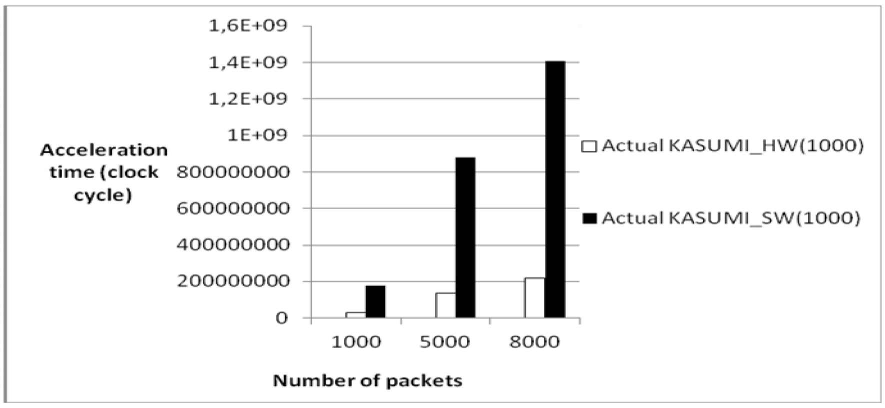


Figure 37. Number of cyles for HW/SW Enc/Dec on 1000 byte packets

As the figure above shows, the KASUMI SW implementation is hugely affected by the larger data packets compared to the HW implementation case. As the size of the data block increases, so does the number of cycles needed for encryption and decryption.

6.1.1.6 KASUMI HW Enc/Dec vs. only Enc and Dec for 1000 bytes

In this test case, this thesis will look very closely at where the processor consumes most when processing the encryption of data and decrypting again while processing the larger amount of data packets. The KASUMI encryption and decryption processing cycles are measured against only encryption or decryption processing cycles. The result of the experiments is shown in the figure below.

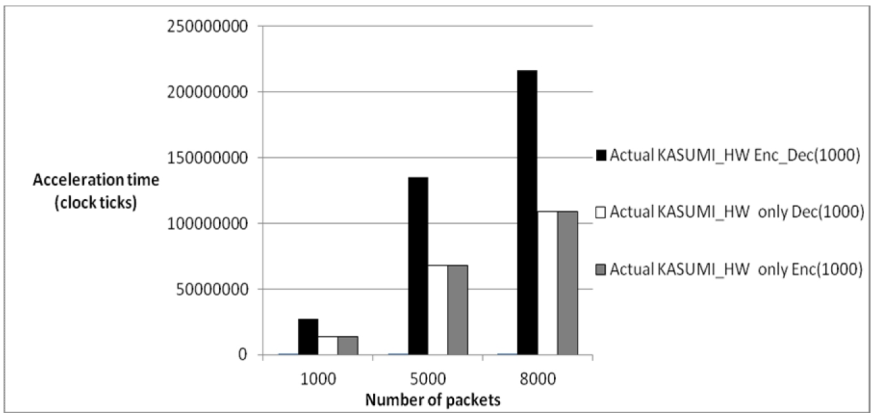


Figure 38. Comparison between KASUMI HW implementations

As the figure above shows, there is no difference between the encryption and decryption of KASUMI HW acceleration with regard to larger amount of data or even an increase in the sizes of the data blocks.

6.1.1.7 KASUMI SW Enc/Dec vs. only SW Enc & SW Dec on 1000 bytes

A similar test case to the KASUMI HW implementation is performed for the KASUMI SW implementation with large amount of data size of 1000 bytes, where the encryption and decryption activities are measured separately. The figure below shows the result of the experiment.

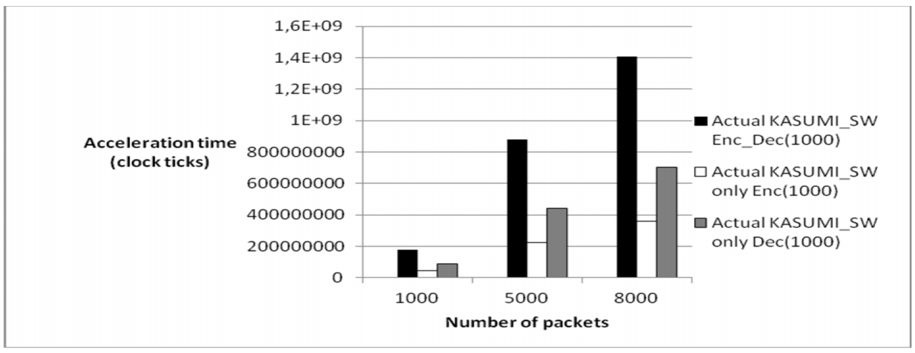


Figure 39. Comparison between KASUMI SW implementations

As the figure above shows, decryption with the KASUMI SW implementation took twice as many processing cycles as encryption processing and this difference increases as the sizes of the data blocks increase.

6.1.2 Higher Data Rate with Multi-core

In this subchapter, this thesis would like to look deeply at how different implementations of KASUMI algorithms behave with a higher data rate (512 byte) when the test of processing is allocated to multi-cores instead of one core, as in the previous subchapter (6.1.1).

6.1.2.1 Comparison between KASUMI HW/SW Enc/Dec with 16K

In this case, the thesis chose to allocate all the 16 cores of the processor under the test for executing encryption and decryption activities and the total time taken to perform Encrypting and Decrypting on 512 byte date rate with data blocks of 16K. The processing is measured in cycles. The results are depicted in the table below.

Table 15. KASUMI HW/SW Enc and Dec on 512 bytes with 16K data blocks

Number of Cores	KASUMI HW Enc/Dec	KASUMI SW Enc/Dec
1 core	226267770	1479950202
2 core	114029706	740598295
3 core	76032894	493725111
4 core	57025474	370315046
5 core	45621119	296256279
6 core	38026777	246920450
7 core	32593830	211648620
8 core	28520206	185178950
9 core	25354455	164615296
10 core	22828187	148147723
11 core	20749230	134715937
12 core	19038818	123520021
13 core	17574271	113980502
14 core	16311631	105836316
15 core	15227210	98799627
16 core	14294010	92594551

The figure below shows a comparison between the two implementations when all the 16 cores are populated with above shown 512 byte long packets.

