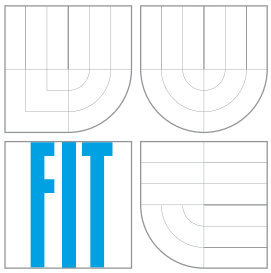
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁVRH A ANALÝZA VÝKONNOSTI PARALELNÍHO ZPRACOVÁNÍ SRTP PŘENOSŮ

DESIGN AND PERFORMANCE ANALYSIS OF PARALLEL PROCESSING OF SRTP PACKETS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

JAN WOZNIAK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETER JURNEČKA

BRNO 2013

Abstrakt

Šifrování multimediálních datových přenosů v reálném čase je jednou z úloh telekomunikační infrastruktury pro dosažení nezbytné úrovně zabezpečení. Rychlost provedení šifrovacího algoritmu může hrát klíčovou roli ve velikosti zpoždění jednotlivých paketů a proto je tento úkol zajimavým z hlediska optimalizačních metod. Tato práce se zaměřuje na možnosti paralelizace zpracování SRTP pro účely telefonní ústředny s využitím OpenCL frameworku a následnou analyzu potenciálního zlepšení.

Abstract

Encryption of real-time multimedia data transferes is one of the tasks for telecommunication infrastructure in order to provide essential level of security. Execution time of ciphering algorithm could play fundamental role in delay of the packets, therefore it provides interesting challenge in terms of optimization methods. This thesis focuses on paralellization possibilities of processing SRTP for the purposes of private branch exchange with the use of OpenCL framework and analysis of potential improvement.

Klíčová slova

AES, obecné výpočty na GPU, OpenCL, paralelní výpočty, SRTP, SIP, telefonní ústředna, brána, VoIP.

Keywords

AES, general-purpose GPU, OpenCL, parallel computations, SRTP, SIP, private branch exchange, gateway, VoIP.

Citace

Jan Wozniak: Design and Performance Analysis of Parallel Processing of SRTP Packets, diplomová práce, Brno, FIT VUT v Brně, 2013

Design and Performance Analysis of Parallel Processing of SRTP Packets

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Petera Jurnečky.

.....
Jan Wozniak
May 12, 2013

Contents

List of Figures

Chapter 1

Introduction

One of the essential metrics for measuring VoIP gateway's performance is the number and quality of simultaneous calls. It is affected mostly by the computational demands of used communication protocols and number of registered users. While the count of registered users provides very limited room for improvement by the nature of the problem itself, there could be wide variety of approaches in implementing the protocol stacks.

The communication protocols for VoIP gateway can be divided into two groups. Signalization, which consists mostly of textually represented protocols, where the messages' occurrence is either periodical with quite small frequency, or based on the users initiative which is a stochastical event depending on the activity of the user. However, generally the recurrence of both is rather similar. Comparably more resources during indirect simultaneous call sessions consumes processing the second group of protocols, transport of multimedia packets. Since security has recently grown to be necessary feature in VoIP communication and the encryption and decryption processes are designed with the idea of optimization, it is primary scope of interest of this thesis.

Development and results in the areas of parallel architectures shows that many procedures could be distinctively accelerated by executing the algorithm on the processing unit capable of parallel computations. Therefore, target of this thesis is implementation and analysis of parallel processing of encrypted real-time multimedia data transfer.

Chapter 2 describes the structures and algorithms used in Secure Real-time Transport Protocol. Increased attention is devoted to explanation of Advanced Encryption Standard, which is default cipher used in SRTP, including brief theoretical background and analysis of SRTP and AES. Because SRTP doesn't provide key exchange mechanism for symmetric AES cipher, the chapter also includes description of selected protocol extensions for this task.

Chapter 3 provides basic information about graphic processing unit and the usage of GPU for general purpose computations. Part of the chapter is principal explanation of OpenCL framework and its elementary usage for the developer. As the parallel processing is diverse and wide study, the area of parallel paradigm that could be associated to the further implementation of this thesis is mentioned with particular interest and focus.

Chapter 4 defines the term SIP gateway for the context of this thesis, discusses the design of such gateway and includes listing of selected further implemented protocol stacks, their mutual interaction and possible improvement of processing the passing data. The highest amount of attention is devoted to the comparison of different approaches to design of SRTP stack and identification of main characteristics of native OpenCL programming pattern in contrast to persistent thread model. The advantages of both parallel implementations over

serial code executed on the same hardware is mentioned as well. Short introduction and description of used design patterns is included in order to provide better comprehensibility of the application schemes.

Chapter 5 covers the reference implementation of the previous theoretical part of this thesis, used techniques and algorithms and reasoning behind their selection. Even though the focus of the thesis is primarily research of available contemporary methods there were many restrictions. The requirements of this chapter arise from currently used implementation and hardware limitation of the gateway.

Finally chapters 6 and 7 summarize the potential benefits of usage the GPGPU for the number of maximal simultaneous calls and shows visualization of achieved results in improvement and decrease of latency. Also these chapter discusses possible contribution to related topics, such as transcoding of media compressing codecs which parallel implementation may provide even higher level of improvement.

Chapter 2

Secure Real-time Transport Protocol

To achieve confidentiality and necessary security for real-time multimedia transmission over TCP/IP connection there has been invented SRTP[16]. Except previously mentioned, it provides message authentication and replay protection for both RTP and RTCP traffic, however, the thesis is going to focus on the implementation and computation time of the security. The default cipher is AES in counter mode.

2.1 Packet Structure

SRTP packet can be described as RTP extension. It keeps the RTP fields of the packet such as:

- Version (V) – two bit number which currently is equal to 2.
- Padding (P) – boolean value whether the padding is set.
- Extension (X) – if this field is set, fixed header must be followed by exactly one extension header.
- CSRC count (CC) – number of CSRC identifiers that follow the fixed header.
- Marker (M) – interpretation defined by a profile.
- Payload Type (PT) – identifies the type of payload
- Sequence Number (SEQ) – increments by one for each RTP packet.
- Timestamp (TS) – reflecting the exact moment the payload was sampled.
- Synchronization Source Identifier (SSRC) – identifier of RTP synchronization source within the single RTP session.
- Contributing Source Identifiers (CSRC) – list of 0 to 15 items identifying contributing sources.

The SRTP protocol defines that only payload is encrypted and also describes new fields in the RTP header.

- Master Key Identifier (MKI) – unique identifier of the master key (previously signaled) to be used in session key derivation.
- Authentication Tag – carries message authentication data. If both encryption and authentication are used, encryption should be applied first.

The packet length is variable and depends on number of CSRC used and length of payload. The following scheme describes the packet with proportional sizes of each field.

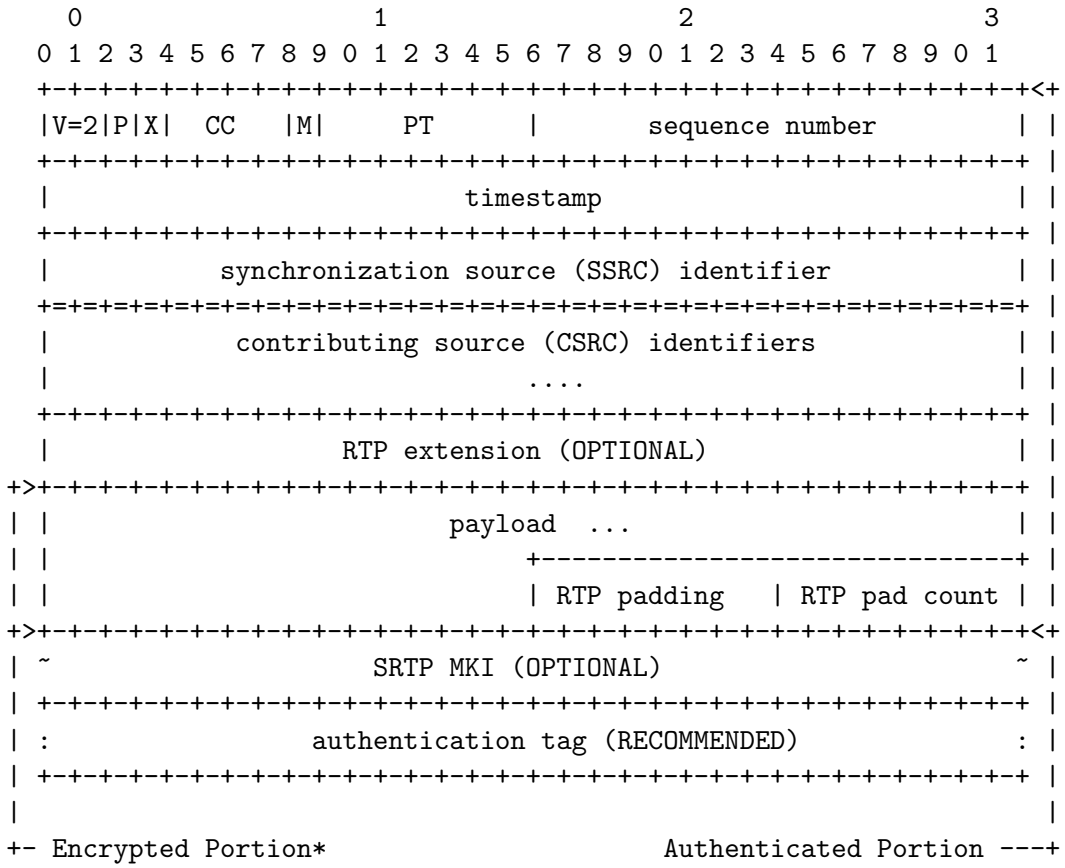


Figure 2.1: SRTP packet structure.

2.2 Cryptographic Context

In order to implement SRTP stack in the application, it is necessary to preserve certain information about each encrypted session, which is called *cryptographic context*. It must consist of the following:

- Rollover Counter – 32-bit unsigned number, records how many times has the RTP sequential number been reseted to zero passed the value 65 535.
- Highest Received SEQ – 16-bit unsigned number
- Identifier of the Encryption Algorithm – the cipher and its mode

- Replay List – containing indices of recently received and authenticated SRTP packets
- MKI – if the MKI is present in current session, the length of the MKI field in octets, actual value of currently used MKI
- Master Keys – enumeration of random and secret master keys and counter for each key of how many packets have been sent with that key. Single Master Key identifies SRTP stream and corresponding SRTCP stream.
- Session Keys – current key for encryption and authentication including stored their lengths in `n_e` and `n_a`

And for every master key, the cryptographic context may contain also random but possibly public *Master Salt* which will be used in key derivation.

2.3 Master Key Exchange

There are three most common protocols for key exchange in SRTP session between the end users – SDES, MIKEY, and ZRTP. They differ in what protocol in VoIP communication they extend, provided security guarantees and possible communication overhead.

ZRTP is a protocol extension of RTP for secure establishing session key using Diffie-Hellman key exchange improved for detection of man-in-the-middle attack, which is briefly described in section 2.4, [34]. Another advantage of the improvement is that it doesn't require any prior shared secret nor public key infrastructure.

SDES is protocol extension of SDP[24, 15] typically in SIP[32] message. It is responsibility of the SIP stack to protect the key as secured secret, which is possible via TLS connection for instance.

MIKEY defines the key exchange as part of SDP payload in SIP message. The algorithm is basic Diffie-Hellman which requires either prior shared secret or PKI[1]. The SIP stack doesn't have to protect the transferred information any further.

2.4 Protocol Summary

Main concerns about the use of SRTP are whether the increase of computational complexity and packet size don't make RTP hardly usable and what degree of security does it provide.

Computational Overhead

In VoIP communication the time has essential impact on the quality of transmitted information, therefore it is important that ensuring the security of RTP wouldn't increase the latency over the acceptable level. Among common limitations of real-time communications belong[31]:

- Maximal tolerable latency round-trip time 300ms.
- Smaller packet loss than 5%.
- Sensitivity to factors that are difficult to objectively measure such as jitter.

