UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

TELECOMMUNICATION ENGINEERING

Yang Qian

APPLIED CRYPTOGRAPHY IN EMBEDDED SYSTEMS

Master ´s thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, 26 October, 2013.

Supervisor Prof. Mohammed Elmusrati

Instructor Tobias Glocker

ACKNOWLEDGEMENT

This thesis is aimed to study both the principle and practice of cryptography and security for embedded systems.

First of all, I would like to express sincere appreciation to my thesis instructor Tobias Glocker for his tremendous and patient instruction and guidance during my thesis composing process. Moreover, I shall present big thanks to the staffs in Vaasa University who have provided the essential equipment for my thesis. As well as my classmates and friends in Finland who encourage me and help a lot in study and daily life. At last, I sincerely express my gratitude to Tobias Glocker for his immense practical advice in completing my thesis.

TABLE OF CONTENT

ABBREVIATIONS

AES	Advanced Encryption Standard
BSGS	Baby Step/Giant Step Method
CM	Complex Multiplication
CRT	Chinese Remainder Theorem
DES	Data Encryption Standard
DHP	Diffie-Hellman Problem
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECC	Elliptic Curve Cryptosystem
ECPP	Elliptic Curve Primality Providing Method
ECDLP	Elliptic Curve Discrete Logarithm Problem
GF	Galois Field
HTTP	Secure Hyper-Text Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IDEA	International Data Encryption Algorithm
KDC	Key Distribution Centre
LCM	Least Common Multiple
LED	Light-Emitting Diode
MOV	Menezes-Okamoto-Vanstone attack

NAF	Non-Adjacent Form
NFS	Number Field Sieve
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OEF	Optimal Extensive Field
ONB	Optimal Normal Basis
PB	Polynomial Basis
PIN	Personal Identification Number
PKC	Public Key Cryptography
PRF	Pseudorandom Function
PRNG	Pseudorandom Number Generator
RAM	Random Access Memory
RC2	Rivest Cipher
RNS	Residue Number System
RSA	RSA Cryptosystem
SCA	Side Channel Attack
SD	Signed Digit
SEA	Schoof-Elkies-Atkin algorithm
SEC	Standard for Efficient Cryptography
TRNG	True Random Number Generator

UNIVERSITY OF VAASA
Faculty of Technology
Author: Qian Yang
Topic of the Thesis: Applied cryptography in Embedded Systems
Name of the Supervisor: Professor Mohammed Salem Elmusrati
Instructor: Tobias Glocker
Degree: Master of Science in Technology
Department: Department of Computer Science
Degree Program Degree Program in Information Technology
Major of Subject: Telecommunication Engineering
Year of Entering the University: 2010
Year of Completing the Thesis: 2013 Page: 98

ABSTRACT

Nowadays, it is widely recognized that data security will play a central role in the design of IT devices. There are more than billion wireless users by now; it faces a growing need for security of embedded applications.

This thesis focuses on the basic concept; properties and performance of symmetric and asymmetric cryptosystems. In this thesis, different encryption and decryption algorithms have been implemented on embedded systems. Moreover, the execution time and power consumption of each cryptography method have been evaluated as key performance indicators. CAESAR and AES are implemented for the microcontroller (ATmega8515). The STK 500 board is used for programming of the ATmega8515. Furthermore it is used for the communication between the microcontroller and PC to obtain the performance advantages of the cryptography methods. Time and power consumption are measured by using an oscilloscope and a multimeter. Furthermore the performance of different cryptography methods are compared.

KEYWORDS: Cryptography, Embedded System, AES, ECC, security, encryption, decryption

1. INTRODUCTION

The embedded systems and handheld devices have been widely developed in comparison to a few years ago. From video equipment to mp3 players, cars to smart phones, and washing machines to home thermostats, more and more embedded devices interact with the real world and are connected to the internet, and then it's very common that those devices meet attacks, hackers and threats. Security issues might result in physical side effect as potential damages, personal injury, and even death, so it will play a central role in the design of future IT systems.

Due to the rapid growth of network communication, embedded devices and other transactions face the challenge of an increasing demand of data security, which concerns authentication for user admission, intrusion detection as well as any forms of attacks. Therefore, the security requirements have become critical. The main security issues of embedded systems will encounter when the data is routed over communication channels such as Ethernet, Wi-Fi, WiMAX or Bluetooth. Unfortunately, the technology of security applied in desk computing and enterprise cannot be executed in embedded systems. But security issues for embedded systems are more than the problems being addressed for desktop computing.

The possibility of adding security can be specified by hardware or by implementing the cryptography algorithms in software. This project focuses on cryptography and the

implementation of a cryptographic algorithm to protect data by using encryption technology on embedded systems.

There are several requirements and challenges of the implementation of cryptography algorithms on embedded systems. Embedded systems are highly cost sensitive, the length of cryptography key cannot be too big; a slow running cryptographic algorithm will lead to a long waiting time. The cryptographic technology can be divided into the two most common algorithmic models: symmetric cipher model and asymmetric-key. Asymmetric-key algorithms are very computationally intensive compared to symmetric-key operations. Sufficient cryptographic algorithms need to be selected according to the hardware and processor of the embedded systems.

The thesis consists of six chapters. In the first three chapters, the theory of cryptosystem is explained, symmetric cryptography and asymmetric cryptography. Chapter four introduces several attack methods. After the theoretical introduction it follows the most significant part of the thesis, the experimental part. Chapter five describes the software and hardware for the implementation, as well as the implementation algorithms, result and analysis of the encryption and decryption. Conclusion and future development regarding to this topic is given in the last chapter “CONCLUSION AND FUTURE WORK”.

2. SYMMETRIC CRYPTOGRAPHY

Symmetric encryption is also known as conventional encryption or single-key encryption. The encryption and decryption processes are performed by using the same key. It contains five elements: Plaintext, Encryption Algorithm, Private Key, Ciphertext and Decryption Algorithm, as the symmetric cryptosystem model is showed in **Figure 1.** This system requires a strong encryption algorithm and the security of the private key.

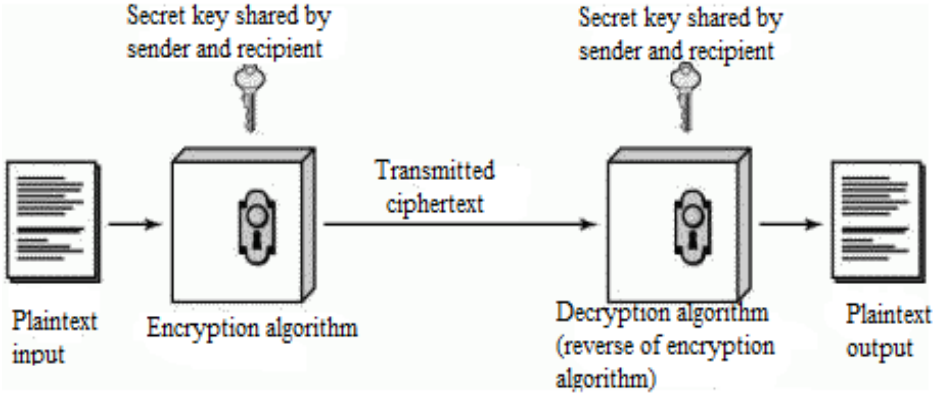

Figure 1. Simplified model of symmetric cryptosystem (Stallings 2011: 57).

2.1. Block Cipher Principles

Block Cipher is a type of symmetric encryption/decryption scheme that transform a fixed-length block of plaintext into a ciphertext block of the same length. The encryption transformation process is under the use-provided private key (See **Figure 2**). Decryption is the inverse process of encryption to the ciphertext using the same private key and will result in the original plaintext which was encrypted. Typically the block size is 64 bits or 128 bits.

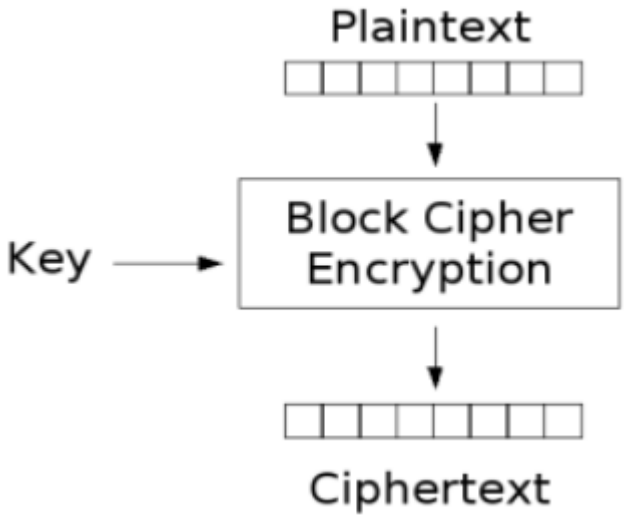

Figure 2. Block Cipher (Stallings 2011: 93).