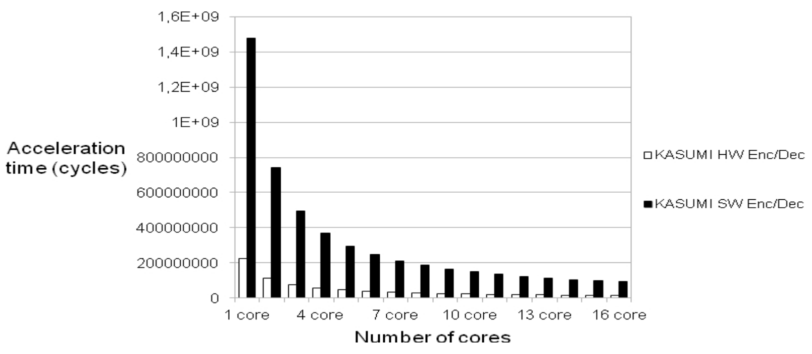


Figure 40. Number of cyles for HW/SW Enc/Dec on 512 byte packets with 16K

As shown in the figure above, the KASUMI SW implementation is far slower than the KASUMI HW implementation and this difference decreases gradually as the number of cores used increases.

6.1.2.2 Comparison between KASUMI HW Enc/Dec with 24K

In this case, the thesis increases the data block size from 16K to 24K, in order to see how HW acceleration behaves with this load. The table below shows the results of the experiment.

Table 16. KASUMI HW Enc/Dec on 512 byte with 24K data blocks

Number of Cores	total time (cycles)
1 core	339387964
2 core	170993785
3 core	114001301
4 cores	85508565
5 core	68407334
6 core	57008174
7 core	48868814
8 core	42754498
9 core	38022262
10 core	34214159
11 core	31103754
12 core	28523310
13 core	26342940
14 core	24466989
15 core	22839632
16 core	21418915

The figure below shows the total time for processing data packets of 24K in size with 512 bytes. The capacity of the processor is calculated to be 977.4 MB, or 0.9 Gbits.

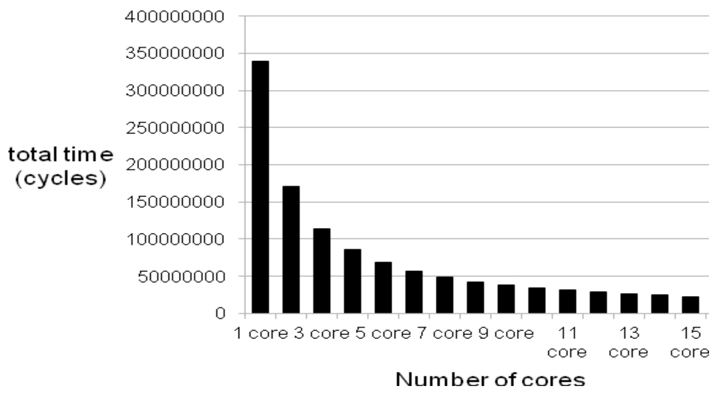


Figure 41. KASUMI HW Enc/Dec on 512 byte with data blocks of 24K

6.1.2.3 Comparison between KASUMI HW Enc/Dec with 64K

In this test case, the number of data blocks is increased again from 24K to 64K with same data packet of 512 byte and memory shortage is noticed in this case. In order to run this test case successfully, memory size should be increased to 2KB in order to handle the buffer allocation for the date. The results of the experiment are shown in the table below.

Table 17. KASUMI HW Enc/Dec on 512 byte with 64K data blocks

Number of Cores	total time (cycles)
1 core	924712655
2 core	462384033
3 core	308261050
4 cores	231196454
5 core	184951438
6 core	154137398
7 core	132114097
8 core	115606133
9 core	102767678
10 core	92500483
11 core	84087311
12 core	77098218
13 core	71163047
14 core	66081000
15 core	61674809
16 core	57831752

As shown in the figure below, with data block sizes of 64K to be handled, the memory should be sufficient for that reason; the memory was increased to 2KB for this test case.

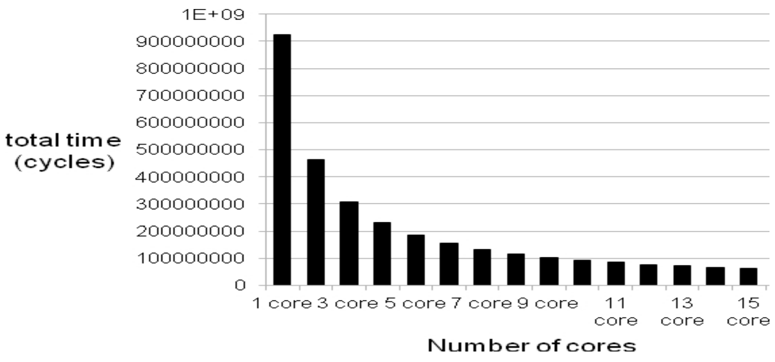


Figure 42. KASUMI HW Enc/Dec on 512 byte data packet of 64K

7. CONCLUSIONS AND FUTURE WORK

In this thesis work, a 3G security architecture is outlined, and especially the 3G KASUMI algorithm is defined. Chapter 3 is studied in order to see KASUMI HW-based acceleration performance in comparison with the KASUMI Software implementation. After a brief discussion of the main principle of the OCTEON processor and its architecture, Thesis work has been concentrated on experimental test cases that are used to evaluate the performance of the implementations of the KASUMI cipher.

Thesis investigation is mainly of comparisons of the performance of the two implementations and what the other parameters is that may affect the performance. The experimental work performed in this thesis enabled the following conclusions to be drawn that demonstrate the superiority of the KASUMI HW-based implementation to the SW-based implementation:

1- The KASUMI SW-based implementation is much slower than the KASUMI HW-based implementation and this difference increases gradually as the packet block size is doubled;

2- An increase in the packet size processed does not make any significant change to the number of processing cycles per bit for both implementations;

3- With lower data rates, neither of the KASUMI implementations shows much difference between encryption or decryption processing functions, regardless of an increase in the amount of data that are involved in the processes;

4- An increase in the number of data packets has profound effects on the data processing effectiveness of both the KASUMI implementations but this effect is especially huge in the SW-based implementation;

5- When all the 16 cores of the OCTEAN processor are populated, as the number of core increases, the processing cycles decrease accordingly. Another observation was that when the number of cores in use exceeds 5, it does not make much difference to the reduction in the number of processing cycles;

6- As the number of data blocks increases, the memory allocation for buffering becomes essential; for instance, when data blocks of more than 24K are being processed, it requires at least a 2K byte cache size memory allocation;

7- Hardware-based KASUMI acceleration experiments shows that it is not necessary to allocate more than one or two cores for the KASUMI algorithm handling task, while other cores could be allocated to other system functionalities.