[1] Public Key Infrastructure for digital certificates

It has been proven that increase in size of the packet SRTP is insignificant compared to the RTP[13, 14]. Average throughput of secured VoIP is usually around 2% more than unsecured VoIP.

Security

VoIP suffers from many similar security threats as other standard internet services.

Man in the middle in computer security is form of active eavesdropping. The attacker creates connections to both endpoints of the session which allows him to monitor, record or modify the packets in communication making the endpoints believe that the conversation is secured. Protection against such attack could be achieved by key negotiating protocol *ZRTP* which is able to detect this activity[34].

Denial of service is considered an attempt to make target machine unavailable to to its intended users. Typical method of this attack is to saturate the target machine with excessive requests that could lead to overloading the machine. Replay protection mechanism of SRTP with replay lists and authentication headers provide sufficient protection against DoS attack[16, 9].

2.5 AES

This section treats necessary theoretical background of Advanced Encryption Standard, which is the default cipher, and as the text has been written the only cipher, of Secure Real-time Transport Protocol used in VoIP communication.

Advanced Encryption Standard is symmetric block cipher which means it uses the same key for both encryption and decryption and encodes the input in uniform sized blocks. The algorithm was developed to supersede *Data Encryption Standard* due to various security reasons[2] in electronic data transmission.

For this purpose National Institute of Standards and Technology (NIST) announced public competition for new encryption standard in 1997 and considering multiple requirements the *Rijndael*[3] was selected as the most suitable algorithm for the task[12].

2.5.1 Mathematical Preliminaries

All the bytes in AES are interpreted as 8-bit values in finite field 2^8. For better readability the values are printed using hexadecimal notation. Following mathematical therms and operations are used in AES algorithm:

Galois field

In algebra Galois field is finite field with finite number of elements. Common notation is $GF(p^k)$ where p is prime number and k is positive natural number. Therefore it is possible to classify the Galois fields by their size, because only single $GF(p^k)$ exists for each p and k. Characteristics of the field is equal to the p.

[2] For instance COPACOBANA is FPGA based machine that could find an exhaustive key for DES in no longer than a week[25].

[3] Rijndael was original name of the AES as abbreviation of authors' names – Joan Daemen and Vincent Rijmen.

Each byte is in fact a polynomial with degree equal to 7 with coefficients b_i 0 or 1 and this notation $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$. The decimal number 95 could be represented as:

- $5F$ in hexadecimal
- 0101 1111 in binary as a byte
- $x^6 + x^4 + x^3 + x^2 + x^1 + 1$ as polynomial with degree equal to 7

Addition

Addition is defined as addition of coefficinets of both polynomials modulo 2. This operation has the same result as bitwise XOR and because each value is its own inversion, addition and substraction are equal operations.

Multiplication

Multiplication is defined as multiplication of both polynomials modulo irreducible polynomial of degree eight. For AES the irreducible polynomial is defined as

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.1}$$

Multiplication by x

Multiplication of binary polynomial by polynomial x results in polynomial of higher degree therefore the result must be reduced modulo $m(x)$. Following equation is the binary polynomial multiplied by polynomial x.

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \tag{2.2}$$

If $b_7 = 1$ the result must be XORed with the polynomial $m(x)$. This operation can be accomplished as bitwise left shift and XOR with $1B$.

2.5.2 Algorithm Description

The AES is block cipher, therefore both encryption and decryption processes are performed on a matrix of 4x4 bytes called *state*. Even though state has fixed block size 128-bit, supported key sizes are 128-bit, 196-bit and 256-bit.

Encryption process as described in pseudocode 1 has 4 operations performed on each state of the data in specific number of cycles which varies from key length.

- 10 cycles for 128-bit key
- 12 cycles for 196-bit key
- 14 cycles for 256-bit key

Algorithm 1 AES encryption

Cipher(State, Key)
state $\leftarrow$ *AddRoundKey*(State, Key[0])

for $i \leftarrow (1..n-1)$ **do**
    state $\leftarrow$ *SubBytes*(state)
    state $\leftarrow$ *ShiftRows*(state)
    state $\leftarrow$ *MixColumns*(state)
    state $\leftarrow$ *AddRoundKey*(state, Key[i])
end for

state $\leftarrow$ *SubBytes*(state)
state $\leftarrow$ *ShiftRows*(state)
state $\leftarrow$ *AddRoundKey*(state, Key[n])

return state

Key Expansion

Round keys are derived from cipher key through process called *key expansion*. For the ciphering and deciphering purposes, the round keys could be thought as array of 4x4 8-bit values, which length is 10, 12 or 14 according to the used key size. The first matrix is copy of first 128 bits of cipher key. The following round keys are always calculated from the previous key and *rcon* array as explained in the algorithm 2.

Algorithm 2 Key Expansion

ExpandRoundKey(Key, size)
rk[0] $\leftarrow$ Key[0]

for $i \leftarrow (1..size)$ **do**
    k.col(0) $\leftarrow$ Key[$i-1$].col(3).rotate(1).map(sbox $\oplus$ Key[$i-1$].col(0)) $\oplus$ rcon
    for $j \leftarrow (1..3)$ **do**
        k.col(j) $\leftarrow$ Key[i-1].col(j) $\oplus$ k.col($j-1$)
    end for
    rk[i] $\leftarrow$ k
end for

return rk

Ciphering Process

`AddRoundKey` is XOR operation on the state with specific round key. Round key is extracted from the cipher key in *ExpandRoundKey*. Since this operation uses XOR, it is its own inverse form as well.

s_{00}	s_{01}	s_{02}	s_{03}	$\oplus$	k_{00}	k_{01}	k_{02}	k_{03}	$=$	a_{00}	a_{01}	a_{02}	a_{03}
s_{10}	s_{11}	s_{12}	s_{13}		k_{11}	k_{12}	k_{13}	k_{10}		a_{11}	a_{12}	a_{13}	a_{10}
s_{20}	s_{21}	s_{22}	s_{23}		k_{22}	k_{23}	k_{20}	k_{21}		a_{22}	a_{23}	a_{20}	a_{21}
s_{30}	s_{31}	s_{32}	s_{33}		k_{33}	k_{30}	k_{31}	k_{32}		a_{33}	a_{30}	a_{31}	a_{32}

Table 2.1: AddRoundKey on state s with key k where $a_{ij} = s_{ij} \oplus k_{ij}$.

`ShiftRows` is performed on each row of the state matrix. The first row is not shifted, second row is shifted by one byte to the left, third row is shifted by two bytes to the left and fourth row is shifted by three bytes to the left. Inverted ShiftRows for decryption is simply reversion.

a_{00}	a_{01}	a_{02}	a_{03}		a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}	$\longrightarrow$	a_{11}	a_{12}	a_{13}	a_{10}
a_{20}	a_{21}	a_{22}	a_{23}		a_{22}	a_{23}	a_{20}	a_{21}
a_{30}	a_{31}	a_{32}	a_{33}		a_{33}	a_{30}	a_{31}	a_{32}

Table 2.2: State on the right is the first state after ShiftRows is performed.

`MixColumns` together with ShiftRows provides diffusion in the AES algorithm. During this operation each column of the state is multiplied in Galois field 2^8 by matrix 2.3.

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \tag{2.3}$$

As a result of this multiplication, each column $[s_{0c}, s_{1c}, s_{2c}, s_{3c}]$ is replaced by the column $[a_{0c}, a_{1c}, a_{2c}, a_{3c}]$ which could be calculated:

$$\begin{aligned} a_{0c} &= 2 \cdot s_{0c} \oplus 3 \cdot s_{3c} \oplus s_{2c} \oplus s_{1c} \\ a_{1c} &= s_{1c} \oplus 2 \cdot s_{0c} \oplus 3 \cdot s_{3c} \oplus s_{2c} \\ a_{2c} &= s_{2c} \oplus s_{1c} \oplus 2 \cdot s_{0c} \oplus 3 \cdot s_{3c} \\ a_{3c} &= 3 \cdot s_{3c} \oplus 2 \cdot s_{2c} \oplus s_{1c} \oplus s_{0c} \end{aligned} \tag{2.4}$$

`SubBytes` is non-linear transformation of the input *state*. Each byte in the state matrix is replaced with byte from substitution array of 256 8-bit values called S-box. The S-box A for encryption is generated by determining the multiplicative inverse for a given number in $GF(2^8)$ Rijndael's finite field and then affine transformation. The S-box A for decryption uses the same matrix but has first applied addine transformation and then the multiplicative inverse. For implementation purposes both S-boxes are precomputed.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{2.5}$$

In this transformation $[x_0, .., x_7]$ is the multiplicative inverse as vector, and $\oplus$ is XOR operation.

2.5.3 Block Cipher Modes

During encryption the same key is applied repeatedly on the uniform length blocks of data to whose the message is separated into. Large amount of ciphered data with the same encryption key might present security threat unless the ciphering algorithm provides form of randomization the output value. Such procedure might be achieved by additional input value.

There are many variations on block cipher to provide this confidentiality[20], for AES algorithm the most often used are *counter mode* and *f8-mode*. Both algorithms keep standard high level of confusion of the AES algorithm and provides necessary diffusion[4].

Both algorithms share some similar terminology and acronyms:

- IV – initial value used for encrypting the first block
- C_i – ciphertext block number i
- P_i – plaintext block number i
- E_K – encryption function
- D_K – decryption function

Counter Mode

The counter mode (CTR) turns AES block algorithm into stream cipher with possibility for parallel computations[33]. It is possible to decrypt the cipher text even with loss of number of blocks because the encrypted blocks are not dependent on the previous blocks. Instead the additional diffusion value are achieved by specific counter.

Equation 2.6 describes computation of counter value, equation 2.7 describes ciphering the counter value, equation 2.8 is encryption process – XOR operation of plaintext with encrypted counter value which produces ciphered text and equation 2.9 is decryption process.

$$CTR_i = (IV + i - 1) \mod 2^B \tag{2.6}$$
$$H_i = E_K(CTR_i, key) \tag{2.7}$$
$$C_i = P_i \oplus H_i \tag{2.8}$$
$$P_i = C_i \oplus H_i \tag{2.9}$$

[4] *Confusion* and *diffusion* are basic two properties of secure cipher introduced by Claude Shannon[17].

The last block of the plaintext doesn't have to be padded[5], it is common to use only the most significant bits of ciphered counter to be XORed with plaintext in cipher algorithm (and in similar way for deciphering).

F8-mode

The f8-mode is a variant of commonly known Output Feedback Mode (OFB) [20] with more elaborate initialization and feedback function [16]. The first output block O_1 is computed from IV, then it is XORed with plaintext to produce the first ciphertext block. The output block from previous step O_{j-1} is used to compute the current output block O_j which is always XORed with current plaintext in encryption algorithm.

The equation 2.10 describes the improved initializing function where m is the mask. The equations 2.11 and 2.12 describes computation of value, which is used for ciphering algorithm to produce output values in equation 2.13. Equation 2.14 describes ciphering and equation 2.15 describes deciphering.

$$IV' = E_K(IV, key \oplus m) \tag{2.10}$$
$$I_1 = IV' \tag{2.11}$$
$$I_j = O_{j-1} \oplus IV' \oplus j \tag{2.12}$$
$$O_j = E_K(I_j, key) \tag{2.13}$$
$$C_j = P_j \oplus O_j \tag{2.14}$$
$$P_j = C_j \oplus O_j \tag{2.15}$$

[5] Padding can be used for the plaintext that is not aligned to the multiplies of the block.

Chapter 3

General Purpose GPU

This chapter describes the basic ideas and techniques behind GPU parallel programming model and architecture. Following text will focus on possibilities of effective implementation for GPGPU and integrated GPU in modern CPU using OpenCL framework, brief description of selected principles and development of parallel applications.