There are many common block ciphers in use today. These are outlined in **Table 1**.

Table 1. Common Block Cipher Features (Ian McCombe 2007).

Name	**Block Size (bits)**	**Key Size (bits)**	**Year Developed**
DES	64	56	1975
RC2	64	8-128(default 64)	1987
AES	128	128,196 or 256	1998
IDEA	64	128	1991
Lucifer	48	48	1971
BlowFish	64	32-448(default 128)	1993
Intel Cascaded Cipher	128	128	2006

2.2. Data Encryption Standard (DES)

The Data Encryption Standard is adopted in 1977 by the National Institute of Standards and Technology (NIST), and the most widely used encryption scheme is based on it. The encryption process of DES is to transform a 64-bit input in a series of steps into a 64-bit output within a 56-bit key. The same key is used for the decryption process.

In **Figure 3**, the first step is permutation and the last step is inverse permutation, after permutation, the block is broken into two 32 bits blocks, the left one is L_i and the right part is R_i. Then there are 16 rounds of identical operations, but each of them uses an individual key K_i $L_i := R_{i-1}$ $R_i := L_{i-1} \oplus f(R_{i-1,}, K_i)$ (1)

In decoding, $R_i := L_{i-1}, L_{i-1} := R_i \oplus f(L_i, K_i)$ (2)

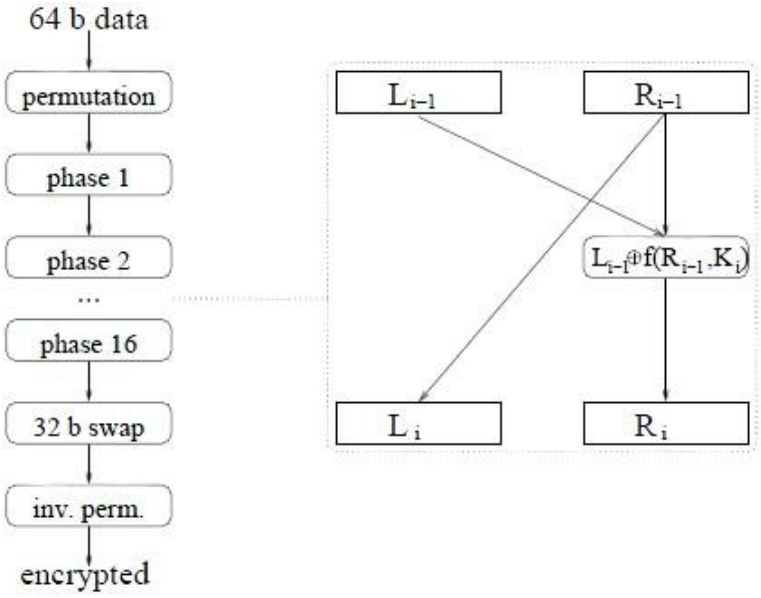

Figure 3. DES working process (Martti 2009:22).

Traditional DES has only 56 keys and therefore it does not meet the requirements of the current distributed open network data encryption security. DES is considered as unsafe after increasing of the clock rate of the computer.

2.3. Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is the most popular and secure symmetric system used in the professional industrial application, which is intended to replace DES for commercial applications. AES is a specification for encryption of the electronic data and it was published in 2001 by the National Institute of Standard and Technology (NIST). AES is the first publicly accessible and open cipher approved by the National Security

Agency (NSA) for the top secure information. In Comparison to AES, DES is insecure due to the small key. (Wikipedia AES 2012a.)

Sometimes the algorithm is called Rijndael, which is combined by the names of the two Belgian cryptographers, Joan Daemen and Vincent Rijmen. The basic structure of AES is substitution-permutation network, which can work fast on both software and hardware. The cipher takes the plaintext block size of 128 bits. The key sizes can be 128, 192 or 256 bits. (Wikipedia AES 2012a.)

AES operates on a 4×4 square matrix of bytes. This block is termed into the State array, and the AES cipher consists of a number of repetitions of transformation rounds, where the number of rounds depends on the key length (**Table 2**).

Table 2. Round and key length.

No. of rounds	Key Length (bytes)
10	16
12	24
14	32

The overall AES algorithm structure can be divided in the following steps as shown in **Figure 4**, which models the whole process of the AES encryption and decryption and indicates the sequence of the transformation in each round.

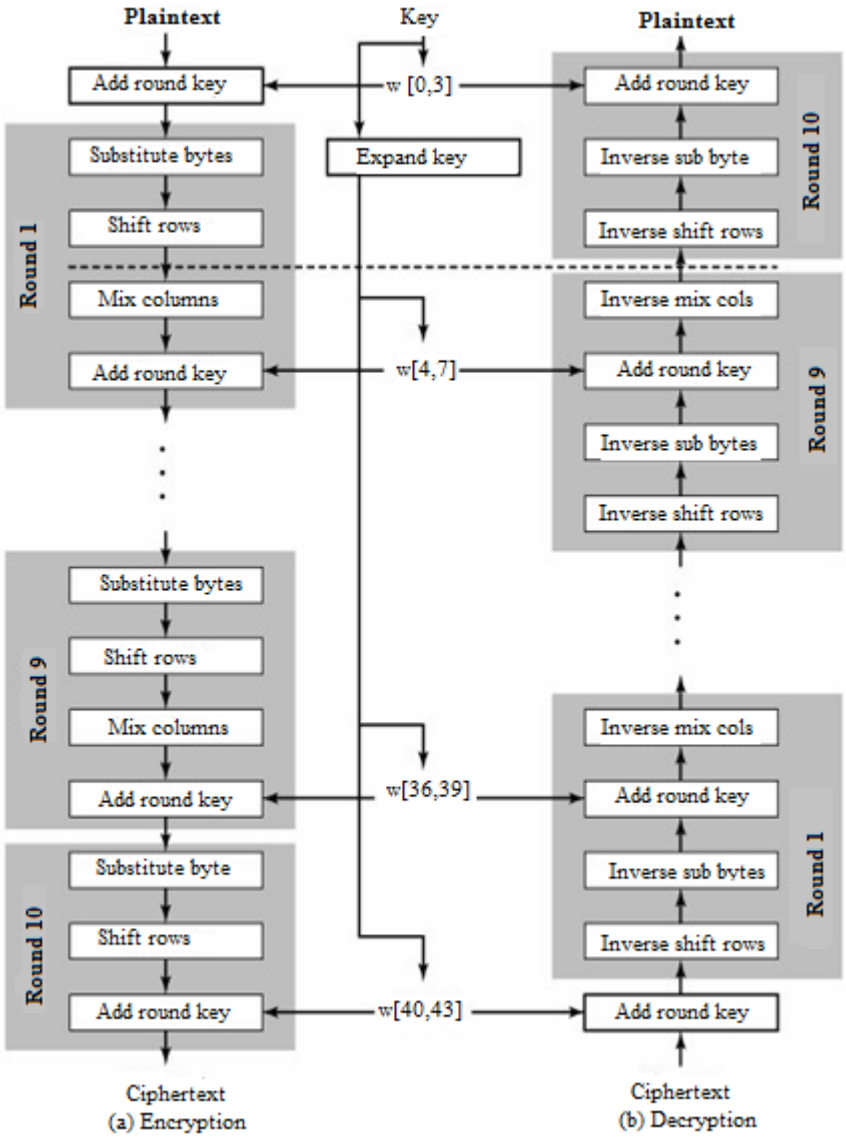

Figure 4. AES Encryption and Decryption (Stallings 2011: 178).

Refer to the **Figure 4**, the process of each steps can be listed as:

1) Key Expansion.

AES processes the data block as a single matrix during each round using substitutions and permutations. Round keys are derived from cipher key which is expanded into an array of forty-four 32-bit words.

2) Initial Transformation.

A simple bitwise XOR is applied to each byte of the state and the portion of the expanded key.

3) Rounds.

Four different stages are used; one of permutation and the others are substitution:

Substitute Byte

ShiftRows

MixColumn

AddRoundKey.

Those four stages are repeating each round except the final round.

4) Final Round (no *MixColumn*)

The final round of both encryption and decryption consist three stages:

Substitute Byte

ShiftRows

AddRoundKey.