Finally, Thesis wish to suggest some directions for further research, For instance, the SW-based KASUMI implementation should be optimised at least partially (blocks), if not the whole implementation. Second, in order to see where the resources of the processor are used most in the KASUMI SW implementation case, for further optimisation, if needed, it would be good to perform detailed tests on the internal processes of the SW implementation by using the tools that the OCTEON analysis software toolkit offers.

Last, but not least, the current KASUMI software implementation used in these experimental tests is not fast enough and cannot be used unless it can be significantly optimized.

REFRENCES

3GPP TS 23.101. General Universal Mobile Telecommunications System (UMTS) Architecture (2007 v7.0.0)

3GPP TS 23.110. General Universal Mobile Telecommunications System (UMTS) Access Stratum (2007 v7.0.0)

3GPP TS 23.002. Technical Specification Group Services and Systems Aspects; Network Architecture (2007 v7.1.0)

3GPP TR 23.930. Technical Specification Group Services and Systems Aspects; Iu Principles (1999 v3.0.0)

3GPP TS 23.930. Technical Specification Group Core Network and Terminals; Mobile radio interface Layer 3 Specification; Core network protocols; Stage 3 (2010 v7.15.0)

3GPP TS 25.420. Technical Specification Group Services and Systems Aspects; UTRAN Iur Interface General Aspects and Principles (2007 v7.3.0)

3GPP TS 25.401. Technical Specification Group Radio Access Network; UTRAN Overall Description (2002 v3.10.0)

3GPP TS 25.301. Technical Specification Group Radio Access Network; Radio Interface Protocol Architecture (2006 v7.3.0)

3GPP TS 25.331. Technical Specification Group Radio Access Network; Radio Resource Control (RRC) Protocol Specification (2006 v7.3.0)

3GPP TS 33.102 . Technical Specification Group Services and Systems Aspects; 3G Security; Security Architecture (2006 v7.1.1)

3GPP TS 21.133. Technical Specification Group Services and Systems Aspects; 3G Security; Security Threats and Requirements (2002 v4.1.0)

3GPP TS 33.120. Technical Specification Group Services and Systems Aspects; 3G Security; Security Principles and Objectives (2001 v4.0.0)

3GPP TS 35.201. Technical Specification Group Services and Systems Aspects; 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms (2007 v7.0)

3GPP TS 33.105. Technical Specification Group Services and Systems Aspects; 3G Security; Cryptographic Algorithm Requirements (2007 v7.0.0)

3GPP TS 35.202 . Technical Specification Group Services and Systems Aspects; 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms: Document 2: KASUMI Specification (2007 v7.0.0)

Agar, John (2005). Constant Touch: A Global history of the Mobile Phone. Illustrated, reprint. Icon: London.180p. ISBN 1840465417

Arkko, J & Haverinen H, (2006). Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA[online]. IETF :RFC 4187.[cited 20.10.2010], Available from internet: http://tools.ietf.org/html/rfc4187

Bernhard H. Walke, P. Seidenberg, M.P Althoff (2003). UMTS: The UMTS: The fundamentals. Illustrated edition. Michigan: WILEY. 312 p. ISBN 0470845570

Cavium Networks (2009). Security Processors(Product Brief). Available from internet at: http://www.cavium.com/pdfFiles/NITROX_PX_PB Rev 1.2.pdf. Referenced on16.08.2010

Cavium Networks (2009). OCTEON II Internet Application Processor (AIP) Family (Product overview). Available at : http://www.caviumnetworks.com/OCTEON_II_MIPS64.html referenced on 6.10.2010

Cavium Networks (2009c). Product Table. Available at: http://www.caviumnetworks.com/Table.html. referenced on 16.10.2010

Cavium Networks (2009d). OCTEON MIPS64 Processors Architecture Advantages, Cavium Networks internal document

Cavium Networks (2009e) OCTEON CN58XX-HRM-1.2 Manuals from Cavium Networks Tech Support Site. referred to 18.9.2010

Cavium Networks (2009f). OCTEON User´s Manual, Cavium Networks internal documents. referred to 18.11.2010

Cavium Networks (2009i). OCTEON Programmer’s Guide, Cavium Networks internal document. referred to 20.11.2010

Cavium Networks (2009g). Quick Start Guide for the OCTEON Software Development Kit (SDK) and Evaluation Boards (EVB), version 1.9, May 5, 2009

Freescale Semiconductor (2008). Understanding Cryptographic Performance. Rev 2.0

H.Kim, Y.Choi, M.Kim and H.Ryu (2002). Hardware Implementation of the 3GPP KASUMI Crypto Algorithm. In Proceeding. of the 2002 International Technical Conference on Circuit/Systems, Computers and Communication ITC-CSCC pp.317-320

Horn. Günther, P. Howard & Niemi Valtteri (2002).UMTS Security Mechanism. In Electronic & Communication Engineering Journal, Volume 14, Issue 5, 191-204

Gardner J.Scott. Integrated CPU cryptography acceleration secures small computers. available at http://pdf.cloud.opensystemsmedia.com/smallformfactors.com/Advantage.RG07.pdf, referred on 3.4.2012

G.Kambourakis, A Rouskas, and S.Gritzalis (2004). Performance Evaluation of Public Key-Based Authentication in Future Mobile Communication Systems. In: EURASIP Journal on Wireless Communications and Networking, Volume 1, page 184-197

G.F. Grecas, S.I.Maniatis, and I.S. Vernieris (2003). Introduction of the Asymmetric Cryptography in GSM, GPRS, UMTS, and its Public Key Infrastructure Integration. In: Mobile Networks and Applications, Volume 8, Number 2, 145-150

Geoff, Waters and Kurt, Stammberg (2009). Understanding Crypographic Performance Part 1 and Part 2, [online] Embedded: Cracking the code to systems development. Available at: http://www.embedded.com/design/218400877 referred on 08.08.2010

Jääskeläinen, Jukka (2003). Performance evaluation of software ciphering in UMTS radio network controller, Master thesis, Helsinki University of Technology