Parallel machines have impressive performance to cost ratio compared to the common sequential machines[19], but bring well known problems for software development such as run-time resource allocation and resource sharing. Mapping parallel program to multiprocessor machine is complex problem that needs to decide about task allocation, scheduling of processes, communication patterns and much more.

While current CPUs are powerful and sophisticated chips, their design must be focused on wide variety of tasks, therefore vast majority of resources might not be as fully utilized as could have been. The GPU chips provide much better theoretical performance for certain tasks for smaller price[30]. Interest among developers has grown in using the power GPUs provide for other tasks than graphics pipeline.

In order to achieve improvement in certain algorithm it is necessary to analyze the procedures and find possibilities for parallelization and take under consideration that usage of additional processing unit brings computational overhead. The characteristics of such application are[29]:

- Utilization of data-parallelism – many non-graphical problems might be separated into fractional procedures and computed separately, such as matrix calculations individially for each cell.
- Large portion of computation – GPU processors are optimized for computations over handling conditional evaluations.
- Throughput over Latency – computations on GPU are designed for large overall throughput of entire data rather than short response time of each individual operation.

The current trend in development shows that parallel computations either in the form of GPU computations and APU[1] are worth examination and research. SIMD[2] has already proven it's value on improving performance with parallelization of various algorithms[6, 2].

[1] Accelerated Processing Unit – in this context it means CPU with GPGPU chip.

[2] Single Instruction Multiple Data – multiple processing elements that perform the same operation on multiple data points simultaneously[21].

APU

Usual solutions with graphics card can have high power consumption. The modern trend and need of transportable forced development to reduce negative effects of GPUs while keeping as much of latest visual experience as possible[18]. Both solutions utilize a portion of computer's system RAM memory.

APU is Accelerated Processing Unit that is designed to accelerate certain type of computations outside of CPU in single chip. It could include GPU, FPGA or similar specialized processing unit. Among the best known there are Intel HD Graphics[4], AMD Fusion[1] and NVIDIA Project Denver[7].

3.1 OpenCL

Development for parallel computation brought need for infrastructure. OpenCL is an industry standard framework for programming heterogenous systems composed of a combination of CPUs, GPUs, DSP and other processing units[26]. With OpenCL it is possible to write a software that will run on wide variety of platforms from cell phones or computers to massive supercomputers.

The *OpenCL programming language* has syntax based on the language C with few additions and limitations arising from the design and architecture of heterogenous platforms. Among most important limitations it omits the use of recursion, function pointers and header files. On the other hand, the language is extended to the use of parallelism with build in types and synchronization. Also it defines many functions and four new keywords as memory region qualifiers: `__global`, `__local`, `__constant` and `__private`.

For further reading of the text and better comprehensibility, there are listed necessary words from OpenCL terminology[26].

- Context – contains one or more devices used for kernel execution and are used for managing command queues, memory and program.
- Kernel – function written in OpenCL programming language that is executed on OpenCL device.
- Work-item – instance of executing kernel.
- Work-group – organisation of work-items.
- Command-queue – interaction between the host and OpenCL device through commands posted by the host and provides synchronization methods for the execution of the commands.

OpenCL platform includes single *host* that communicates with the user and the OpenCL program. The host is connected to one or more OpenCL *devices* where the *kernels* are executed. *Kernel* could be considered as the entry point between host and GPU. In order to achieve parallelism, the device consists of many *work-items* whose execute multiple instances of kernel at the same time. The work-items are organized in integer indexed orthogonal grid where the unique index of a work-item is called global ID. The identification of work-item is possible through combination of its local ID inside a specific *work-group* and the work-group global ID.

3.1.1 Platform Model

The OpenCL provides a high-level abstraction model representing any heterogenous platform. The *host* is a bridge between parallel computations on one or more devices and interaction with external environment. *Device* could be CPU, GPU, DSP or any other processing unit supporting OpenCL and consists of compute units which are further divided into processing elements. *Processing element* is abstraction of a work-item and *compute unit* is in similar way representation of work-group.

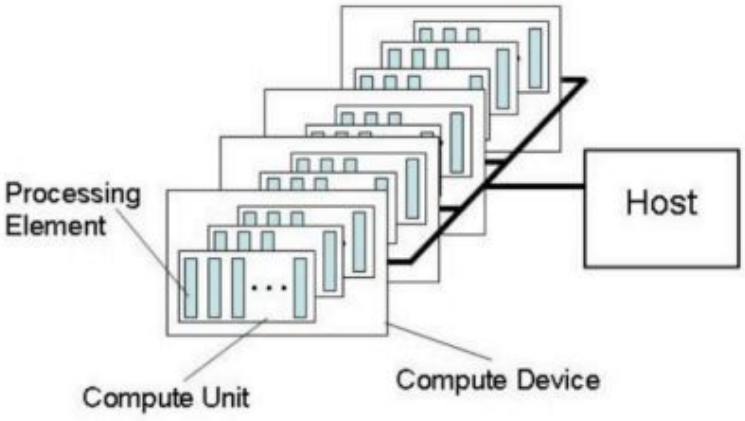


Figure 3.1: OpenCL platform model with one host and multiple devices[26].

3.1.2 Execution Model

The OpenCL software executes on two levels

- **Host code** – the OpenCL doesn't define any restrictions about the host part of the application, it defines only the interaction between host and devices. It consists of selection and initialization of the context – selected platform and devices.
- **Device code** – written in OpenCL programming language in the form of short functions, kernels, that usually transform an input array through series of processes into output array. It is compiled via OpenCL compiler and executed on the device's work-items.

The host program takes care of synchronization and plans the execution of each kernel on the devices. Each instance of kernel runs in separate work-item and the work-items within each work-group execute concurrently.

3.1.3 Memory Model

OpenCL defines two types of memory objects. The *buffer object* is versatile type that could be used for representation of any data type available in C or OpenCL language. The *image object* is restricted to containing pictures only and is optimized for the specific needs of image processing.

OpenCL uses a hierarchically structured memory. The types differ in access time, availability and types of usage[26]:

- **Private Memory** – each work-item has it's own private memory which could be thought of as analogy to CPU's registers. It is the fastest type of memory used in OpenCL.
- **Local Memory** – designed for sharing data between work-items who belong to the same work group. It is used to reduce the number of accesses to the global memory. Local memory is slower than private memory but faster than global memory. The programmer is denied both direct access and control over local memory. The analogy could be the cache in CPU.
- **Global Memory** – shared among all work-items in the same context.
- **Host Memory** – memory visible only for the host, OpenCL only defines how the host interacts with OpenCL objects and constructs.

There could be another type of memory in graphic cards that OpenCL doesn't define

- **PCI Memory** – type of memory that could be used by the program and GPU, part of host memory. It is slower than global memory.

Chapter 4

Design

The aim of the implementation is to determine whether the parallel processing of SRTP could increase the limitations on modern VoIP softgates. The development of sophisticated softgate requires elaborate engineering and implementation of various communication protocols that would overshadow the effort in parallel processing. Therefore only narrow selection of well know communication protocols has been implemented. For VoIP telephony, registration and maintenance of users serves *SIP* protocol, for the media transmission description and session description *SDP* protocol, and for secure media transport *SRTP* with *ZRTP*. There is also implementation of *LCP* stack[1].

4.1 Design Patterns

More complex the application is the higher level of considerate design it requires. There are plenty of already well tested design patterns from which the implementation could be based on and as the field of VoIP communication has been known for decent amount of time, there are currently couple of advised design patterns, from which the particular implementation for this thesis stands on four – mediator pattern, singleton pattern, factory method pattern [22] and protocol stack pattern [8]. None of these design patterns could be thought as contribution of the thesis as they all belong to common public knowledge and their examination was not the main topic of the research. However, their explanation is provided in order to make the rest of the chapter more comprehensible.

4.1.1 Mediator Pattern

In object oriented design the common problem be the large number of classes and their mutual interaction. One of the possible solutions for the latter can be behavioral pattern called mediator, which is named after the way it alters the running behavior. The pattern consists of following participants:

- **Mediator** – defines an interface for communicating with colleague objects
- **ConcreteMediator** – implements cooperative behavior by coordinating colleague objects, it knows and maintains its colleagues

[1] Light-weight Control Protocol – communication protocol for Siemens prototype VoIP phone.

- **ConcreteColleague** – each colleague knows its mediator object and it communicates with its mediator whenever it would have otherwise communicated with another colleague

The mediator object communicates with multiple colleague object through defined interface, the interaction between colleague objects is strictly limited. One of the issues of such design is that the data flow might be bottlenecked by the only option of mutual communication is realised via single object if there is a need for critical section and their exclusion.

4.1.2 Singleton Pattern

During creation of the application architecture certain class may be required to provide global point of access to it while preserving only one instance. One approach could be to have global variable but that is not complete fulfilment of the requests, because multiple instances could still be created. Singleton design pattern offers a solution when the class itself is responsible for the number of instances which ensures that nowhere in the code multiple object of such class may be created. This doesn't affect the rest of the design, only one single class, therefore, it has only one participant:

- **Singleton** – there must be exactly one instance of the class and class must prevent from instantiation of multiple instances, it must be globally available from well known access point

4.1.3 Protocol Stack Pattern

There are two design patterns closely related to the protocol stack design pattern which it uses as higher level of abstraction in explaining the correspondign relations in design.

- **Protocol Layer** – provide a common interface for implementing different layers of communication protocol stack.
- **Protocol Packet** – unification and simplification of internal packet buffers and their access.

This pattern's usage is concentrated but not limited to dynamic exchange of protocol layers from the stack, their insertion and removal, thanks to separate view and decoupling of each implemented protocol and its layers.

The participants of protocol stack pattern are:

- **Protocol Stack** – contains and maintains list of used protocols.
- **Protocol Layer** – provides interface and communication point for each individual layer. The certain layers are abstracted from the actual type of the upper layer and lower layer classes.

4.2 SIP Gateway

In order to create a session for VoIP communication between two endpoints, there must be device that will be able to create such connection and negotiate protocols for their

interaction and data exchange. In telecommunications such device is called *gateway*. The essential function of gateway is protocol translation to interconnect networks and devices using different protocol technologies. The SIP gateway used in this thesis provides protocol conversion between subset of SIP protocol and full implementation of LCP protocol.

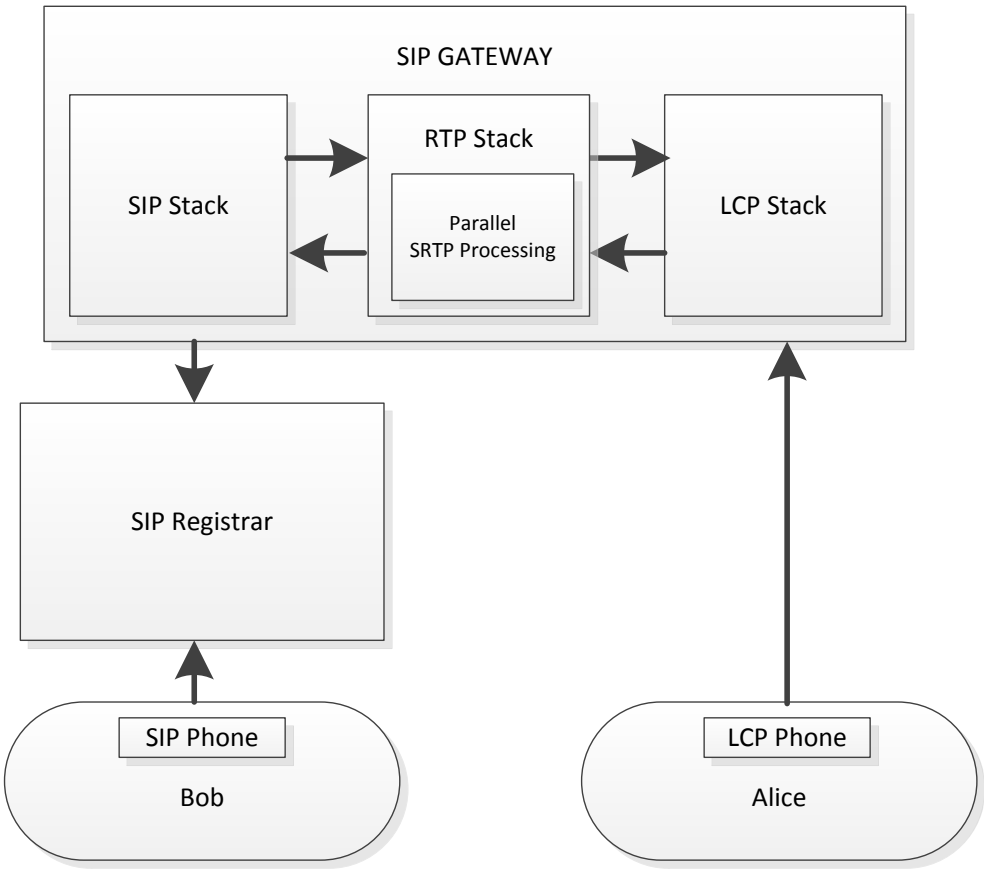


Figure 4.1: SIP gateway with two VoIP telephones accessible, LCP phone directly connected and SIP phone through SIP Registrar.