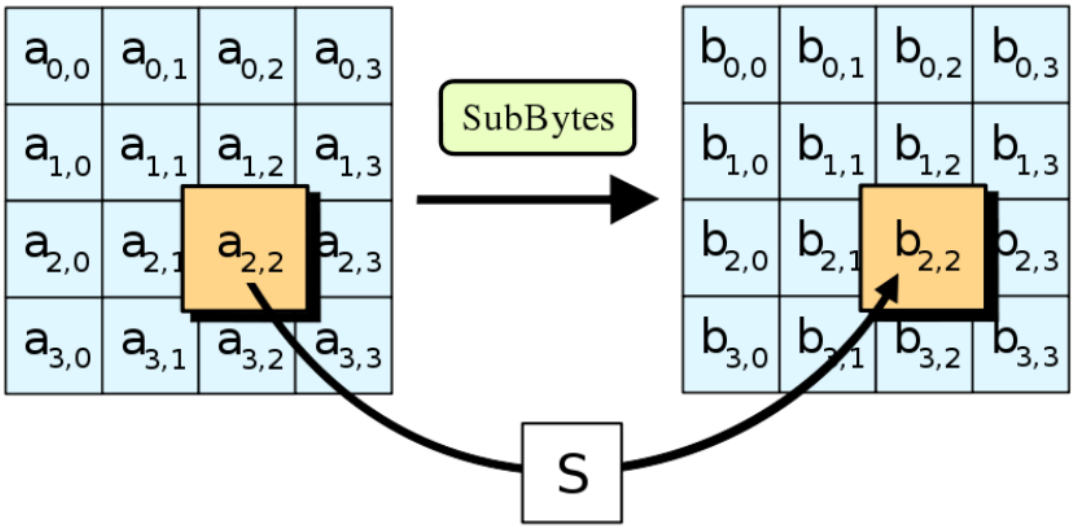

Figure 5. SubBytes step (Wikipedia AES 2012a).

Substitute Byte—each byte is replaced with another one according to an S-box (in APPENDIX 1), it is a non-linear substitution step shown in **Figure 5**. The element $a_{2,2}$ is replaced by the $b_{2,2}$ using the S-box. The S-box contains all possible 256 2-byte values for permutation. The substitution process is working in the following way: the left side byte is used as row value and the right side byte is used as column number. Then lookup the S-box within these row and column values to pick another 2-byte output value.

$$b_{i,j} = S\lfloor a_{i,i} \rfloor \quad (3)$$

Galois Field is called finite field, which is a field that contains a finite number of elements. The method of S-box substitution is based on the property of GF (2^8), the addition in GF (2^8) is XOR.

1) Inverse in GF(2^8), as the input element ωϵGF(2^8), the inverse element of ω is X:

$$\text{X}= \omega^{-1} = \begin{cases} \omega^{254} & \omega \neq 0 \\ 0 & \omega = 0 \end{cases} \tag{4}$$

2) Then the sub element of X form byte to bits are ($x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0$), According to (Stallings 2011: 178) the transformation is:

$$\begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \\ x'_5 \\ x'_6 \\ x'_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{5}$$

ShiftRows—is the row forward shift process. The first row remains the same. For the second row, shift to left 1-byte circular. For the third row, shift to left 2-byte circular. Then the fourth row, shift to left 3-byte circular.

The transformation can be expressed as:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix} \tag{6}$$

The whole process is represented in **Figure 6** clearly.

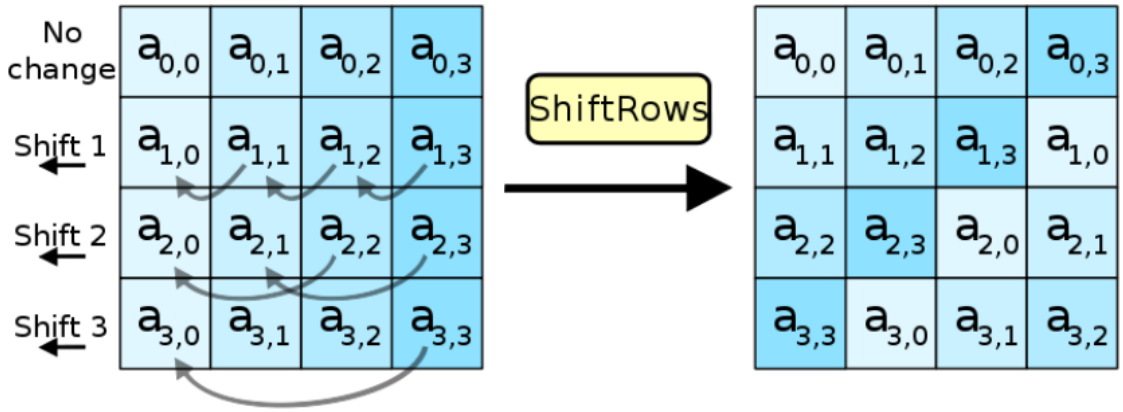

Figure 6. Shift Rows (Wikipedia AES 2012a).

MixColumn—is a forward mix column transformation. The mathematical model of intermixing between the different columns is in order to reach the confusion of the encrypted order. The whole process can be defined by the following mathematical model:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \tag{7}$$

AddRoundKey—is a forward and inverse transformation. This step is the process that 128 bits of the state are bitwise XORed with the 128bit of the round key. The mathematic model can be expressed as:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{8}$$

Refer to **Figure 7**; the detail of the transformation process is represented.

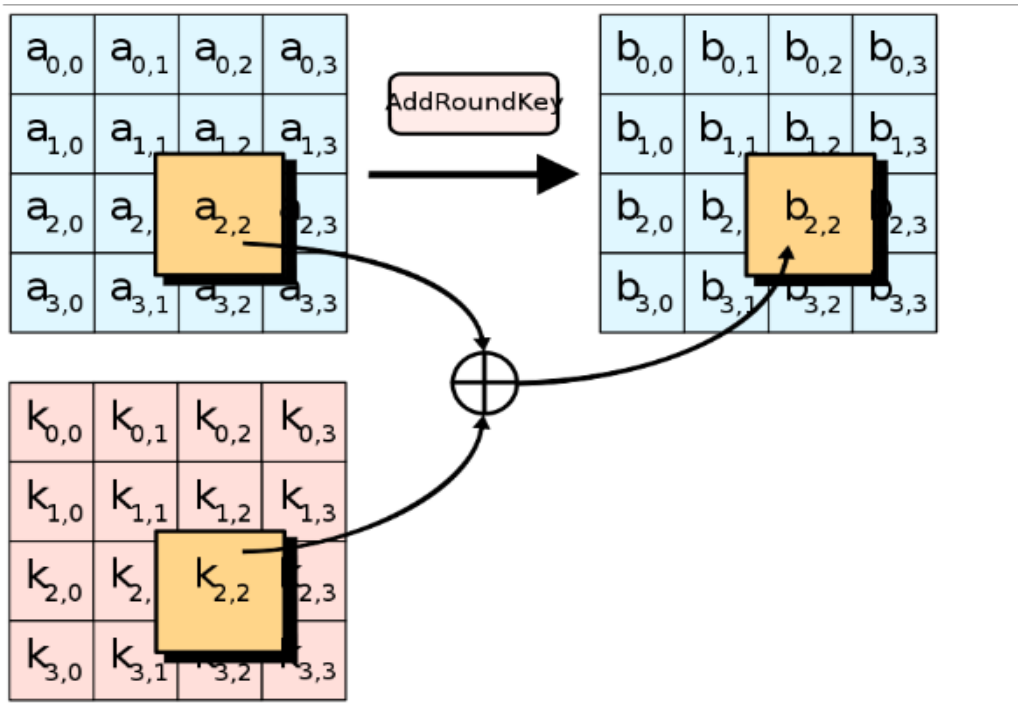

Figure 7. AddRoundKey (Wikipedia AES 2012a).

The algorithm for decryption makes use of the expanded key in reverse order. The step AddRoundKey is the same as in encryption. However, the decryption algorithm is no identical to the encryption algorithm. All the four stages are reversible, and encryption and decryption are going in opposite vertical directions.

2.4. Pseudorandom Number Generation and Stream Ciphers

The real random number (or random events) in a generating process is according to the experimental performance of distribution probability, the result is unpredictable, is not visible. The pseudorandom number is generated according to a certain algorithm simulation, the sequences of numbers that are not statistically random. And the result is certain and visible.

Random numbers are widely used in cryptography based on a number of network security algorithms and protocols. There are some random and pseudorandom number generators. TRNG is the true random number generator. It is the source of true analog randomness to a binary output. PRNG is a pseudorandom number generator. PRF is a pseudorandom function. Those two generators are used to produce pseudorandom numbers. Both require a fixed value as input, called the seed that should be different every time to guarantee randomness and unpredictability.

2.5. Blowfish

Blowfish is a substitute for the DES and IDEA encryption algorithm. It is a symmetrical block cipher (secret or private key), use that a variable key length from 32 to 448 bits.