Kaaranen. Heikki, Ahtiainen. Ari, Laitinen. Lauri, Siamäk Naghian and Niemi. Valtteri (2001). UMTS Networks; Architecture, Mobility, and Services .John Wiley & Sons. 302 pages

MIPS Technology (2001). MIPS64 Architecture For Programmers: Volume I: Introduction to the MIPS64 Architecture, Revision 2.0, Nov 5, 2010. Available at: http://scc.ustc.edu.cn/zlsc/lxwycj/200910/W020100308600768363997.pdf

Mitsuru, Matsui (1997). New Block Encryption Algorithm MISTY. In Proceeding of the 4th International Workshop on Fast Software Encryption, Vol. 1267 of LNCS, Springer Verlag, 54-68

National Institute of Standards and Technology (1981). DES Modes of operation [online]. Available from internet: http://www.itl.nist.gov/fipspubs/fip81.htm

Niemi, Valtteri & Nyberg, Kaisa (2006).UMTS Security. John Wiley & Sons. 286p. ISBN 9780470091562

Theodore S, Rappaport(2002). Wireless Communications: Principles and Practice. 2nd Edition. Prentice Hall PTR. 707 pages

Timo Alho (2006). Cryptographic Hardware Implementations for Embedded Devices. M.Sc. Thesis, 2006, 63 pages, Tampere University of Technology

Tomás Balderas-Contretras (2004). Hardware/Software Implementation of the Security Functions for Third Generation Cellular Networks. Master Thesis, Insituto Nacional de Astrofísica, Óptica y Electrónica, MEXICO

Tomás, Balderas-Contretras, and René, Cumplido (2005) . High performance Encrption cores in 3G networks. In Proceeding of the 42nd annual Design Automation Conference, ACM. DAC ´05: 240-243

V.H, Macdonald (1979). The Cellular Concept. Bell System Technical Journal, 58:1, 15-43

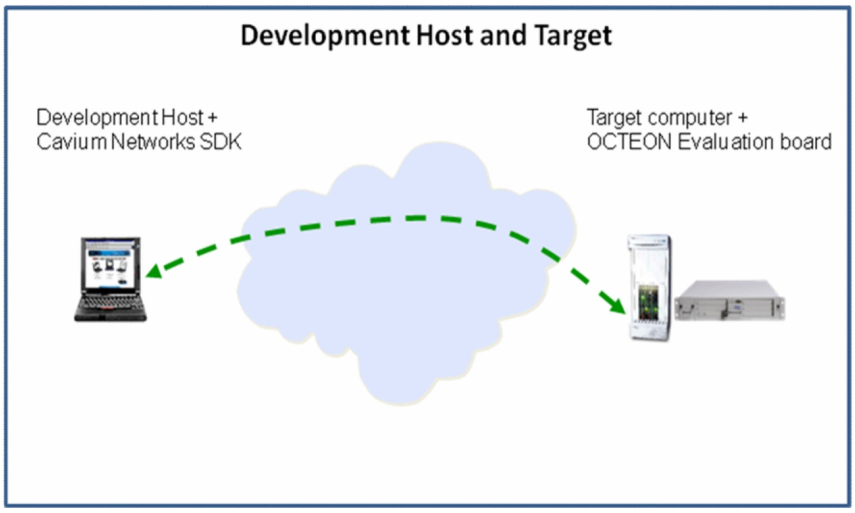
APPENDIX 1. Testing Environment

Test environment consists of following components:

- **Development Host:** The term development host will be used to describe the x86_64 machine which is used as a cross-development platform. In this test experiment, Linux based DELL Laptop (Latitude D610) is used.
- **Development Target:** The term development target refers to be OCTEON evaluation board connected to the development host which has a KASUMI hardware accelerator unit.
- **Software under the test: Software under the test or** to be tested software is 3GPP based KASUMI cipher software implementation. Tests contain two different KASUMI implementations. The first one is based on 3GPP reference implementation from (3GPP TS 35.202, 2007) with several SW performance improvements. SW based performance improvements are described in detail in section 4.2. The second KASUMI implementation will include sections that will take benefit of Octeon HW based security accelerator. Details of OCTEON HW accelerator based implementation are described in section 4.1.
- **Software Development Kit (SDK) of Cavium Networks:** The SDK is software consists of two software packages, the base SDK and OCTEON Linux that is based on Debian. This SDK is used for compiling the source application in order to run on OCTEON processor.
- **Test Cases:** Test cases consist of number of configuration test cases, that to be tested in order to verify the 3GPP reference results as well as to satisfy the existing product requirements. Header and script files which are part of tested software and are used to help to configure the test case setups.

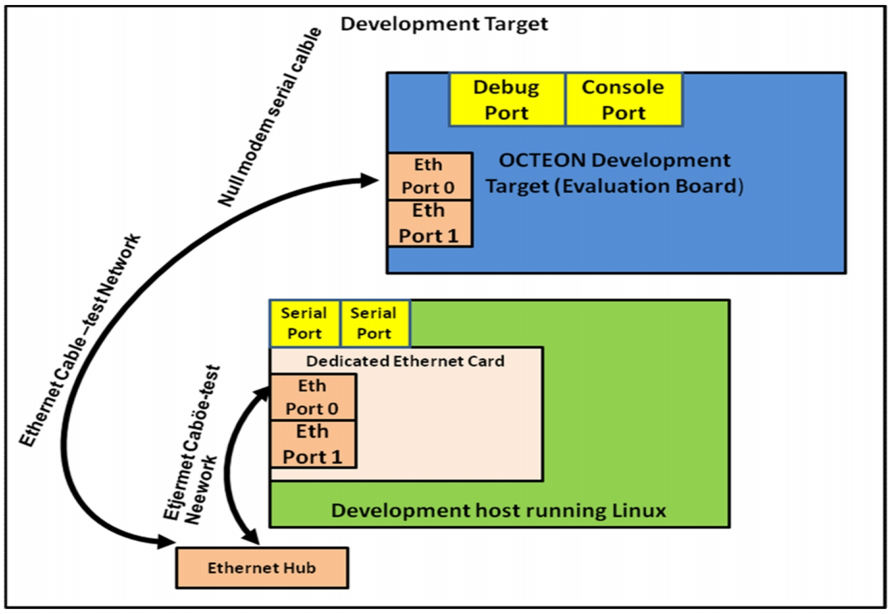
KASUMI application that is to be tested is built on the development host. Once the application is built, it will be downloaded to stand-alone development target board connected to the development host. Following figure illustrates the high level view of the test environment setting.