Multiple SIP or LCP telephones are connected to the SIP Gateway whose appear as users to the SIP registrar[2]. The SIP gateway in this scenario works as bridging point between the SIP telephones and SIP registrar, LCP phones and SIP registrar or LCP phones directly.

The modules of SIP gateway are implemented in different programming languages and each serve specified purpose.

- **Gateway core** – implemented in Java, provides communication between each module and encapsulates basic functionality of a Gateway.
- **SIP Stack** – Java API for SIP stack called JAIN SIP[27], implemented basic SIP features.
- **LCP Stack** – implemented in Java fully covering LCP protocol v1.0.

[2] Used registrars were Asterisk and Siemens HiPath 4000

- **RTP Stack** – for non-direct connections where the telephones couldn't agree on communication channel for the session, RTP stack implemented in C/C++ and OpenCL provides necessary bridging point.

Measurement of utilization of computational resources during execution of ciphering algorithm does provide correct and exact results, however in real deployment the effectivity could be negatively affected by the other processes running on the softgate. SIP Gateway is a collection of programs and utilities whose together implement a server for lightweight LCP phones and supplement a SIP functionality for each phone to be able to connect to an actual SIP registrar.

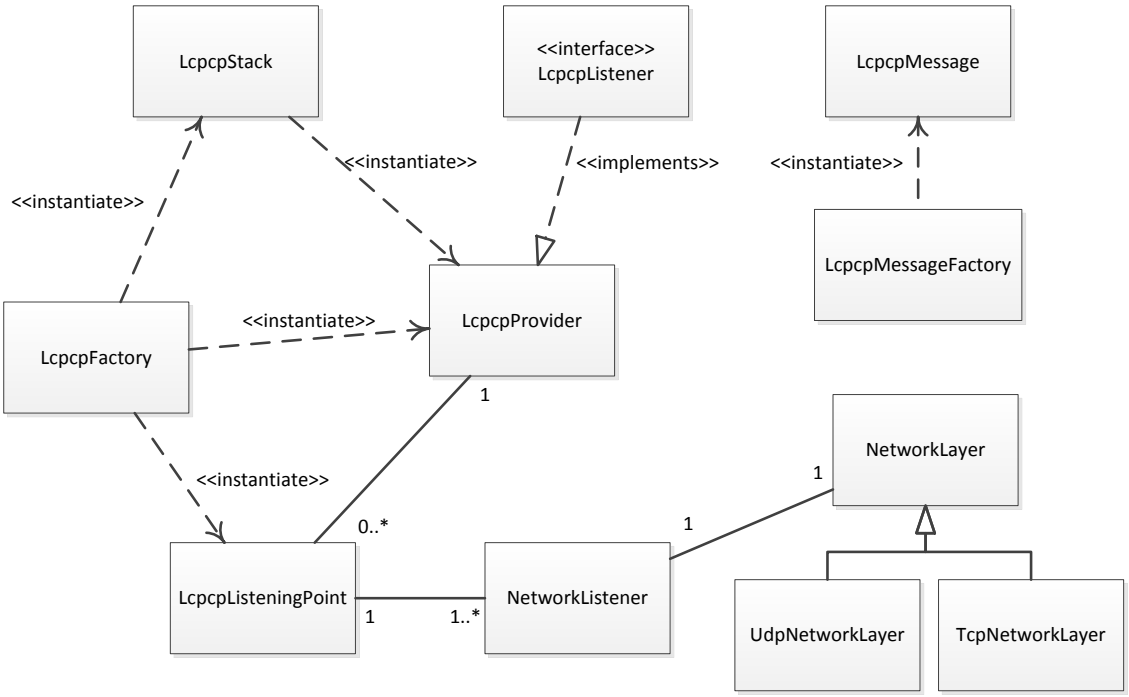


Figure 4.2: Architecture of LCP stack, design was inspired by the JAIN-SIP api[27].

The core application should offer simple management via command line for both development and tracing of the flowing communication and basic functionality for communication and session management.

The core class of the gateway is `Daemon`, which controls the flow of data inside the application and provides interfaces to communicate with external applications. During the composition of gateway the mediator design pattern was used where the daemon is mediator and all directly communicating classes are colleagues.

`SIP Stack` provides the interface to communicate with SIP Registrar. The single SIP Stack is shared for all users, implicitly runs on well known port for SIP communication 5060, which could be explicitly changed if necessary.

`LCP Stack` visualized on the figure 4.2 was designed to reflect the elaborate design used in JAIN SIP[27]. While SIP is much richer protocol than LCP, the design of the stack was extremely shortened but the basic structure of elementary components and their interaction remained the same. LCP stack runs implicitly on the recommended port 4066, but as well as SIP stack port, the port could be variable if needed. Each SIP/LCP client is instance of

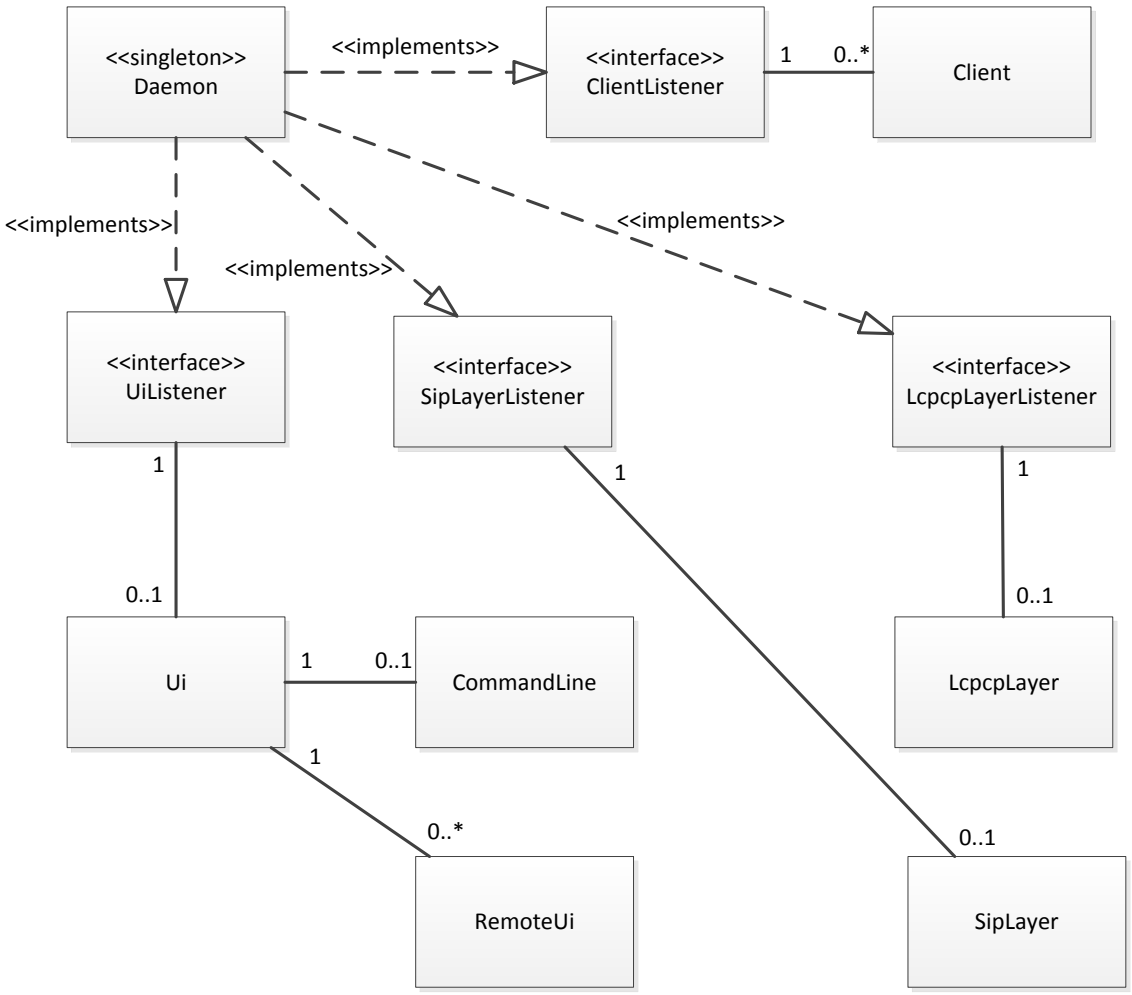


Figure 4.3: Architecture of SIP Gateway with singleton design pattern. Consists of *Daemon* class, multiple LCP phones connected via LCP stack and represented as instances of *Client* class and multiple user interfaces for control over the gateway.

`Client` class, and universal interface for remote communication and administration shall be provided as well.

`RTP Stack` is devoted increased amount of attention in design because it covers the focus of this thesis. All of the stacks are interchangable and during their design were used recommendations from protocol stack design pattern and its related patterns.

4.3 SRTP Stack

The essential point of implementation improvement lies in design of SRTP stack as it has been mentioned in previous text that it consumes majority of resources of the gateway during indirect media sessions. Proper implementation must not lack following properties:

- **encryption module** – implementation of AES-128b cipher as defined in RFC-3711 [16] in at least CRT mode that provides protection of transferred data with different keys for each endpoint in all concurrent sessions.

- **input and output buffers** – in order to avoid exhaustive allocation and deallocation of structures for input and output packets, the data storage should be implemented as thread safe pool of buffers with sufficient size and both, synchronization techniques and memory override protection.
- **transcoding module** – due to various reasons, endpoints may not be able of negotiate the same media compressing codec. The SRTP stack should allow the transcoding and then encapsulate the process without unneccessary additional demands for the gateway.
- **integration interface** – most of the procedures implemented in SRTP stack should be encapsulated to minimize overloading data transferres with the gateway providing only essential and minimal interface with callback features to simplify and unify the integration process.

An advanced techniques like **jitter buffer** may improve overall quality of VoIP communication, however, each end device capable of such communication must implement these techniques as well, therefore, it may render itself redundant and generating minimal, but still additional latency.

4.3.1 SRTP Processing

Advantage of usage AES in CM is that it allows out-of-order processing. Because majority of RTP implementations are build on UDP transport layer, which is simple model with minimal protocol mechanisms, neither order nor delivery of the packets are guaranteed in exchange for smaller average delay and smaller traffic.

The exact size of payload in SRTP packet can differ widely according to the used codec, its bit rate, and sampling frequency. The selection of used voice codecs, their sampling periods and payload size are mentioned in table 4.1.