(The U.S. government prohibits the encryption output software to use the key which key-length is more than 40, unless special-purpose software). Blowfish algorithm is an alternative encryption method, proposed in 1993 by Bruce Schneier. After the birth of the 32-bit processor, the speed of blowfish algorithm in the encryption beyond the DES attracted the attention of the people. Blowfish is a not registered patent, it can be used free. The round function is shown in **Figure 8.**

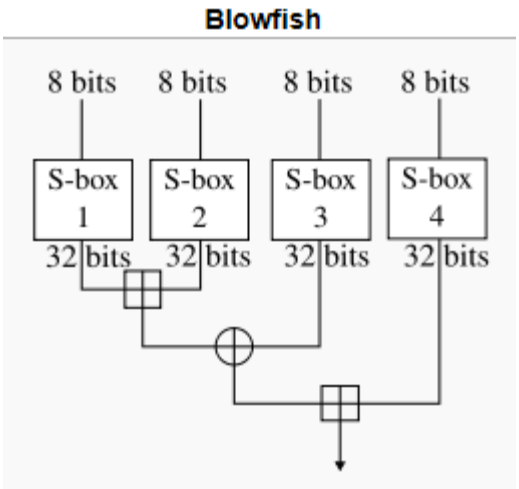

Figure 8. The round functions of Blowfish.

There are some features of blowfish:

- Blowfish is fast
- Blowfish needs only 5 KB of memory is easy to implement and compact
- Blowfish is considered secure
- Encryption consist 16+1 phases, each phase consists of $\oplus$, + and S-box operation
- Decryption is identical to encryption; keys are used in inverse order. (Martti 2009:35.)

3. ASYMMETRIC CRYPTOGRAPHY

Asymmetric cryptography is also known as public-key cryptography, which is a form of a cryptosystem in which encryption and decryption are performed by different keys. There is one public key and one private key. The public key is widely distributed, while the private key is kept secure.

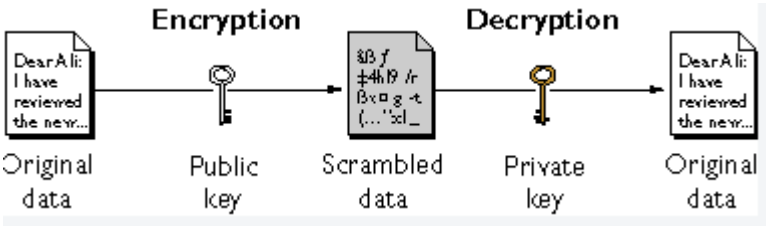

Figure 9. Simplified model of asymmetric cryptography (Information Security).

The process of asymmetric cryptography is to transform the plaintext into ciphertext by using the public key of the receiver (see **Figure 9**). The receiver decrypts the ciphertext with the private key. Anyone who wants to send a message to Alice can encrypt it using Alice's public key but only Alice can decrypt it with her private key. The private key should be kept secret at all times.

The use of public-key cryptosystem can be divided into three categories: Encryption/ decryption; digital signature; key exchange. (Stallings 2011: 57.)

3.1. The RSA algorithm

RSA is one kind of public-key algorithm; it stands for Ron Rivest, Adi Shamir, and Len Adleman who first publicly described it in 1978.

The RSA scheme is a block cipher in which the plaintext and ciphertext are integers between 0 and n-1 for some n. The public key is created and published by the product of two large prime numbers, along with an auxiliary value. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, only someone with knowledge of the prime factors can feasibly decode the message. At present RSA is the most widely used public key cryptography, which is based on the principle of trapdoor one-way function, as it is shown in the **Figure 10**,

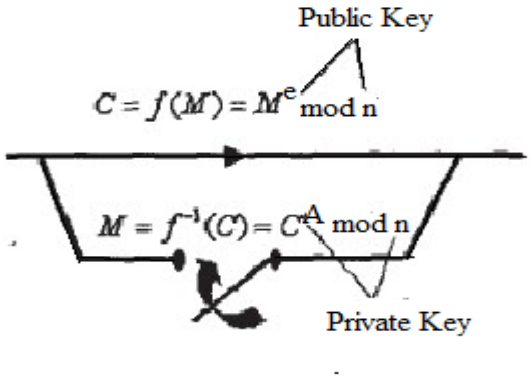

Figure 10. RSA is based on trapdoor one-way function principle.

The RSA algorithm is simple and easy to use. But as decomposition method of big integers is progressing, and the improvement of the speed of the computers and the

development of computer networks, the key length has to be increased in order to guarantee the safety of RSA. Increasing the key length will slow down the encryption and decryption speed. Hardware based implementation would be difficult. Thus RSA will be limited in terms of the key length.

Compared to DES, the speed of RSA is 1000 times slower than DES in hardware. In software, RSA is 100 times slower than DES. Those numbers might be changed slightly as technology changes, but the speeds of RSA can never approach the symmetric algorithms. Refer to the **Table 3**, it shows RSA speeds for different modulus lengths with 8-bit public key.

Table 3. RSA Speeds for Different Modulus Lengths with an 8-bit Public Key (J.B.Lacy 1993).

	512 bits	768 bits	1,024 bits
Encrypt	0.03 sec	0.05 sec	0.08 sec
Decrypt	0.16 sec	0.48 sec	0.93 sec
Sign	0.16 sec	0.52 sec	0.97 sec
Verify	0.02 sec	0.07 sec	0.08 sec

3.2. Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange is a security protocol. The math method is simple. Alice and Bob can use this algorithm to generate a secret key. First, Alice and Bob agree on a large prime, n and g, $g < n$ and g is a primitive root of n. These two integers don't need

to be secret; Alice and Bob can agree to them over some insecure channels, which even are common among a group of users.

The algorithm works as follows:

(1) Alice Key Generation:

$X = g^x \bmod n$, x is a large random integer

(2) Bob Key Generation:

$Y = g^y \bmod n$, y is a large random integer

(3) Calculation of secret key by Alice:

$k = Y^x \bmod n$

(4) Calculation of secret key by Bob:

$k' = X^y \bmod n$

Diffie-Hellman key exchange protocol can easily be extended to work with more people, just add more people and more rounds of computations.

This algorithm is not suitable for embedded systems. It can be used for the key distribution. Both sides can use this algorithm to generate a secret key, but it cannot be used for encryption and decryption of the message. (Stallings 2011: 327.)

3.3. El GAMAL Cryptographic System

The ElGamal algorithm is public-key cryptography which is based on Diffie-Hellman key exchange. ElGamal contains key generation, encryption algorithm and decryption

algorithm which were described by Taher Elgamal in 1984. It can be used for both digital signature and message encryption.

Key generation

Alice generates a key pair, first two random numbers are chosen, g and x; a prime p, g and x are smaller than p.

Alice computes $y = g^x \bmod p$, the public key is y, g and p, can be shared among groups of users. And x is kept private.

ElGamal Encryption

Plaintext is M, a random number k is chosen, k is relatively prime of p-1.

a, b are ciphertexts, the length is two times of plaintext,

$$a = g^k \pmod{p} \tag{9}$$

$$b = y^k M \pmod{p} \tag{10}$$

Decrypting: $M = b / a^x \pmod{p}$ (11)

ElGamal Signatures

The signing message is M, a random number k is chosen, k is relatively prime of p-1,

$$M = (ax+bk) \bmod (p-1) \tag{12}$$

This signature is the pair a and b. The value of k should be kept private.

Verifying: $y^a a^b \pmod{p} = g^M \pmod{p}$ (13)

Table 4. Gives some sample software speed of ElGamal (J.B.Lacy 1993).

	512 bits	768 bits	1024 bits
Encrypt	0.33 sec	0.80 sec	1.09 sec
Decrypt	0.24 sec	0.58 sec	0.77 sec
Sign	0.25 sec	0.47 sec	0.63 sec
Verify	1.37 sec	5.12 sex	9.30 sec

Table 4 shows that ElGamal when comparing the measurement with **Table 3** is slower than RSA.

3.4. Elliptic Curve Cryptosystem

In 1985, Miller and Koblitz firstly suggested to use Elliptic Curve in cryptography independently, which is based on the algebraic structure of elliptic curves over finite fields. Elliptic Curves are becoming more popular is because the keys size is much shorter than the public key systems which are based on the integer factorization or finite field discrete logarithm problem. Compared to RSA, the security level of ECC is higher. A key of 160 bits in ECC is secure as a 1024 bits RSA key as shown in **Table 5**. As a result, due to the short key length, the elliptic curve cryptosystem needs less bandwidth, less running time as well as lower power cost and it is suitable for the development of security products like PDA, mobile phone and embedded card .It will replace RSA in the near future. ECC becomes one of most efficient public-key cryptosystem.

Table 5. Key length of ECC and RSA with same security level.