Figure A1. Test Environment configuration


Development Host and Target
Development Host + Cavium Networks SDK
Target computer + OCTEON Evaluation board

Detail description of how development target is setup is shown in following figure.

Figure A2. Detail view of development target



In above shown figure, OCTEON Development Target consists of One PC that are used for multiple purpose, which is acting as normal Server machine that has OCTEON evaluation board, TFTP Server and has a dedicated Ethernet connection to Development Host.

Development Host is x86_64 laptop platform computer which has NSN Red Hat Enterprise Linux operating system. It has Cavium Networks SDK software, Kasumi software application, and Testing scripts and test cases.

Test system it is not configured to have an isolated Ethernet connection to Target board which is separate from office Ethernet but it has been carried out through remote connection from office to laboratory.

System Administration

Create a personal account on the development host and login this user and do all the work as this user name. It is important to obtain *root* permission on the development host, the reason is that it is needed for SDK installation, to build Linux kernel and to use the PCI tools.

Lastly, t is import to remember that if user´s home directory is on a NFS-mounted drive, then user must also have a separate non-NFS-mounted workspace.

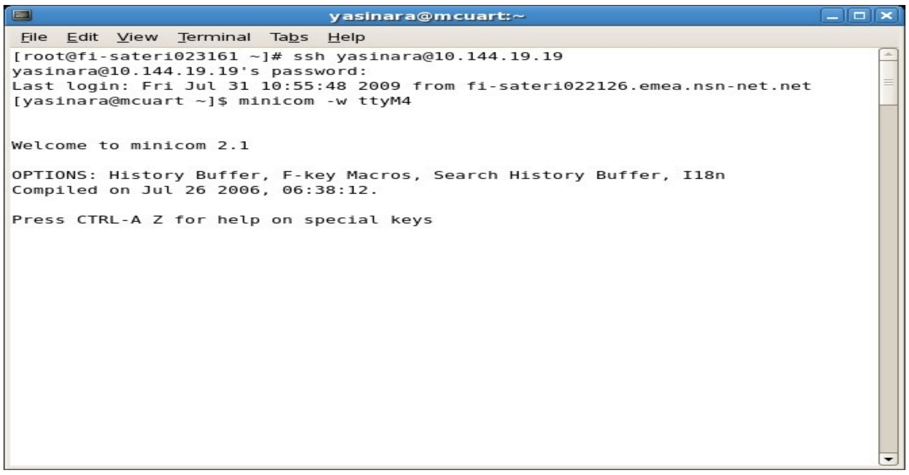
Quick Start Guide (Cavium Networks, 2009g) document that is provided by Cavium Networks describes how Development Host is set-up and be configured and connects to Development Target.

Viewing the Target Board Console Output

The easiest way to view the target board console output is by running the Minicom utility on the host, and connecting to the target board via a null-modem serial cable. The console output for the target board is directed to UART0 on the target board. UART0 is connected via a serial cable to the first port on the development host. Linux connects to the fourth serial port on the device */dev/ttyM4*.

Following snapshot shows the minicom console running on host computer:

Figure A3. Console running on host computer



```
[root@fi-sateri023161 ~]# ssh yasinara@10.144.19.19
yasinara@10.144.19.19's password:
Last login: Fri Jul 31 10:55:48 2009 from fi-sateri022126.emea.nsn-net.net
[yasinara@mcuart ~]$ minicom -w ttyM4


Welcome to minicom 2.1

OPTIONS: History Buffer, F-key Macros, Search History Buffer, I18n
Compiled on Jul 26 2006, 06:38:12.

Press CTRL-A Z for help on special keys
```

Install the SDK

Installation CD which accompanies the evaluation board contains the SDK (The base SDK and OCTEON Linux) and other files. There are optional products also included like support of special hardware capabilities on OCTEON: Crypto, ZIP and DFA. Also Software to implement packet send/receive over the PCI bus, etc. Detail description of Installation can be referred to SDK installation document from Cavium Networks or Installing from the Support Site instead of a CD" from Cavium Networks´s site.

In this thesis work tests are carried out with SDK 1.8.0. After the correct installation, $*OCTEON_ROOT* is working directory and it needs to do couple of environment variable value settings. Following variables are set by the *env-setup* script:

1- $*OCTEON_ROOT*- This variable will be set to the directory the env-setup script is executed.

2- The *PATH* environment variable is modified to add the directories containing the tool chain binaries and development host utilities

3- OCTEON_MODEL [add used model]

4- OCTEON_CPPFLAGS_GLOBAL_ADD

Building Applications

Application is created by using Makefiles. These are typically files named *makefile, Makefile,* or *.mk (as in *cvmx.mk* or *application .mk*). A Makefile can be included other Makefiles.

When application is built, first Simple Executive code located in the $ *OCTEON_ROOT/executive* directory is built, and the archive program (ar) package the Simple Executive object files into one library: *libcvmx.a.* The object files and library put in application directory */obj* directory. In SE-S application, there are two possible Makefile targets and the target is selected by setting OCTEON_TARGET equal to the desired target on the *make* command line.

SE-S Targets:

- *OCTEON_TARGET = cvmx_64*
- *OCTEON_TARGET = cvmx_32*

Each target is built with the object files in separate directory and each ELF file has a unique name. In this work 64-bit applications were used.

APPENDIX 2. More test case scenarios

A2.1 Hardware based KASUMI acceleration case simulation

a) KASUMI HW Encryption on 40 bytes packets

In this test case, HW encryption in accelerated on data length of 40 bytes in order to see how long it takes to process only encryption functions. Following table shows results how KASUMI HW Encryption behaviours when data length is 40 byte.