Codec and Bit Rate	Payload Size	Sampling Period	Packets Per Second
G.711 – 64 Kbps	160 bytes	20 ms	50
G.729 – 8 Kbps	20 bytes	20 ms	50
G.726 – 32 Kbps	80 bytes	20 ms	50
G.726 – 24 Kbps	60 bytes	20 ms	50
G.728 – 16 Kbps	60 bytes	30 ms	33

Table 4.1: Selected codecs and payload information [11].

Fixed block size of AES is 16 bytes, which means that one or more states could be mapped to the packet using any of the mentioned common codecs. Parallelization of the encryption process could be performed either on a single state, where value during every method of the AES of each cell of the state is computed separately, therefore a work-item can be mapped on computing for each cell. Theoretical common hardware should be capable of utilizing 16 work-items in a single work-group which is the maximal number of needed by this design.

Another possible approach for codecs with larger payload size, such as G.711, could be to map multiple states for the parallel execution of entire packet, which for the particular codec would require significantly more computational units.

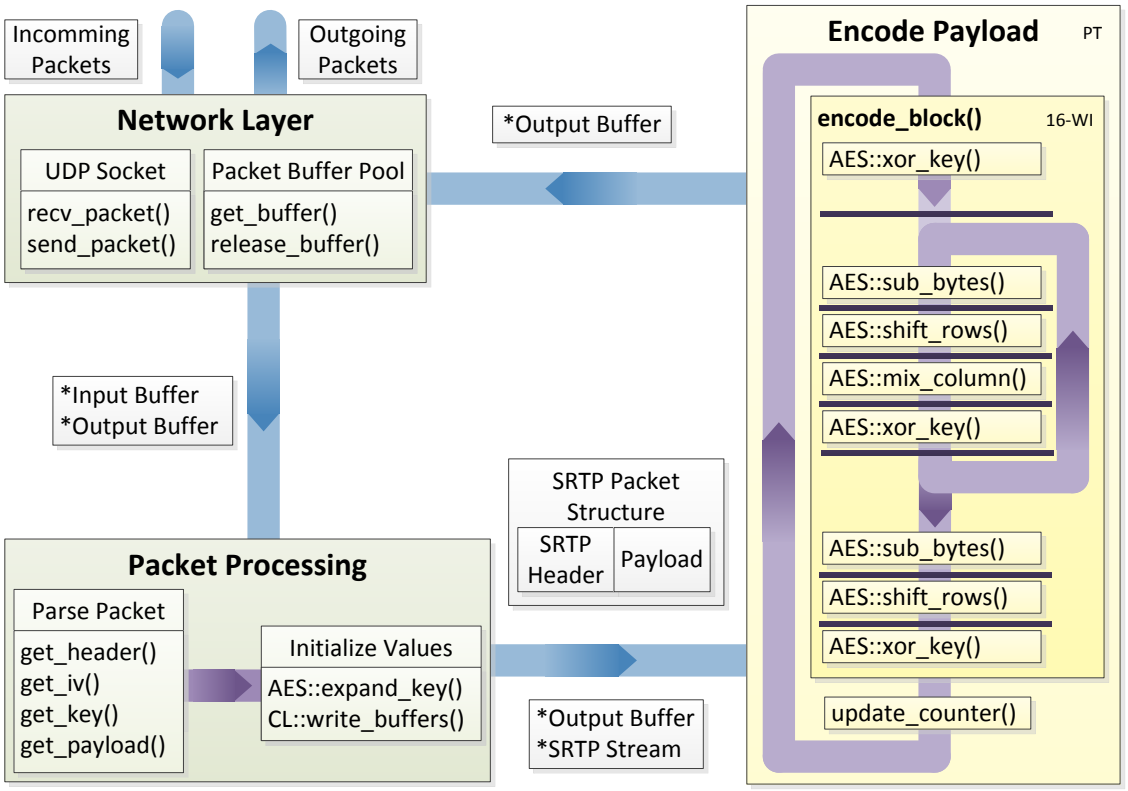


Figure 4.4: SRTP processing scheme.

The SRTP processing scheme from figure 4.4 visualizes the ideas behind the design of SRTP stack and encapsulates implementation details for easier explanation from the multi-threaded application design point of view. The entire stack runs in three separate threads which shall minimize the delay caused by waiting on modules with varying time of execution per packet.

- **Network Thread** – the incoming and outgoing data are captured via two sockets, for RTP and RTCP. This thread includes a pool of buffers for the storage of packets and another the processed data.
- **Stack Thread** – the interaction and selection attributes for the processing thread is taken care in the stack thread as well as interface for higher layers of the application using the SRTP stack.
- **Packet Processing Thread** – extraction of important values from the packet header, encoding and decoding provided with transcoding interface of the entire packet payload according to the previously extracted data.

The thread design could be mapped to another type of view on the layers of the stack design. The SRTP layers as shown in the scheme 4.4 are subset of the entire SRTP stack functionality and the classes from the scheme have following purposes:

- **Network Layer** – enables the communication with external devices through SRTP protocol, transfere of the multimedia data packets between endpoints and implements buffer pool for packet data.

- **Packet Processing Layer** – without unneccessary data reallocation the proper structures are casted for easier readability and extraction of important data from the incomming packets.
- **Payload Encoding** – complete implementations of encryption and decryption of the packet payload.

Serial Processing

The designed application captures data from network in the network layer which ensures communication with both endpoints of multimedia session and is running in its own thread. It contains buffer pools for incoming and outgoing data to ensure maximal level of parallelization in each layer of application. Pointers for input and output buffers are passed for further packet processing where are extracted information such as header and payload from the packet, copied data from the memory to OpenCL data structures and serial implementation of AES key schedule.

For better understanding of improvement this thesis is provided with reference serial implementation which design will be analyzed as well. The payload encryption design as visualized in figure 4.4 shows, that the execution is separated into multiple consecutive callings of AES algorithm with updating of counter in between for CTR mode. Thanks to the decomposition of the code, the design of parallel encryption is done in similar fashion.

The design prevents the executing implementations from creating any additional temporary buffers to decrease unnecessary allocations. These can be predicted in the startup of SRTP stack and already preallocated with maximal size a packet can have. This approach consumes more memory, but improves memory management and saves execution time during critical sections. Also it is assumed that softgate gateway code runs on machine with enough memory and these buffers most certainly shouldn't mean any excessive memory consumption.

The buffer pools provided are used in both serial and parallel encryption, and to increase the level of algorithm categorization, their implementation is based on template classes. The pool guarantees to protect buffer from overwrite and data persistency.

Massive Parallel Processing

Traditional parallel programming style relies heavily on SIMT[3] and SPMD[4] programming paradigms [21]. The native OpenCL approach is based on abstracting the units of work from the programmers code into virtual threads – work-items. The convenience it offers in allocation of resources brings couple of limitations as well.

The figure 4.5 demonstrates the processing of packet payload of G.711 codec on the chip with work-group size 16. Workload on the work-items can be highly irregular and each work-item execution is finished after the processing of the particular AES block, therefore, this code will need 160 invocations of work-items during the kernel execution. That would bring unnecessary computational overhead.

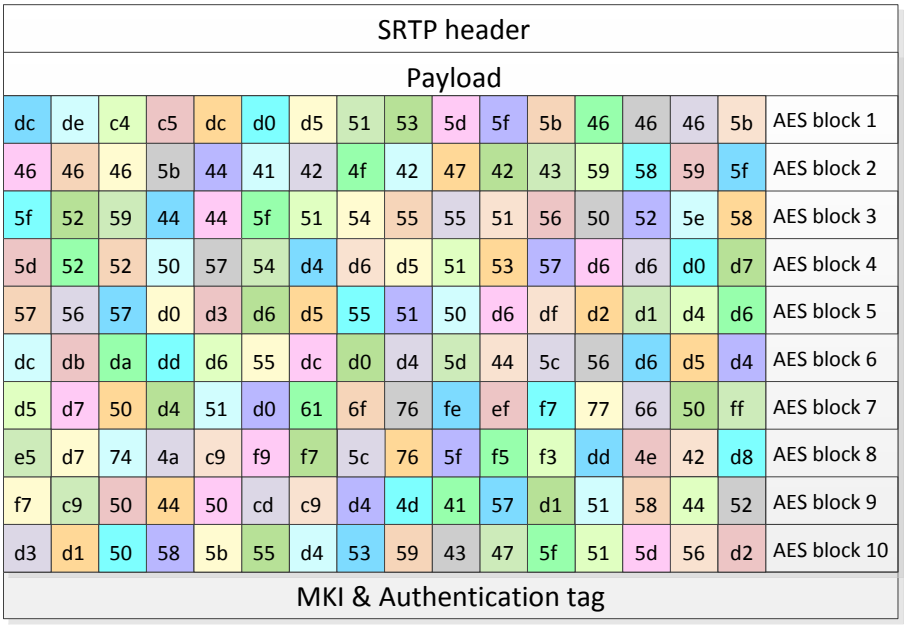


Figure 4.5: Work-item mapping on packet payload with native OpenCL approach.

Persistent Thread Processing

The requirements and attributes imposed by massive parallel processing style divide the workload into multiple blocks, more than can be simultaneously executed during kernel launch time, and the synchronization is ensured by the OpenCL. For massive parallel applications the obvious approach would be to utilize as much of machine's power as possible to gain the largest speed-up in every single execution. However, the aim of this thesis is to minimize large delays for multiple sessions which requires rather careful allocation of resources. Persistent threads is special type of programing paradigm combining both, the possible gain of mapping the program for parallel computation and considerate usage of resources [23].

Since the initialization of computational kernel can consume significant amount of time compared to the actual execution, larger kernel reusing its resources for multiple similar computations could render the initialization negligible trading off portion of parallelization.

[3] SIMT – Single Instruction Multiple Thread
[4] SPMD – Single Program Multiple Data

This approach has been chosen for packet parsing, while instead of mapping 160 OpenCL work-items on the G.711 packet's payload it uses one work-item for each AES block cell in a loop that goes through the data.

Maximal simultaneous work-items launched during the kernel execution is equal to the number of blocks in AES algorithm and it must not be larger than the amount of work-items in work-group. The persistent thread style provides couple of relevant improvements that are not resolved in common parallel implementation to the satisfactory degree.

- **Global synchronization** – as the kernel uses only as many work-items as can be simultaneously scheduled, the tools OpenCL offers for synchronization within work-group can be used to synchronize calculations through the entire execution at any given point which is used in synchronization across AES blocks for update of *round key*.
- **Computational overhead** – the amount of computations in a work-item for 128-bit AES is larger than initialization, startup and cleanup of the kernel, but those factors are not completely insignificant. Limiting the number of consecutive executions decreases the ratio of OpenCL overhead and the algorithm performance in positive way.
- **Resource requirement consistency** – memory requirements are similar for both persistent thread and massive parallel style, but size of the payload for single packet may consume up to 160 work-items on the GPU if kernel is programmed in non persistent thread style. As it doesn't seem to be much for one packet, if the stack should take care of multiple SRTP streams, the resources may promptly become insufficient which will increase the weight of OpenCL overhead. Persistent thread kernel will not use more than 16 work-items per packet.

SRTP header																
Payload																
dc	de	c4	c5	dc	d0	d5	51	53	5d	5f	5b	46	46	46	5b	AES block 1
46	46	46	5b	44	41	42	4f	42	47	42	43	59	58	59	5f	AES block 2
5f	52	59	44	44	5f	51	54	55	55	51	56	50	52	5e	58	AES block 3
5d	52	52	50	57	54	d4	d6	d5	51	53	57	d6	d6	d0	d7	AES block 4
57	56	57	d0	d3	d6	d5	55	51	50	d6	df	d2	d1	d4	d6	AES block 5
dc	db	da	dd	d6	55	dc	d0	d4	5d	44	5c	56	d6	d5	d4	AES block 6
d5	d7	50	d4	51	d0	61	6f	76	fe	ef	f7	77	66	50	ff	AES block 7
e5	d7	74	4a	c9	f9	f7	5c	76	5f	f5	f3	dd	4e	42	d8	AES block 8
f7	c9	50	44	50	cd	c9	d4	4d	41	57	d1	51	58	44	52	AES block 9
d3	d1	50	58	5b	55	d4	53	59	43	47	5f	51	5d	56	d2	AES block 10
MKI & Authentication tag																

Figure 4.6: Work-item mapping on packet payload using persistent thread paradigm.