ECC key length (bits)	RSA key length (bits)	Crack Time /MIPS (year)	ECC/RSA key length rate
106	512	10^4	5:1
160	1024	10^{11}	7:1
210	2048	10^{20}	10:1
600	21000	10^{78}	35:1

In general, cubic equations for elliptic curves take the **Weierstrass equation**:

$$y^2 + axy + by = x^3 + cx^2 + dx + e \tag{14}$$

Where a, b, c, d, e are real numbers and x, y take the values of real numbers. The elliptic curve can be seen as a set of all solutions to equations of the form:

$$y^2 = x^3 + ax + b \tag{15}$$

The curve discriminant equation is: Δ=-16($4a^3 + 27b^2$) (16)

A group can be defined on a set E (a, b) for specific values of a and b in Equation (14), the following condition is met:

$$4a^3 + 27b^2 \neq 0 \tag{17}$$

The process of ECC Diffie-Hellman Key Exchange can be done by the following step. First pick a large integer q, which is a prime or an integer of the form of 2^m. Then the

elliptic curve parameters a, b must be applied for Equation (14) or (15), which defines the elliptic group of the points $E_q(a,b)$. In the next step a base point G= (x_1, y_1) in $E_q(a,b)$ is selected, whose order is larger than value n.

A key exchange between user Alice and Bob can be described in **Figure 11**.

User Alice Key Generation	
Select private n_A	$n_A < n$
Calculate public P_A	$P_A = n_A \times G$
User Bob Key Generation	
Select private n_B	$n_B < n$
Calculate public P_B	$P_B = n_B \times G$
Calculation of Secret Key by User Alice	
$K = n_A \times P_B$	
Calculation of Secret Key by User Bob	
$K = n_B \times P_A$	

Figure 11. ECC Diffie-Hellman Key Exchange (Stallings 2011: 343).

ECC Encryption and Decryption

Choose Elliptic curve over GF (2^m) for instant, the preparation assignments are:

- Select GF(p)
- Select elliptic curve E
- Select base point G(x,y)
- Applied algorithm for transforming plaintext into the points of elliptic curve, called encryption process
- Generating the private and public keys between sender Alice and receiver Bob. Select one private key n, calculate the public key $P = nG = n(x, y)$. For Alice, the private key is n_A, and the public key is $P_A = n_A G = n_A(x, y)$. For Bob, the private key is n_B and the public key is $P_B = n_B G = n_B(x, y)$.

Encryption: Alice sends encrypted message to Bob

- Choose random number *k*, $1 \le k \le p-1$
- Get the corresponding points (x_m, y_m) by encoding the plaintext;
- Calculate the cipher text $C_m = \{k(x, y), (x_m, y_m) + kP_B\}$, and the cipher text here turns into two points on the elliptic curve.

Decryption, Bob decrypts the received message from Alice:

- Calculation

$$\begin{aligned} &((x_m, y_m) + kP_B) - n_B(kG) \\ &= ((x_m, y_m) + k(n_B(x, y))) - n_B(k(x, y)) \\ &= (x_m, y_m) + kn_B(x, y) - n_B k(x, y) \\ &= (x_m, y_m) \end{aligned} \tag{18}$$

- Get the corresponding plaintext by decoding the points (x_m, y_m).

The study and implementation of elliptic curve cryptography is now becoming a focus in public-key cryptosystems, and its foundation relies on the difficulty to solve the discrete logarithm of the elliptic curve Abelian group.

The set of points on the elliptic curve, together with a special point O called the point at infinity can be equipped with an Abelian group structure by the following addition operation.

Additional algorithm:

Input: modulus p, integer $a, b \in [p-1]$

Output: c=(a+b) mod p

$c_0 = a_0 + b_0$

For i from 1 to t-1 do: $c_1 = a_1 + b_1 + carry$

If carry =1, then c=c-p

If $c \geq p$, then c=c-p

Return (c)

Subtraction algorithm

Input: modulus p, integer $a, b \in [p-1]$

Output: c= (a+b) mod p

$c_0 = a_0 - b_0$

For i from 1 to t-1 do: $c_1 = a_1 - b_1 - carry$

If carry =1, then c=c+p

Return (c)

3.5. Hash Functions

A hash function is any algorithm that maps large data sets of variable lengths to smaller fixed length data sets. For instance, an address name, having a variable length, could be hashed to a single integer. The values returned by a hash function are called hash values, hash codes, hash sums, checksums or simply hashes.

Some common uses hash functions:

$$f(x) := x \bmod \max M \tag{19}$$

When max (M) is a prime and normal not close to 2^n;

$$f(x) := trunc((x / \max X) * \max longit) \bmod \max M \tag{20}$$

This is used for integer;

$$f(x) := (x * x div 1000) \bmod 1000000) \quad (21)$$

First Square meter then get the middle value.

In original sense, good hash functions are usually required to meet certain properties listed below:

Determinism--the hash procedure must be deterministic. It means for a given input value the output hash value must be the same.

Uniformity-- A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability.

Variable range—in many applications, the hash function range may be different for each run of the program.

Variable range with minimal movement—the hash table is refers to a dynamic hash table when the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk.

Data normalization—accomplishes normalizing the input before hashing it. That is, any two inputs that are considered equivalent must yield the same hash value.

Continuity—Hash function is used for searching similar data, which must be as continuous as possible.

3.6. Key Management and Distribution

Key management includes key generation, storage, distribution, using and destroy. The main target is to make sure that the key delivery via the public network is safely. A good key management system should include three aspects:

- Keys are hard to be stolen
- Under certain conditions, it should be useless to steal the keys because they have a limited lifetime
- Key distribution and exchange process are transparent to the users; users don't need to manage the keys personally.

The key distribution is the way that delivers a key to two parties who are intended to exchange secure encrypted data. A protocol is needed that provides a secure distribution of the keys. There are two kinds' keys that are involved in key distribution. Master keys are infrequently used and they last for a long time. The other is session keys, which are generated and distributed for temporary use between two parties.

Key distribution technique is a term that refers to the means of delivering a key between two parties that want to exchange data without allowing others to see the key. So far, the two main techniques are described Key Distribution Centre (KDC) and Diffie-Hellman. The process of Diffie-Hellman is described in chapter 3.2.

Kerberos is an authentication service designed for key distribution environment, it applies symmetric cryptography algorithms to establish a trustable third party KDC

verification system and verifies the authenticity of the two parties that communication with each other. The main function of Kerberos is to solve the key management and distribution. There are three parties in this communication: two communication parties that need to be verified and a trustable third party (KDC). Each party should only keep the encryption key with KDC secure, and KDC will safeguard the different encryption keys for individual users. When two parties want to communicate, they apply to KDC and KDC will encrypt the session keys by their individual keys .Then keys will be sent back (Stallings 2011: 435).The process is illustrated in **Figure 12**.

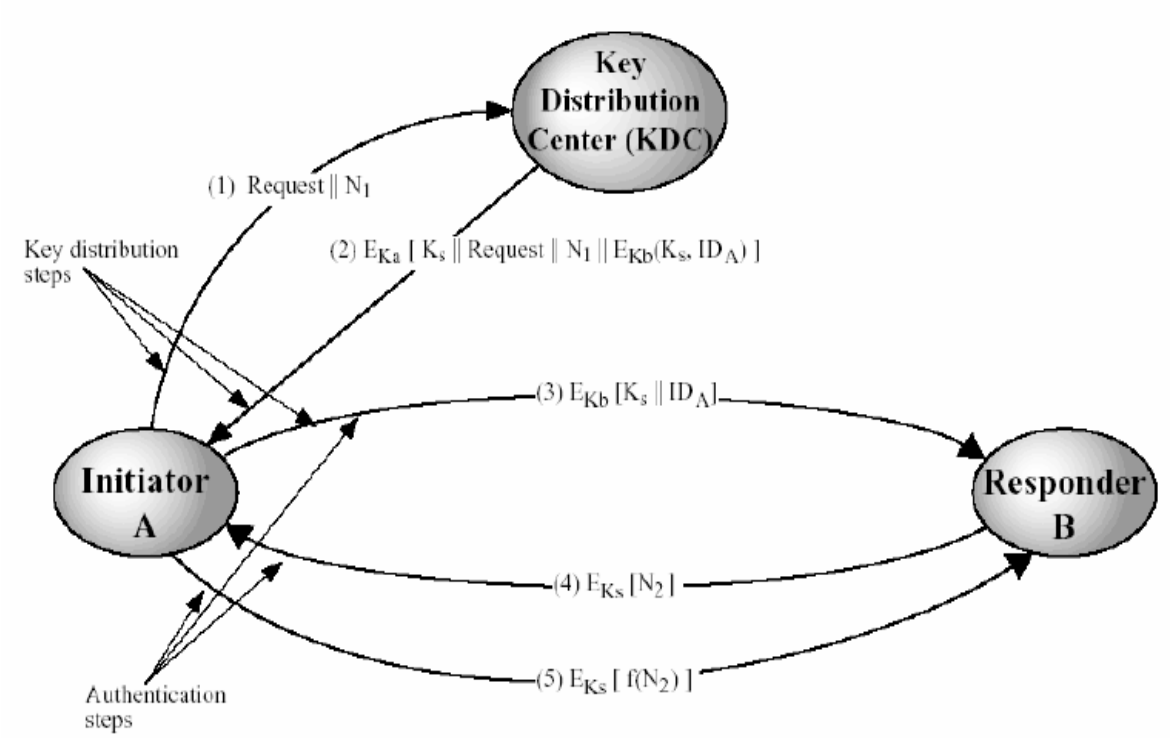

Figure 12. Key Distribution Scenario (Stallings 2011: 439).