Table A1. KASUMI HW Encryption on 40 byte packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi HW Enc (ticks) Tics for data transfer	Kasumi HW Enc (ticks) Tics for processing packets	Actual Kasumi HW Enc (ticks) Processing time – Transfer time
1000	40	320000	219612	843617	624005
2000	40	640000	439118	1687179	1248061
3000	40	960000	660397	2532857	1872460
5000	40	1600000	1031367	4215528	3184161
8000	40	2560000	3285773	6976989	3691216

b) KASUMI HW Encryption on 88 bytes packets

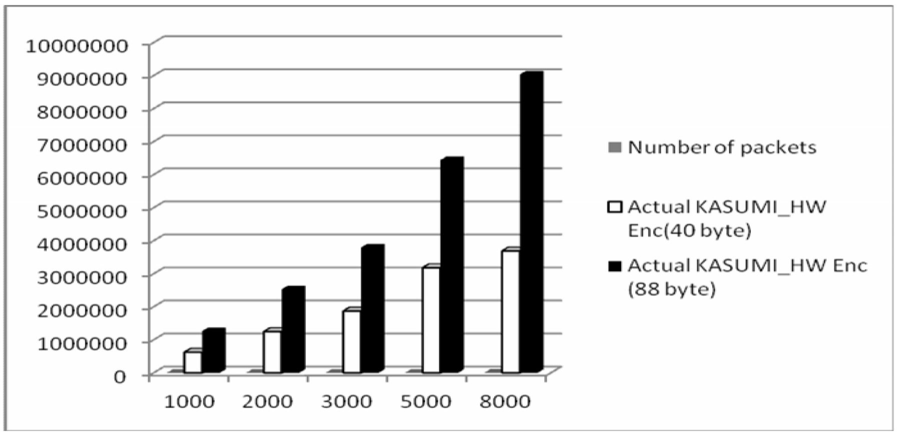
In this test case, the HW encryption is accelerated on data length of 88 bytes in order to see how long it to process encryption functions. Following table shows results of KASUMI HW Encryption behaviours when data block size increases to 88 byte.

Table A2. KASUMI HW Encryption on 88 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW Enc (ticks) Tics for data transfer	Kasumi HW Enc (ticks) Tics for processing packets	Actual Kasumi HW Enc (ticks) Processing time – Transfer time
1000	88	704000	219655	1484582	1264927
2000	88	1408000	439125	2969078	2529953
3000	88	2112000	660238	4455994	3795756
5000	88	3520000	1009960	7453475	6443515
8000	88	5632000	3260919	2296532	9035613

Following graph is depicted both HW Encryption processes in 40 byte and 88 byte.

Figure A4. KASUMI HW Encryption on 40 and 88 byte packets



c) KASUMI HW Encryption on 500 bytes packets

In this test case, the HW encryption is accelerated on data length of 500 bytes in order to see how long it to process encryption functions. Following table shows results of KASUMI HW Encryption behaviours when data block size increases to 500 byte. The result of this test case is depicted in following table:

Table A3. KASUMI HW Encryption on 500 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi HW Enc (ticks) Tics for data transfer	Kasumi HW Enc (ticks) Tics for processing packets	Actual Kasumi HW Enc (ticks) Processing time – Transfer time
1000	500	4000000	220294	7120846	6900552
2000	500	8000000	450726	14266298	13815572
3000	500	12000000	692503	21982685	21290182
5000	500	20000000	2788925	37204829	34415904
8000	500	32000000	5068858	59649301	54580443

d) KASUMI HW Encryption on 1000 bytes packets

In this test case, the HW encryption is accelerated on data length of 1000 bytes in order to see how long it to process encryption functions. Following table shows results of KASUMI HW Encryption behaviours when data block size increases to 1000 byte.

Table A4. KASUMI HW Encryption on 1000 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW Enc (ticks) Tics for data transfer	Kasumi HW Enc (ticks) Tics for processing packets	Actual Kasumi HW Enc (ticks) Processing time – Transfer time
1000	1000	8000000	224313	14150973	13926660
2000	1000	16000000	478345	28760045	28281700
3000	1000	24000000	2442241	43351593	40909352
5000	1000	40000000	4612936	72499959	67887023
8000	1000	64000000	6890433	116019725	109129292

The results of the experiments of cases are c and d is depicted side by side in following graph:

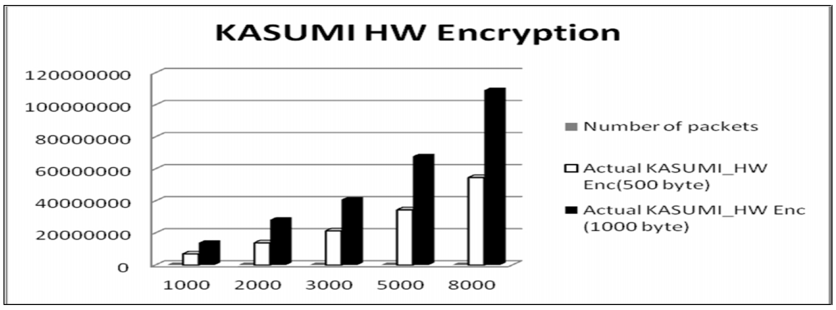


Figure A5. KASUMI HW Encryption on 500 and 1000 byte packets

e) KASUMI HW decryption on 40 bytes packets

In this test case, HW decryption in accelerated on data length of 40 bytes in order to see how long decryption function behaves with different data block sizes. Following table is depicted the test case result.

Table A5. KASUMI HW Decryption on 40 byte packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi HW Dec (ticks) Tics for data transfer	Kasumi HW Dec(ticks) Tics for processing packets	Actual Kasumi HW Dec (ticks) Processing time – Transfer time
1000	40	320000	219612	843614	624002
2000	40	640000	439176	1687133	1247957
3000	40	960000	660397	2532590	1872193
5000	40	1600000	1031367	4215652	3184285
8000	40	2560000	3285773	6973431	3687658

f) KASUMI HW decryption on 88 bytes packets

In this test case, the HW decryption is accelerated on data length of 88 bytes in order to see how decryption functions with different data block sizes. Following table shows the result of HW decryption acceleration:

Table A6. KASUMI HW decryption on 88 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW Dec (ticks) Tics for data transfer	Kasumi HW Dec (ticks) Tics for processing packets	Actual Kasumi HW Dec (ticks) Processing time – Transfer time
1000	88	704000	219655	1484572	1264917
2000	88	1408000	439125	2969021	2529896
3000	88	2112000	660238	4456323	3796085
5000	88	3520000	1009960	7452364	6442404
8000	88	5632000	3260919	12297176	9036257

Following graph shows the results of KASUMI HW accelerations in different data lengths 40 and 88 bytes in order to see closely how encryption process react to different data sizes when data block load is same.