4.4 Transcoding

Essential part of the media server is ability to negotiate the best codec for both endpoints in real-time media session. When all participating devices can not communicate using the same compressing media codec, the gateway must be able of transcoding to provide the channel for communication. RTP protocol defines 127 different codecs for audio and video profile. Therefore, the designed SRTP stack uses plugin system for multimedia codec transcoding.

The design of plugin system takes into consideration the lifetime of packet data buffers and for optimization purposes may defer the release of buffers on the side of codec plugin. The core desing is simple and consists of two parts.

- **Plugin System Module** – part of the gateway, on the startup browses defined directory for any plugins and dynamically links them into the application. The plugin system may offer the management of packet memory buffers on the side of codec plugin, but the system doesn't guarantee the consistency through the entire lifetime and if necessary, the buffer may be rewritten, in which case the flag for data correctness is set off.

- **Codec Plugin** – compiled files implementing the plugin interface capable at least of both transcoding the codec from and into PCM [5] and preferably also optimized transcoding into another codec, if such algorithm is presented. The codec plugin is responsible for implementing or control of any buffers if necessary, concurrently transcode multiple different streams, and must separate the buffers and another saved information from given stream ID. The plugin may not use the optimization option, duplicate the data into its own buffers and keep the memory management on the part of SRTP stack itself.

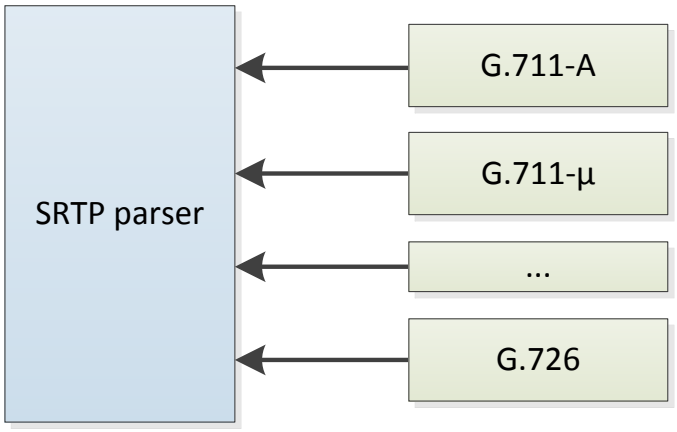


Figure 4.7: Interaction between plugin system on the gateway and separate codec plugins.

[5] PCM – Pulse-code modulation.

Chapter 5

Implementation

The application's specifications were variable through the life-cycle of the entire development. SIP gateway's implementation started as a prototype for translating LCP protocol and subset of SIP protocol enabling basic functionality for new lighweight prototype telephones. As the statement of requirements included reference Java application that combined LCP server and multiple SIP clients, the implementation languages differ from the SRTP stack which is the core of this thesis.

As mentioned before, the reference application implements only a subset of the full communication protocols and instead of understanding all of the complex scenarios the protocols offer, it brings research value examining the possibilities of improvement implementing computationally demanding algorithms using parallel programming paradigm. Another benefit this work brings, is experimental study and comparison of established implementations used either commercially or free.

5.1 SIP Gateway

5.1.1 SIP Layer

5.1.2 LCP Layer

5.2 SRTP Stack

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Pellentesque pretium lectus id turpis. Fusce wisi. Nunc tincidunt ante vitae massa. Nulla non lectus sed nisl molestie malesuada. Nulla quis diam. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Phasellus enim erat, vestibulum vel, aliquam a, posuere eu, velit. Curabitur ligula sapien, pulvinar a vestibulum quis, facilisis vel sapien. Quisque tincidunt scelerisque libero.

5.2.1 Buffer Pool

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Pellentesque pretium lectus id turpis. Fusce wisi. Nunc tincidunt ante vitae massa. Nulla non lectus sed nisl molestie malesuada. Nulla quis diam. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Phasellus enim erat, vestibulum vel, aliquam a, posuere eu, velit. Curabitur ligula sapien, pulvinar a vestibulum quis, facilisis vel sapien. Quisque tincidunt scelerisque libero.

```
template <class buffer_item> class Buffer_pool {
    private: //implementation
        buffer_item **pool = nullptr;
        std::queue<int> free_buffer_index;
        int pool_size;

    public: //interface
        // initialize pool and free_buffer_index
        Buffer_pool(int pool_size) { .. };
        // cleanup of resources
        ~Buffer_pool() { .. };
        // returns available buffer ID
        int get_buffer_id() { .. };
        // returns size of pool
        int get_pool_size(){ .. };
        // returns available buffer with ID
        buffer_item* get_item(int id) { .. };
        // makes buffer with ID available
        void release_buffer(int id) { .. };
};
```

To achieve higher degree of code reusability and categorization, buffer pool is implemented as template class offering both the OpenCL implementation use its own pool for memory objects on the host side and network interface use its own pool for incomming and outgoing packets and their structures.

```
class RTP_item {
    public:
        RTP_item();
        ~RTP_item();
        // packet data buffers
        BYTE src[PACKET_SIZE];
        BYTE *payload_src;
        BYTE dst[PACKET_SIZE];
        BYTE *payload_dst;
        BYTE temp[PACKET_SIZE];
        // IPv6 BSD socket structures
        struct sockaddr_in6 src_addr;
        struct iovec iov[1];
        struct msghdr msg;
};
```

The buffer pool item for network packets includes the buffers for incomming and outgoing packets and the pointers to payload to avoid perpetual header removal. Network interface uses GNU C standard implementation of BSD sockets, therefore, this item includes structures for the sockets as well to keep the information persistent through packet processing.

```
class cl_item{
    public:
        cl_mem payload_src; //buffer for payload
        cl_mem payload_dst;
        cl_mem iv_gpu;      //buffer for initial vector
        cl_mem rk;          //round key
        cl_mem t1;          //auxiliary buffers
        cl_mem t2;
};
```

OpenCL requires specific memory objects that can be loaded to the device memory. Buffer pool item for parallel implementation requires only those memory objects.

5.2.2 AES

For the SRTP stack the crucial is implementation of Advanced Encryption Standard. Described are two most relevant implementations, serial for comparison with the current solutions and persistent thread as the representative of the the best examined parallel implementation.

The header file `aes.h` offers the functions for encryption and decryption of packet payload in CTR mode encapsulated in `AES` namespace.

Key Schedule

The master key defined for every SRTP session must be expanded into round key, which is used in every round for encryption. The algorithm for round key schedule is provided only in serial implementation, because its execution is prompt and doesn't provide enough calculations for the kernel device that would justify parallel implementation and diminish OpenCL computational overhead.

```
//precomputed rcon table
static const BYTE rcon[]      = {0x8d, ... };
//key expansion
void AES::expand_key(BYTE *mk, BYTE rk[ROUND_KEY_SIZE][BLOCK_SIZE]){
    get_first_rk(mk, rk[0]);
    for(int i = 1; i<ROUND_KEY_SIZE; i++){
        get_next_rk(mk[i-1], rk[i], rcon[i]);
    }
}
```

Key expansion algorithm is different for the first round key and the rest of round keys. Both methods are described in this chapter.

```
void get_first_rk(BYTE *key, BYTE *rk){
    memcpy(rk, key, 16*sizeof(BYTE));
}
```

The first round key is copy of first 128 bits from the master key.

```
void get_next_rk(BYTE *previous_key, BYTE *rk, BYTE rcon){
    rotate_column(previous_key, rk, 0, 3, 1);
    for(int r = 0; r<ROWS; r++){
       rk[r*COLUMNS] = sbox[rk[r*COLUMNS]] ^ previous_key[r*COLUMNS];
    }
    rk[0] = rk[0] ^ rcon;
    for(int c = 1; c<COLUMNS; c++){
        for(int r = 0; r<ROWS; r++){
            rk[r*COLUMNS+c] = previous_key[r*COLUMNS+c] ^ rk[r*COLUMNS+c-1];
        }
    }
}
```

Other round keys are derived from the previous round key. The algorithm is devided into two steps, first calculates the first column of the key state and then are calculated columns two, three and four always derived from the previous column. Because every array in the SRTP stack is considered one dimensional exactly as the incomming packet, the index to the state matrix memory must be computed exclusively.

Serial Encryption

The AES encryption is divided into 4 steps exactly as described in previous chapters. Five constants necessary for AES will be mentioned through this chapter in many code snippets and are listed in the following algorithm for encryption. Their definitions are later skipped to avoid information redundancy.

```
//AES constants
#define ROUND_KEY_SIZE 11
#define ROUNDS         10
#define BLOCK_SIZE     16
#define ROWS            4
#define COLUMNS         4
//CTR encryption algorithm
void AES::srtp_encode(BYTE *src, BYTE *dst, BYTE *key, BYTE *iv, int len){
    xor_key(key,iv,key);
    expand_key(key,round_key);
    int i = 0, j = 0;
    int last_offset = len/BLOCK_SIZE*BLOCK_SIZE;
    //encryption of counter and XORing with blocks
    for( ; i < length; i+=BLOCK_SIZE){
        encode_block(counter, dst+i, round_key);
        xor_key(dst+i,dst+i,src+i);
        update_counter(counter);
    }
    //encryption of counter is full but XORing only up to packet size
    BYTE *last_block = dst+last_offset;
    encode_block(counter, last_block, round_key);
    for(i=i-BLOCK_SIZE; i < length; i++, j++){
        dst[i] = last_block[j] ^ src[i];
    }
}
```

The function takes five arguments, four of those are input parameters and one is output, result of the encryption. The first argument `src` is pointer to the buffer with packet payload

that needs to be encrypted, `dst` is pointer to the buffer with outgoing packet payload after encryption, arguments `key` and `iv` represent encryption properties and finally the last argument `len` has length of the packet payload.

```
void encode_block(BYTE *counter, BYTE *dst, BYTE **round_key){
    BYTE temp[BLOCK_SIZE];
    xor_key(counter, temp, round_key[0]);
    for(int i = 1; i < ROUNDS; i++){
        sub_bytes(temp, dst);
        shift_rows(dst, temp);
        mix_columns(temp, dst);
        xor_key(dst, temp, round_key[i]);
    }
    sub_bytes(temp,dst);
    shift_rows(dst,temp);
    xor_key(temp,dst,round_key[ROUNDS]);
}
```

Implementation of the block encode follows the algorithm described in chapter 2 in algorithm 1. Since the CTR mode encrypts the counter, packet payload doesn't have to be present.

```
//precomputed AES tables
static const BYTE g2[] = {0x00, ... };
static const BYTE g3[] = {0x00, ... };
//optimized multiplication in Galois field
void mix_columns(BYTE *src, BYTE *dst){
    BYTE a0, a1, a2, a3;
    for(int col = 0; col < COLUMNS; col++){
        a0 = src[col];
        a1 = src[col+4];
        a2 = src[col+8];
        a3 = src[col+12];
        //using precomputed tables
        dst[col]    = g2[a0] ^ g3[a1] ^ a2     ^ a3;
        dst[col+4]  = a0     ^ g2[a1] ^ g3[a2] ^ a3;
        dst[col+8]  = a0     ^ a1     ^ g2[a2] ^ g3[a3];
        dst[col+12] = g3[a0] ^ a1     ^ a2     ^ g2[a3];
    }
}
```