3.7. User Authentication

In most computer security contents, user authentication is a mean of identifying the user and verifying that the user is allowed to access some strict network. For example, a user must be identified as a particular student to access the universities weboodi system or webmail service. A user must be identified as a member of IEEE to in order to view the IEEE materials. Furthermore, a user must be identified as a system administrator in order to access the document about the network administration. User authentication is the basis for access control and for user accountability.

There are two steps for remote user authentication:

- **Identification step**: Presenting an identifier to the security network, like user name and password.
- **Verification step**: Server and database. Generate authentication information to confirm the user's access right.

There are four possibilities that can be used individually or together to authenticate the users.

- **Individual knows**: A password, a PIN, or answers to a prearranged set of questions.
- **Individual possesses**: Tokens, like cryptography keys, smart cards, physical keys and electronic keys.
- **Static biometrics**: Fingerprint, face and retina.
- **Dynamic biometrics**: Voice pattern and handwriting characteristics

The remote User-authentication can be divided into two methods: mutual authentication and one-way authentication. Mutual authentication should consider the key distribution issues and should enable two communication parties to satisfy themselves mutually about the other's identity; one-way authentication can be applied for the e-mail system.

A Remote user authentication can use symmetric encryption and asymmetric encryption with Kerberos service.

4. ATTACKS

Attacks can be active or passive. An ‘active attack’ attempts to delete, add or use other method to affect the channel. A ‘passive attack’ only monitors the channel, does not affect the system resources.

Types of attacks:

- Passive Attack
    - Within ciphertext: attempts to get secret key or plaintext by observing ciphertext
    - Knows some plaintext and relative ciphertext, but this attack it difficult to realize
    - Knows some plaintext, attempts to know the encryption algorithms
    - Choosing relative plaintext to attack
    - Choosing ciphertext to attack
    - Choosing relative ciphertext to attack

Refer to **Figure 13**, which is shown the four attacks on Four Levels:

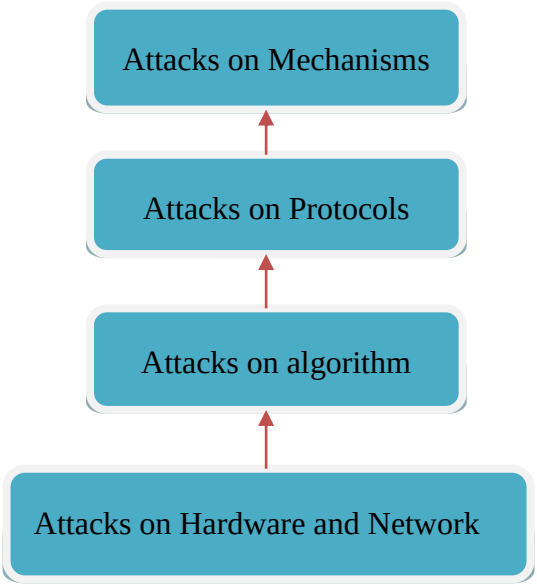

Figure 13. Four different levels attack.

Refer to the elliptic curve cryptosystem; most attacks on ECC are focusing on algorithms.

4.1. Attacks on Hardware and Network

It can be realized by a Side-Channel Attack (SCA), this method is powerful because there is no unified counter plan. SCA can be classified by invasive attacks and non-invasive attacks. Invasive attacks include probing and fault induction attacks. Non-invasive attacks include timing attacks and leaked-information attacks.

4.1.1. Power Consumption and Electromagetic Radiation Attack

In this kind of attack, the rival get system leaked information by means of measures or by analyzing the switch, current and power. Those information include hamming distance, bit string and operation order. Some attacks can even get the RAM information via CPU RAM address.

Similar, they can get the information via the equipment's electromagetic radiation. Because of electromagnetic radiation, an attacker can get data without getting close to the equipment.

4.1.2. Time Attacks

The target of time attack is a computational process nD, where n is fixed, and where D is a rational point on the elliptic curve.This kind of attack is to analyse the selected time. The principle is that for a software or a device, different input consumes time differently.

In theory, time randomization and process interrupt randomization are the ways to resist timimg attack. But in practice, those method are too strict, there is no perfect resist method.

4.1.3. Fault Induction Attacks

Fault Induction Attacks is to do wrong operation deliberately, get the secure information from the output result.

The method for assist this attack is simple, examine intermediate result. If the intermediate result not belongs to the curve point group, recalculate it.

4.1.4. Some Possible Countermeasures

- Non area differentiation in basic operation, at least make operation atomic;
- Group randomized, at least for base point;
- Check if the intermediate result is reasonable;
- Well stored precomputation result in hard disk;
- Electromagnetic shielding;
- Random process interrupt, random timing. (Avanzi 2005.)

4.2. Attacks on Algorithm

These kind attacks mostly rely on the mathematical algorithm, for ECC, it will aim to attack the ECC discrete logarithm ECDLP. Because of the features of ECC are complicated and attractive, it can be observed from different angles and get different properties. Meanwhile, the attacker obtains ideas from its characters.

4.2.1. Uncivilized search

Uncivilized search on elliptic curve is : a given curve E, point P and a random point Q, calculate P, 2P, 3P…until get Q=IP. The worst situation of this algorithm needs process n times elliptic curve addition, complexity is O(n).

4.2.2. Pohlig-Hellman algorithm

This algorithm makes use of factorization. By means of factoring n, the ECDLP how to solve *l* change into how to solve all the prime factors of n. Then regain *l* by CRT.

In order to withstand this attack, when we choose elliptic curve, the curve degree should be aliquot of a big prime n or be a big prime.

4.2.3. Baby-step Giant-step algorithm (BSGS)

We describe the BSBG method for a general finite abelian group, with n elements. By the Pohlig-Hellman simplification it can be assumed that n is prime. This method is the improvement of uncivilized search, but it costs more Random Access Memory (RAM). By means of precalculated and store the number of $\sqrt{n}$ elliptic curve points, the complexity of the worse situation can be decreased to O ($\sqrt{n}$) Pollard Rho Method. And the main problem of the method is that the storage space of O ($\sqrt{n}$) group elements.

This method in practice is a way of integer generation, and can be used for big integer factorization. When solving the ECDLP problem, Pollard Rho method simplify Baby-step Giant-step which saves memory space. After this improvement, the complexity of this method is around O ($\sqrt{n\pi/2}$). Based on this method, it is possible to parallelize Pollard Pho method, arithmetic complexity decreased to O ($\sqrt{\pi n/2r}$)

4.2.4. Semaev Smart Satoh Araki Attack

Prime Field Anomalous can solve ECDLP quickly. But this attack won't diffuse to other infinite field. It is to solve the ECDLP in subgroups of order p, where p is the characteristic of the field of the definition of the curve. An attack can be avoided by checking if the infinite element numbers are equal to the elliptic curve point members.

5. EXPERIMENTAL PART

In the experimental part, the content is about simulation regarding to security and performance on CAESAR and AES.

5.1. Hardware for Simulations

The STK500 board is manufactured by ATMEL in Sweden and it is equipped with an ATmega8515 microcontroller (see **Figure 14**), starter kit for 8-bit AVR. These microcontrollers are available in different configurations. It consists of a RS-232 interface to PC for programming and control, an additional RS-232 port for general use. It works with a regular power supply for 10-15V DC power.

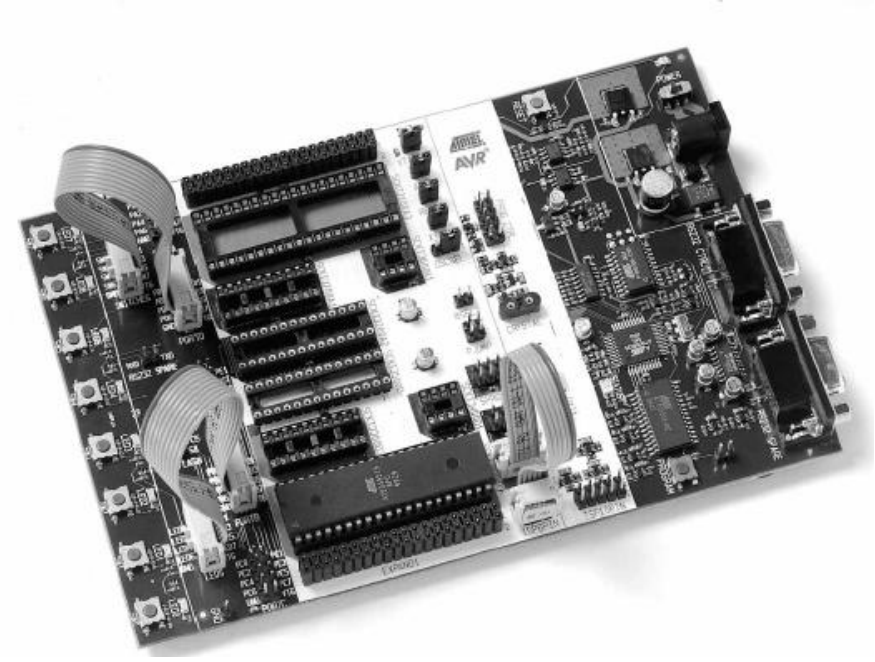

Figure 14. ATMEL STK 500.