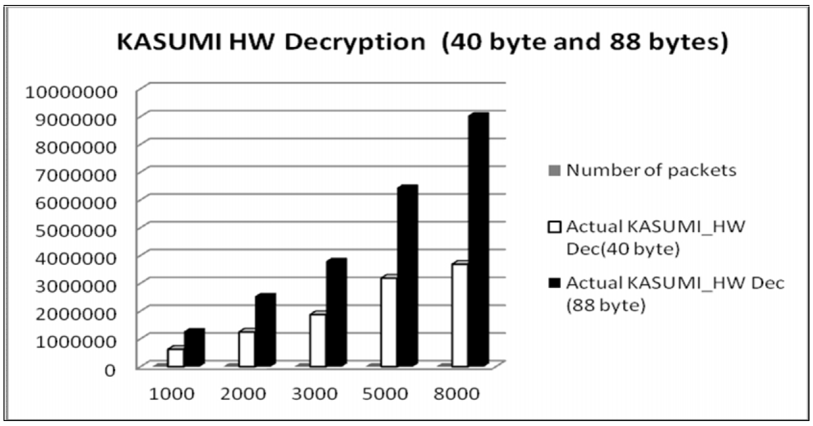


Figure A6. KASUMI HW decryption on 40 and 88 bytes packets

i) KASUMI HW Decryption on 500 bytes packets

In this test case, the HW decryption is accelerated on data length of 500 bytes in order to see how decryption function with different data blocks sizes decryption. Following table shows the result of the experiment when executed:

Table A7. KASUMI HW decryption on 500 bytes packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW Dec (ticks) Tics for data transfer	Kasumi HW Dec (ticks) Tics for processing packets	Actual Kasumi HW Dec (ticks) Processing time – Transfer time
1000	500	4000000	220294	7121027	6900733
2000	500	8000000	450726	14266655	13815929
3000	500	12000000	692503	21982651	21290148
5000	500	20000000	2788925	37207483	34418558
8000	500	32000000	5068858	59649185	54580327

g) KASUMI HW Decryption on 1000 bytes packets

In this test case, only difference with previous case is that HW decryption is accelerated on data length of 1000 bytes in order to see how decryption function when executed with large data blocks decryption.

Table A8. KASUMI HW decryption on 1000 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi HW Dec (ticks) Tics for data transfer	Kasumi HW Dec (ticks) Tics for processing packets	Actual Kasumi HW Dec (ticks) Processing time – Transfer time
1000	1000	8000000	224313	14151830	13927517
2000	1000	16000000	478345	28759716	28281371
3000	1000	24000000	2442241	43351410	40909169
5000	1000	40000000	4612936	72499744	67886808
8000	1000	64000000	6890433	116018790	109128357

Following graph shows the results of KASUMI HW accelerations in different data lengths of 500 and 1000 byte to see closely how encryption process react to different data sizes when data block load is same.

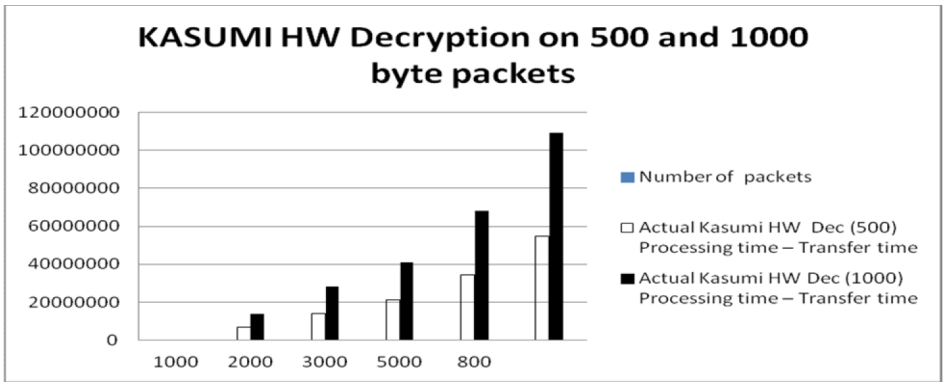


Figure A7. KASUMI HW Decryption on 500 and 1000 byte packets

A2.2 Software based KASUMI acceleration case simulation

a) KASUMI SW Encryption on 40 bytes packets

In this test case, only difference with previous cases is that only SW encryption is accelerated on data length of 40 bytes in order to see which one takes more time encryption or decryption functionalities. Following table shows the result of KASUMI SW based encryption implementation execution on 40 data lengths

Table A9. KASUMI SW Encryption on 40 byte packets

Number of Packets	Data length (byte)	Data size (bits)	Kasumi SW Enc (ticks) Tics for data transfer	Kasumi SW Enc(ticks) Tics for processing packets	Actual Kasumi SW Enc (ticks) Processing time – Transfer time
1000	40	320000	219612	4417669	4198057
2000	40	640000	439176	8835131	8395955
3000	40	960000	660397	13255357	12594960
5000	40	1600000	1031367	22212297	21180930
8000	40	2560000	3285773	35841854	32556081

b) KASUMI SW Encryption on 88 bytes packets

In this test case, only difference with previous test case is that SW encryption in accelerated on data length of 88 bytes in order to see which one takes more time encryption or decryption functionalities. Following table shows the result of KASUMI SW based Encryption implementation execution on 88 data lengths.

Table A10. KASUMI SW Encryption on 88 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi HW Enc (ticks) Tics for data transfer	Kasumi HW Enc (ticks) Tics for processing packets	Actual Kasumi SW Enc (ticks) Processing time – Transfer time
1000	88	704000	219655	8605117	8385462
2000	88	1408000	439125	17209759	16770634
3000	88	2112000	660238	25817179	25156941
5000	88	3520000	1009960	43157767	42147807
8000	88	5632000	3260919	69351433	66090514

Following picture show SW based KASUMI encryption implementation execution results on data lengths of 40 bytes and 88 bytes

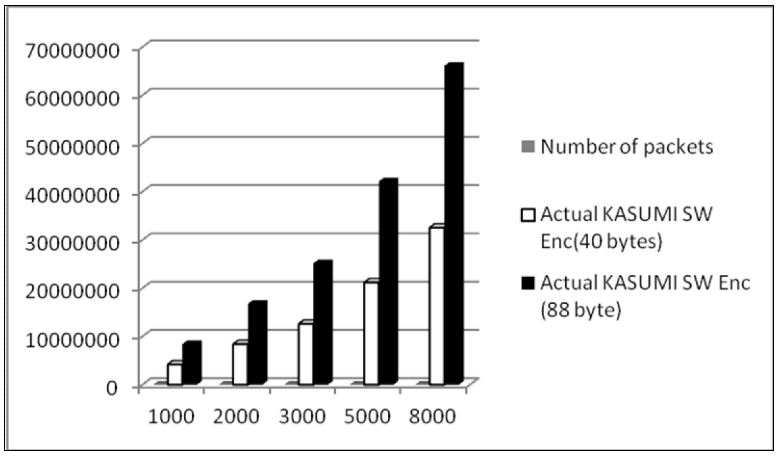


Figure A8. KASUMI SW Encyption on 40 and 88 bytes packets

c) KASUMI SW Encryption on 500 bytes packets

In this test case, only difference with previous cases is that SW encryption in accelerated on data length of 500 bytes in order to see which one takes more time on encryption process. Following table shows the result of the test case.