The multiplication in Galois field by matrix defined in chapter 2 is computationally expensive, therefore, the implementation consits of precomputed values stored in arrays and then the mixture of columns.

```
void xor_key(BYTE *src, BYTE *dst, BYTE *key){
    for(int i = 0; i < BLOCK_SIZE; i++){
        dst[i] = src[i] ^ key[i];
    }
}
```

```
void shift_rows(BYTE *src, BYTE *dst){
    for(int row = 0; row < ROWS; row++){
        rotate_row(src+(row*COLUMNS), dst+(row*COLUMNS), row);
    }
}
```

Both `xor_key()` and `shift_rows()` are simple and selfexplanatory functions. The `shift_rows` uses helper function `rotate_row()`.

```
static const BYTE sbox[] = {0x63, ... };
void sub_bytes(BYTE *src, BYTE *dst){
    for(int i = 0; i < BLOCK_SIZE; i++){
        dst[i] = sbox[src[i]];
    }
}
```

Following are two helper functions for AES algorithm that are neither described in the theoretical part of the thesis nor the AES definition document. Their implementation has single purpose – higher code readability.

```
void rotate_row(BYTE *src, BYTE *dst, int n){
    int max = COLUMNS-n;
    for(int i = 0; i < max; i++){
        dst[i] = src[i+n];
    }
    for(int i = max; i < COLUMNS; i++){
        dst[i] = src[i-max];
    }
}
```

The values in a row specified by arguments `src` and `dst` as pointers to the particular row in both blocks are rotated by the value `n`. Code is split into two cycles when both together cycle through the particular row only once.

```
void rotate_column(BYTE *src, BYTE *dst, int col1, int col2, int n){
    int max = ROWS-n;
    for(int i = 0; i < max; i++){
        dst_state[i*COLUMNS + col1] = src_state[(i+n)*COLUMNS + col2];
    }
    for(int i = max; i < ROWS; i++){
        dst_state[i*COLUMNS + col1] = src_state[(i-max)*COLUMNS + col2];
    }
}
```

Because AES block is stored in the linear memory by rows, when rotating columns it won't be sufficient to pass the pointer to specific column. Therefore, the function takes arguments `col1` as index of column in `src` block and `col2` as index of column in `dst` block. The argument `n` again defines the distance for rotation.

Parallel Encryption

Code for parallel encryption consists mostly of the kernel code in OpenCL language executed in device. The code for host is large but programmed rutines are common and doesn't display any efforts of improvement and own contribution.

Again the AES tables are precomputed and stored in constant memory space which is cached. As a result, a read from constant memory shouldn't cost more than one read from device memory on a cache miss. For the persisten thread reading from the constant cache is as fast as reading from a register as long as all threads read the same address.

```
1  //AES block encryption
2  void encode_block(BYTE *counter, BYTE *dst, BYTE **round_key){
3      BYTE temp[BLOCK_SIZE];
4      xor_key(counter, temp, round_key[0]);
5      for(int i = 1; i < ROUNDS; i++){
6          sub_bytes(temp, dst);
7          shift_rows(dst, temp);
8          mix_columns(temp, dst);
9          xor_key(dst, temp, round_key[i]);
10     }
11     sub_bytes(temp,dst);
12     shift_rows(dst,temp);
13     xor_key(temp,dst,round_key[ROUNDS]);
14 }
```

The round key schedule was moved from the kernel to host code. Other lines are similar to the serial code.

```
1  void encode_block(__local BYTE *counter, __local BYTE *dst,
2                    __local BYTE *temp, __local BYTE *round_key){
3      xor_key(counter, temp, round_key);
4      barrier(CLK_LOCAL_MEM_FENCE);
5      for(int i = 1; i < ROUNDS; i++){
6          sub_bytes(temp, dst);
7          barrier(CLK_LOCAL_MEM_FENCE);
8          shift_rows(dst, temp);
9          barrier(CLK_LOCAL_MEM_FENCE);
10         mix_columns(temp, dst);
11         barrier(CLK_LOCAL_MEM_FENCE);
12         xor_key(dst, temp, round_key+(i*BLOCK_SIZE));
13         barrier(CLK_LOCAL_MEM_FENCE);
14     }
15     sub_bytes(temp,dst);
16     barrier(CLK_LOCAL_MEM_FENCE);
17     shift_rows(dst,temp);
18     barrier(CLK_LOCAL_MEM_FENCE);
19     xor_key(temp,dst,round_key+(ROUNDS*BLOCK_SIZE));
20 }
```

Encoding the block requires local synchronization which is achieved by `barrier()` function. Each step of the algorithm may start computations after the previous step has been finished for all work-items.

```
void xor_key(__local BYTE *src, __local BYTE *dst, __local BYTE *key){
    int i = get_global_id(0);
    dst[i] = src[i] ^ key[i];
}
```

Because both global and local ID of the work-item is cached, calling function `get_global_id()` doesn't bring any slack-off, therefore, it is not necessary to pass the work-item ID to the functions of AES implementation.

```
void mix_columns(__local BYTE *src, __local BYTE *dst){
    int i = get_global_id(0);
    int row = i/4;
    int col = i%4;
    BYTE a0 = src[col];
    BYTE a1 = src[col+4];
    BYTE a2 = src[col+8];
    BYTE a3 = src[col+12];
    // mix only the byte this work-item belongs to
    if(row == 0){
        dst[i] = g2[a0] ^ g3[a1] ^ a2     ^ a3;
    } else if(row == 1) {
        dst[i] = a0     ^ g2[a1] ^ g3[a2] ^ a3;
    } else if(row == 2) {
        dst[i] = a0     ^ a1     ^ g2[a2] ^ g3[a3];
    } else if(row == 3) {
        dst[i] = g3[a0] ^ a1     ^ a2     ^ g2[a3];
    }
}
```

The work-item during kernel execution must identify itself and then find out to which branch of code it belongs.

```
void sub_bytes(__local BYTE *src, __local BYTE *dst){
    int i = get_global_id(0);
    dst[i] = sbox[src[i]];
}
```

```
1  void shift_rows(__local BYTE *src, __local BYTE *dst){
2      int i = get_global_id(0);
3      int row = i/4;
4      int col = i%4;
5      // shift only the byte this work-item belongs to
6      if(row == 0){
7          dst[i] = src[i];
8      } else if(row == 1) {
9          if(col == 3)
10             dst[i] = src[i-3];
11         else
12             dst[i] = src[i+1];
13     } else if(row == 2) {
14         if(col <= 1)
15             dst[i] = src[i+2];
16         else
17             dst[i] = src[i-2];
18     } else if(row == 3) {
19         dst[i] = src[i-1];
20     }
21 }
```

The kernel code requires work-item categorization again as the rows shifting doesn't provide any means for generalization. This branching in mentioned device code doesn't bring any additional computations nor increase overhead.

5.2.3 Transcoding

The plugin system was implemented with usage of the GNU C Library. All of the important functions for run-time dynamic loading are included from header `dlfcn.h`.

- `dlopen()` – loads dynamic library file with `RTLD_GLOBAL` and `RTLD_NOW` flags set to make the symbols available for subsequently loaded libraries and perform eager symbol resolution.
- `dlsym()` – returns address where the particular symbol is loaded into memory.
- `dlclose()` – unloads the dynamic library.

The plugin module searches folder `plugins` for any file ending with extension `.so` and performs plugin test whether the file contains necessary properties.

```
PAYLOAD_TYPES 127 //[RFC3551]
// structure for codec plugin
struct Codec {
    int PT = -1;
    char* encoding_name = nullptr;
    int (*transcode)(BYTE *src, BYTE *dst, int l1, int *l2, int pt, int id);
    void (*to_raw)(BYTE *src, BYTE *raw, int l1, int *l2, int id);
    void (*from_raw)(BYTE *raw, BYTE *dst, int l1, int *l2, int id);
};
// list of plugins
static Codec transcode_plugins[PAYLOAD_TYPES];
// transcode module interface
int transcode(BYTE *src, BYTE *dst, //packet buffers
              int l1, int *l2,      //data lengths
              int pt1, int pt2,     //codec types
              int id);              //stream ID
```

The code sniplet above defines the structure for loaded codec plugins on the host application side and interface for plugin module and rest of the SRTP stack. Transcoding is always executed through `transcode()` function and never directly, because the function handles the possibilities of plugins and selects the option with best effort ratio and in the worst case transcodes through PCM.

```
// codec identification [RFC3551]
extern const char* encoding_name;
extern const int PT;
// transcodes multimedia data from one codec to another codec
int transcode(BYTE *src, BYTE *dst, int l_src, int *l_dst, int pt, int id);
// transcodes codec to raw PCM
void to_raw(BYTE *src, BYTE *raw, int len_src, int *len_dst, int id);
// transcodes raw PCM to codec
void from_raw(BYTE *raw, BYTE *dst, int len_src, int *len_dst, int id);
```

The plugin must define all of the mentioned interface values and functions. To ensure proper implementation, each codec plugin includes header file `transcode_plugin.h`.

5.3 Visualization Tool

Chapter 6

Results

One of the major delays caused on gateway during indirect calls is due to encryption and transcoding. Since the core of this thesis was parallelization of SRTP encryption, the tests and measurements focus on gathering relevant information regarding especially the correct usage of SRTP on the gateway. Even though AES was designed to be fast algorithm, while executed on large amount of flowing data it can cause measurable overhead.

For the purposes of this chapter there was designed loadtest with following attributes:

- each test fulfilled these properties:
  - 300 subscribers executing the calls
  - 50 to 150 concurrent calls in the same time[10]
  - BHCA 2000[1]
- each call from the tests mentioned above:
  - lasted 20 seconds
  - used G.711-a or G.711-μ codec
  - needed encryption of SRTP

The tests were executed on the machine running 32-bit OpenSUSE 12.2 with similar hardware as HiPath4000 softgate is equiped. The following list summarizes the softgate properties and used software products for compilation of SRTP stack.

- processor intel i5 2500k with HD3000 graphics chip
- 8 GB RAM
- OpenCL 1.2
- gcc 4.7

The commercial gateway with optimized hardware and software can hold around 120 concurrent calls [10], therefore, the tests did not aim for much higher numbers and the analysis of the results focus around the known maximal number.

[1] BHCA – busy hour call attempts

6.1 Packet Encryption

The most accurate results can be reached by measuring directly in the code for packet encryption on the gateway. Also this aproach offers the easiest way for comparison with well-known implementations to increase creadibility of the improvement of the reference implementations distributed with this thesis.

AES

For the comparison of effectivity of the AES implementation, in table 6.1 are given time requirements for execution of multiple blocks encrypted by 128-bit AES with the implementation proposed in this thesis and reference implementations M5t[2], EfAesLib[3] and OpenSSL[4].

Implementation	1 block	5 blocks	10 blocks	15 blocks	20 blocks
M5t	26 µs	32 µs	46 µs	69 µs	82 µs
Serial	16 µs	28 µs	48 µs	66 µs	86 µs
Persistent thread	44 µs	45 µs	44 µs	44 µs	45 µs
Massive Parallel	55 µs	55 µs	54 µs	55 µs	62 µs
EfAesLib	XX µs	XX µs	XX µs	XX µs	XX µs
OpenSSL	XX µs	XX µs	XX µs	XX µs	XX µs

Table 6.1: Comparison of selected AES implementations.

The serial implementation of 128-bit AES is presumptively similar in execution speed as the M5t AES implementation, which is the reference value for measuring the degree of improvement. Therefore, later provided results of round-trip time delay can be assumed as relevant for the evaluation the total contribution. Higher execution time for the M5t implementation could be partially due to negative influence of not using the M5t framework in the SRTP stack implementation, therefore, conversion between structures and data types used might hindered the performance.