The key features of STK500 I used are listed below:

- RS232 Interface to PC for programming and control
- Regulated power supply for 10-15V DC power
- Sockets for 8-pin, 20-pin, 28-pin, and 40-pin AVR devices
- Parallel and Serial High-Voltage Programming of AVR devices
- Serial In-System Programming (ISP) of AVR devices
- In-System Programmer for Programming AVR devices in External Target System
- 8 Push-buttons for general use
- 8 LEDS for general use
- All I/O ports easily accessible through pin header connectors
- Additional RS232 port for general use
- Expansion connectors for plug-in modules and prototyping area

And the key Parameter Value of the chipcon is listed below:

- Flash (Kbytes): 8 Kbytes
- Pin Count: 44
- Max. Operating Frequency: 16 MHz
- CPU: 8-bit AVR
- # of Touch Channels: 16
- Hardware QTouch Acquisition: No
- Max I/O Pins: 35
- Ext Interrupts: 3
- USB Speed: No
- USB Interface: No

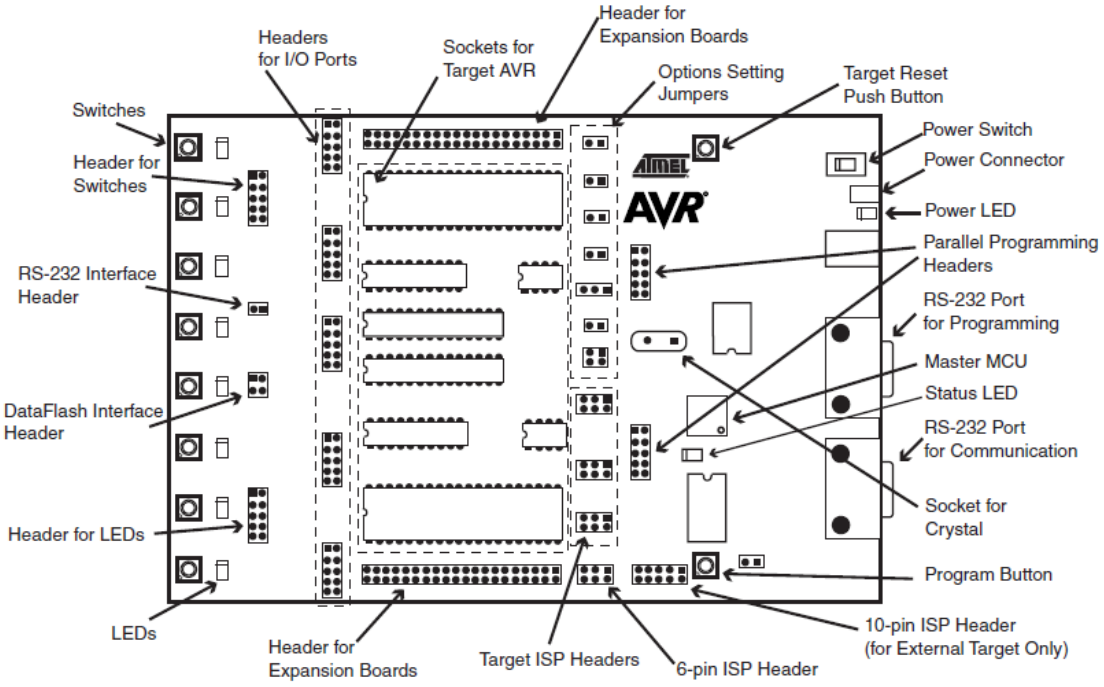

Figure 15. STK 500 Components (ATMEL User guide).

5.2. Software used for implementation and for testing

The software which is used to program the microcontroller on STK500 development board is Atmel Studio 6.0. Realterm is used for the communication between the PC and the microcontroller. The FrontPage of Atmel Studio 6.0 is shown in **Figure 16**.

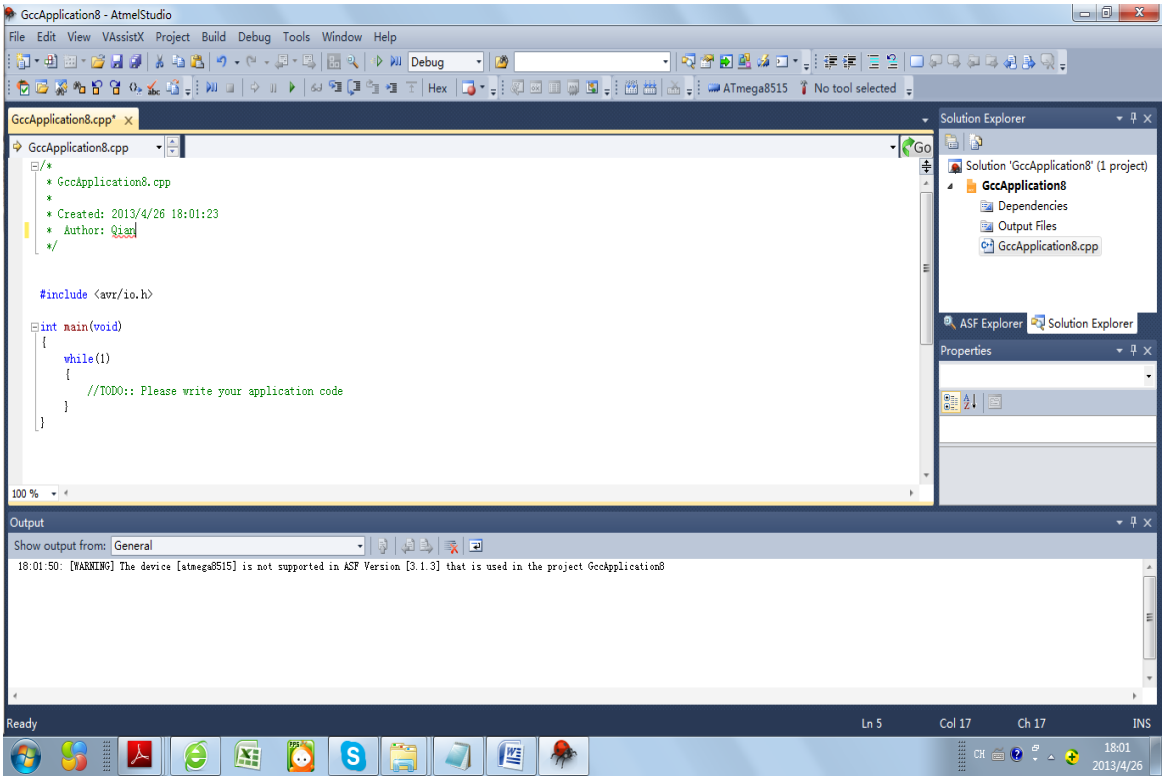

Figure 16. AtmelStudio 6.0 FrontPage.

The experiment will be done with Realterm, which is created mostly to represent a better alternative to the ubiquitous HyperTerminal application. In this project, the most impressive part of this application is the fact that it can emulate almost any kind of terminal used for serial communication. The FrontPage of Realterm is shown in **Figure 17**.