Table A11. KASUMI SW Encryption on 500 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Enc (ticks) Tics for data transfer	Kasumi SW Enc (ticks) Tics for processing packets	Actual Kasumi SW Enc (ticks) Processing time – Transfer time
1000	500	4000000	220294	44993321	44773027
2000	500	8000000	450726	90010210	89559484
3000	500	12000000	692503	135597913	134905410
5000	500	20000000	2788925	226566862	223777937
8000	500	32000000	5068858	362494597	357425739

d) KASUMI SW Encryption on 1000 bytes packets

In this test case, only difference with previous case is that SW encryption is accelerated on data length of 100 bytes in order to see which one takes more time on encryption process. Following table shows the result of KASUMI SW based Encryption implementation execution on 88 data lengths.

Table A12. KASUMI SW Encryption on 1000 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Enc (ticks) Tics for data transfer	Kasumi SW Enc (ticks) Tics for processing packets	Actual Kasumi SW Enc (ticks) Processing time – Transfer time
1000	1000	8000000	224313	88625113	88400800
2000	1000	16000000	478345	177705422	177227077
3000	1000	24000000	2442241	266775160	264332919
5000	1000	40000000	4612936	444828833	440215897
8000	1000	64000000	6890433	711754587	704864154

i) KASUMI SW decryption on 40 bytes packets

In this test case, only difference with previous cases is that SW decryption is accelerated on data length of 40 bytes in order to see which one takes more time encryption or decryption when doing analysis in later chapters. Following table shows the result of SW decryption acceleration.

Table A13. KASUMI SW decryption on 40 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Dec (ticks) Tics for data transfer	Kasumi SW Dec(ticks) Tics for processing packets	Actual Kasumi SW Dec (ticks) Processing time – Transfer time
1000	40	320000	219612	4417682	4198070
2000	40	640000	439176	8835204	8396028
3000	40	960000	660397	13255359	12594962
5000	40	1600000	1031367	22211959	21180592
8000	40	2560000	3285773	35841963	32556190

f) KASUMI SW decryption on 88 bytes packets

In this test case, only difference with previous case is that SW decryption is accelerated on data length of 88 bytes in order to see which one takes more time encryption or decryption. Following table shows the result of SW decryption processing:

Table A14. KASUMI SW decryption on 88 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Dec (ticks) Tics for data transfer	Kasumi SW Dec (ticks) Tics for processing packets	Actual Kasumi SW Dec (ticks) Processing time – Transfer time
1000	88	704000	219655	8605116	8385461
2000	88	1408000	439125	17209668	16770543
3000	88	2112000	660238	25817224	25156986
5000	88	3520000	1009960	43157470	42147510
8000	88	5632000	3260919	69350621	66089702

i) KASUMI SW decryption on 500 bytes packets

In this test case, only difference with previous case is that SW decryption is accelerated on data length of 500 bytes in order to see which one takes more time encryption or decryption. Following table shows the result of SW decryption processing:

Table A15. KASUMI SW decryption on 500 byte packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Dec (ticks) Tics for data transfer	Kasumi SW Dec (ticks) Tics for processing packets	Actual Kasumi SW Dec (ticks) Processing time – Transfer time
1000	500	4000000	220294	44992749	44772455
2000	500	8000000	450726	90009893	89559167
3000	500	12000000	692503	135597265	134904762
5000	500	20000000	2788925	226567913	223778988
8000	500	32000000	5068858	362496067	357427209

j) KASUMI SW Decryption on 1000 bytes packet

In this test case, only difference with previous case is that KASUMI SW based decryption is accelerated on data length of 1000 bytes in order to see which one takes more time encryption or decryption. Following table shows the result of SW decryption processing:

Table A16. KASUMI SW decryption on 1000 bytes packets

Number of packets	Data length (byte)	Data size (bits)	Kasumi SW Dec (ticks) Tics for data transfer	Kasumi SW Dec (ticks) Tics for processing packets	Actual Kasumi SW Dec (ticks) Processing time – Transfer time
1000	1000	8000000	224313	88625309	88400996
2000	1000	16000000	478345	177707020	177228675
3000	1000	24000000	2442241	266773409	264331168
5000	1000	40000000	4612936	444830796	440217860
8000	1000	64000000	6890433	711755564	704865131

Following figure shows SW based KASUMI decryption processing on data lengths of 500 bytes and 1000 bytes with 3 different data block sizes:

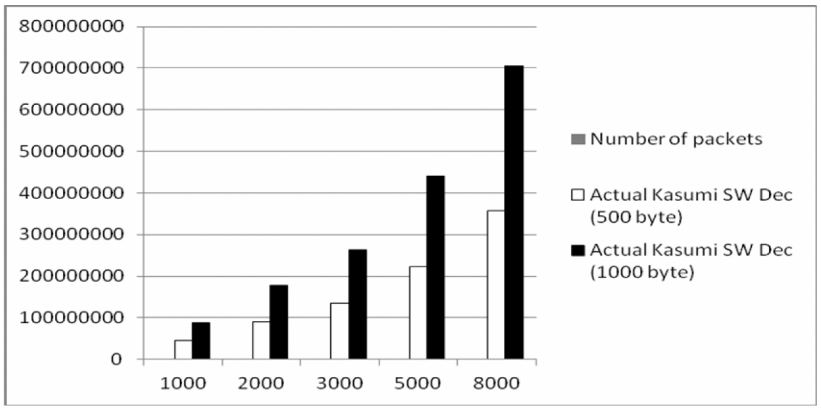


Figure A9. KASUMI SW decryption on 500 and 1000 bytes packets