6.2 Round-trip Time Delay

The total delay is combination of many factors and even though information about measured time of packet encryption on the gateway is exact, it doesn't provide the most important information about how affected the session is overall and what is the delay on endpoints. The same test was executed once more but measurements were collected on modified virtual clients.

Graphs 6.1 and 6.2 include packet delay during concurrent calls. Every captured SRTP packet has been entered into results. Single column shows the distribution of delays during the particular number of concurrent calls where the thicker area of the column visualizes 90% of the packets. The remainding 10% were considered as abnormal and thanks to

[2] M5t framework includes AES implementation currently used in many Siemens devices[5].
[3] EfAesLib is a highly optimized Advanced Encryption Standard library[3].
[4] OpenSSL is open-source implementation of SSL and TLS protocols including cryptographic functions[28].

satisfying the common limitations of real-time communications [31], the exceeding 5% can be neglected.

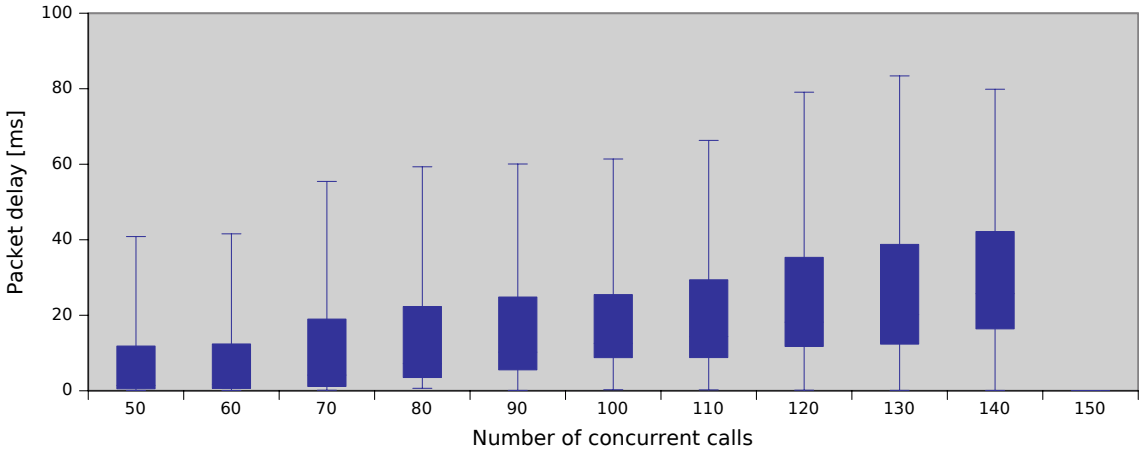


Figure 6.1: Visualization of distribution of delays during SRTP sessions with serial encryption implementation.

Predictably the average delay of the packet increases with the amount of concurrent calls in figure 6.1 for serial implementation. The time for 150 concurrent calls was not included due its excessive values which would make readability of the graph more difficult.

Even though the increase of delay seems to be linear, higher number of concurrent calls shows that the increase is exponential, which can be visible on the figure 6.2 for persistent thread implementation.

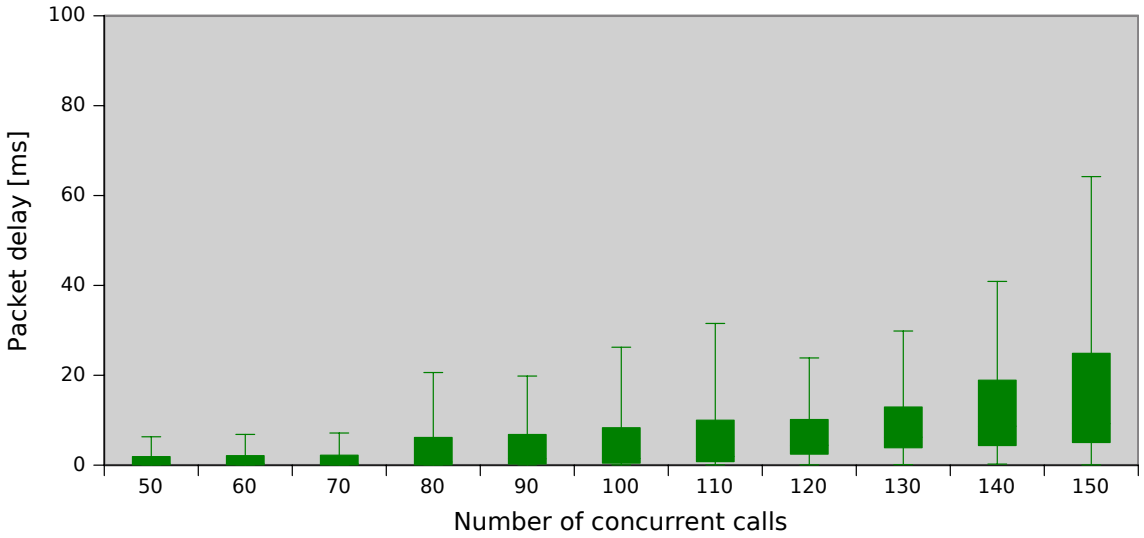


Figure 6.2: Visualization of distribution of delays during SRTP sessions with parallel encryption implementation.

Less predictable increase is visible in persistent thread implementation. Inconsistent extreme values of delays may be produced by host and device synchronization. Also memory management for packet buffer pool and OpenCL buffer pool are two separate implemen-

tations, slowdown in one pool may have negative effect on the other, therefore, both may combine in a negative way.

Comparing persistent thread implementation and serial implementation, the average delay was dropped to one half for smaller amount of concurrent calls (50-90) and the best results of speed-up was achieved in 140 concurrent calls where the average delay was dropped to one third.

The figure **??** centralizes on the most interesting results from previous tests which is the delay for 120 concurrent calls. The detailed distribution of packet delays shows, that parallel implementation, visualized in blue color, has the peak situated around 12 ms when the serial implementation, visualized in purple color, has the most packet delays situated around 27 ms.

Chapter 7

Conclusion

There are lot of possibilities for optimization of SRTP processing. Selected approach focuses on methods of parallelization of encryption and decryption processes of default AES cipher, which offers large potential thanks to recent development in the field of parallel computational units.

Proposed architectures and designs are currently far from being complete. The most effort was invested in correct analysis and understanding of basic principles of further implemented algorithms and elementary knowledge of parallel programming paradigm focused on usage of OpenCL framework for general-purpose computations on graphical processing unit. Modern GPU concentrate large amount of computational power, which could be to a certain extent utilized, if the algorithm is correctly mapped for parallel execution. That brings unusual complications in design whose must be carefully considered.

Another important milestone is definition of integration of RTP stack with SRTP processing into implemented SIP Gateway and their mutual interaction. The SRTP processing is only a fraction of overall load on the gateway and if measured separately while producing exact experimental results may not be equal to the actual results on deployed machine experiencing real traffic.

For the further development number of issues must be taken for notice. For instance the delay generated by the processing of separate SRTP packets should be reliably masked and interpolated across the SRTP stream to reduce possible jitter. On the other hand stands the actual delay of incoming packet, since after certain absolute value the conversation quality becomes unbearable.

Nevertheless, partial value of this thesis lies in the understanding of current technologies for future potential direction of development and exploration of new options in the field communication infrastructure.

Bibliography

[1] AMD Accelerated Processing Units [online]. http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx. Published 2011-6-8, accessed 2012-12-28.

[2] AMD and Leading Software Vendors Continue to Expand Offerings Optimized for OpenCL Standard [online]. http://www.amd.com/us/press-releases/Pages/offerings-optimized-for-opencl-2011jun08.aspx. Published 2011-6-8, accessed 2012-12-28.

[3] EfAesLib – AES Library [online]. http://www.codeproject.com/Articles/57478/A-Fast-and-Easy-to-Use-AES-Library. Accessed 2013-5-7.

[4] Intel HD Graphics [online]. www.intel.com/content/www/us/en/architecture-and-technology/hd-graphics/hd-graphics-developer.html. Accessed 2012-12-28.

[5] M5T Framework [online]. http://www.media5corp.com/m5t-framework. Accessed 2013-5-7.

[6] NVIDIA OpenCL SDK Code Samples [online]. http://mlso.hao.ucar.edu/hao/acos/sw/cuda-sdk/OpenCL/Samples.html. Published 2012-10-1, accessed 2012-12-28.

[7] Project Denver [online]. http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/. Published 2011-1-5, accessed 2012-12-28.

[8] Protocol Stack Design Pattern [online]. http://www.eventhelix.com/realtimemantra/PatternCatalog/protocol_stack.htm#.UYFRqbXQp-p. Accessed 2013-5-1.

[9] Securing Internet Telephony Media with SRTP and SDP [online]. www.cisco.com/web/about/security/intelligence/securing-voip.html. Accessed 2012-1-2.

[10] Siemens Hipath 4000 [online]. http://www.athlsolutions.com/web/en/Products/tabid/128/ProdID/38/Hipath_4000.aspx. Accessed 2013-3-3.

[11] Voice Over IP - Per Call Bandwidth Consumption [online]. http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a0080094ae2.shtml. Published 2006-2-2, accessed 2013-1-7.

[12] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[13] T. Adomkusv and E. Kalvaitis. Investigation of VoIP Quality of Service using SRTP Protocol. pages 195–209, 2008.

[14] A. L. Alexander, A. L. Wijesinha, and R. Karne. An evaluation of secure real-time transport protocol (srtp) performance for voip. In *Network and System Security, 2009. NSS '09. Third International Conference on*, pages 95 –101, oct. 2009.

[15] F. Andreasen, M. Baugher, and D. Wing. Session Description Protocol (SDP) Security Descriptions for Media Streams. (RFC 4568), 2006.

[16] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). (RFC 3711), 2004.

[17] C. E. Shannon. Communication Theory of Secrecy Systems. vol.28-4:656 – 715, 1949.

[18] M. Daga, A.M. Aji, and Wu chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141 –149, July 2011.

[19] J. Darlington, M. Ghanem, and H. W. To. Structured Parallel Programming. In *In Programming Models for Massively Parallel Computers*, pages 160–169. IEEE Computer Society Press, 1993.

[20] M. Dworkin. Recommendation for Block Cipher Modes of Operation. Federal Information Processing Standards Publication 800-38A, 2001.

[21] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, page 14, May 2012.

[24] P. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. (RFC 4566), 2006.

[25] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with copacobana – a cost-optimized parallel code breaker. In *Workshop on Cryptographic Hardware and Embedded Systems – Ches 2006, Yokohama*, pages 101–118. Springer Verlag, 2006.

[26] A. Munshi, B.R. Gaster, T.G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. OpenGL Series. Prentice Hall, 2011.

[27] P. O'Doherty and M. Ranganathan. JAIN SIP Tutorial - Serving the Developer Community. Technical report.

[28] OpenSSL Project.

[29] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrĂźger, A. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[31] C. Perkins. *RTP: Audio and Video for the Internet*. Addison-Wesley, June 2003.

[32] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. (RFC 3261), 2002.

[33] N. P. Tran, M. Lee, S. Hong, and S. J. Lee. Parallel Execution of AES-CTR Algorithm Using Extended Block Size. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 191 –198, August 2011.

[34] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP. (RFC 6189), 2011.

Appendix A

AES Properties

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table A.1: S-box for SubBytes transformation in hexadecimal notation.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table A.2: Inverse S-box for SubBytes transformation in hexadecimal notation.

Algorithm 3 AES decryption

Decipher(State, Key)
state ← $AddRoundKey$(State, Key[n])
state ← $ShiftRows$(state)
state ← $SubBytes$(state)

for $i \leftarrow (n-1..1)$ **do**
    state ← $AddRoundKey$(state, Key[i])
    state ← $MixColumns$(state)
    state ← $ShiftRows$(state)
    state ← $SubBytes$(state)
end for

state ← $AddRoundKey$(state, Key[0])

return state