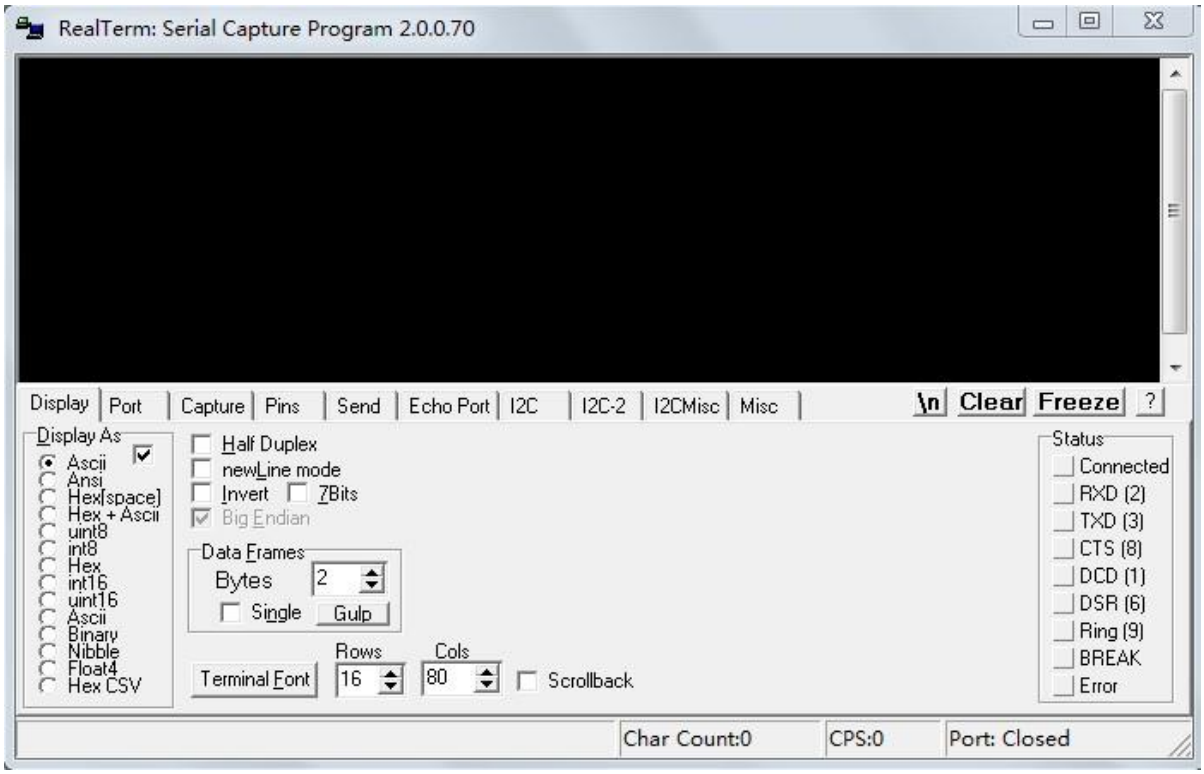

Figure 17. The Realterm FrontPage.

5.3. Selection and Implementation of Cryptographic Algorithms

The experiment is done with three programs containing the implementations of cryptography algorithms CAESAR and AES. The procedure is shown in **Figure 18**. First the program that contains the selected crypto graphical algorithms is compiled and uploaded to the microcontroller. This is done with Atmel AVR studio. In this case serial cable must be connected with the programming port of the STK500 board. Then the serial cable must be connected with the communication serial port in order to communicate with the microcontroller over the Realterm software.

In **Figure 18**, the STK 500 is connected to the PC via RS232 port. There are two RS232 ports on STK 500, one is for programming which is occupied when using Atmel studio 6.0; the other one is for communication when communicates with Realterm.

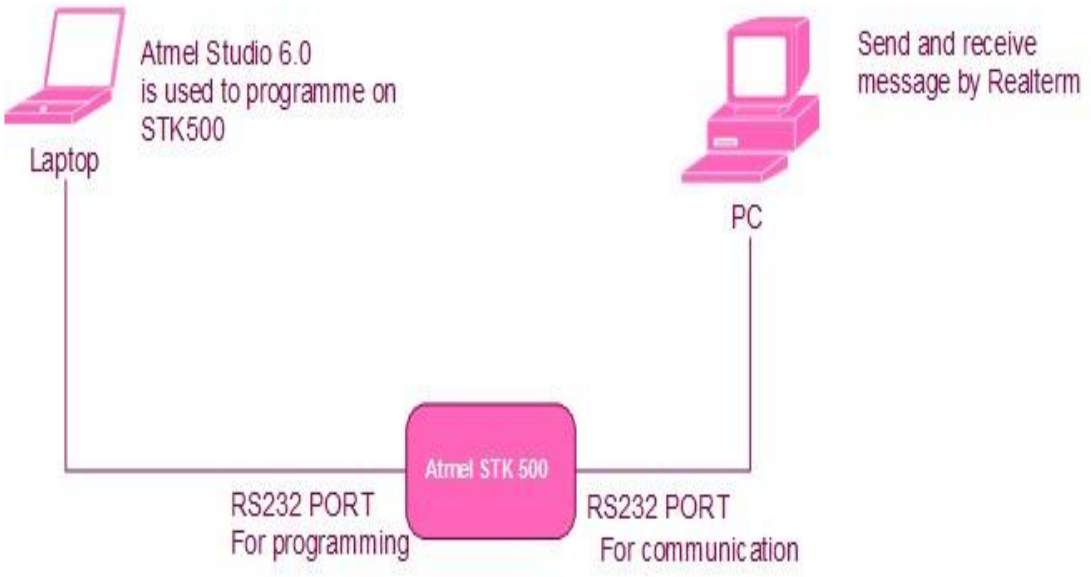

Figure 18. Schematic Diagram.

A. CAESAR

CAESAR encryption (Caesar cipher), known as shift cipher, is the simplest encryption method. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. Each letter in the alphabet will be replaced with a constants right shift k.

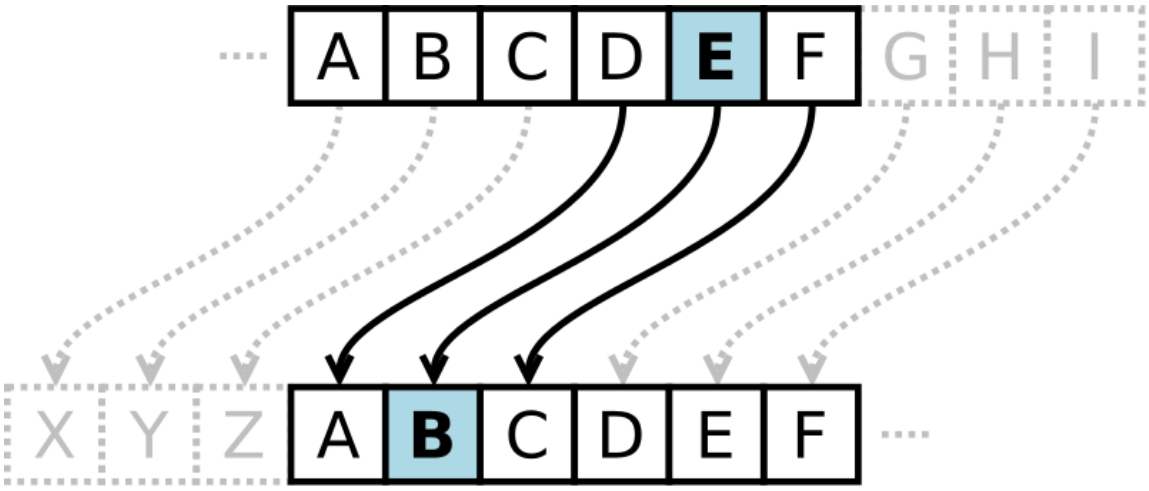

Figure 19. CAESAR letter left shift of 3 (Wikipedia CAESAR 2013).

See **Figure 19**, for example if k equals 3, each letter would move forward by three, and A will be replaced by D. B is replaced with E, and so on. The letters which sit in end of the alphabet will roll back to the beginning. So, W will be replaced by Z, X will be replaced by A.

B. AES

Due to the less security of DES, AES is a specification for encryption of the electronic data and it was published in 2001 by the National Institute of Standard and Technology (NIST). It aims to develop a royalty-free cryptographic technique for public authorities and private department. There are no known approaches for an attack in case of AES-128. (Biryukov & Khovratovich 2009.)

Table 6. AES Parameters (The AES Cipher).

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of rounds	10	12	14
Round key size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (words/bytes)	44/176	52/208	60/240

AES can be used with a variable block and key length; there are 56 bits for DES which will increase the computational power and are easy to break (NIST 2001). Refer to the **Table 6**, if a 128 bits key is chosen, the message M is divided into several blocks m_1, $m_2 \ldots m_n$. A simple bitwise XOR is applied to each byte of the block and the portion of the expanded key is processed in 10 rounds using following operations as well the **Figure 20**.

Substitution: Each byte is replaced with another one according to a 256-byte look-up table called the S-box.

Permutation: Cyclically shifting of lines in state array. The bytes in each of the 4 rows in the state are rotated by (n-1) where n represents the row number from 1 to 4.

Diffusion: Performing matrix multiplication, each byte of a column with every other byte. The state can be considered to be a 4*4 matrix and this transformation can be achieved by multiplying this matrix by:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

In the last round this step must be omitted.

Key Generation: Performing XOR operation. In this transformation, the round key is simply added to the state, which is done by GF (2^8).

The decryption of AES uses the inverse function of encryption and the same key-schedule for the round keys, which need longer processing time due to high complexity.

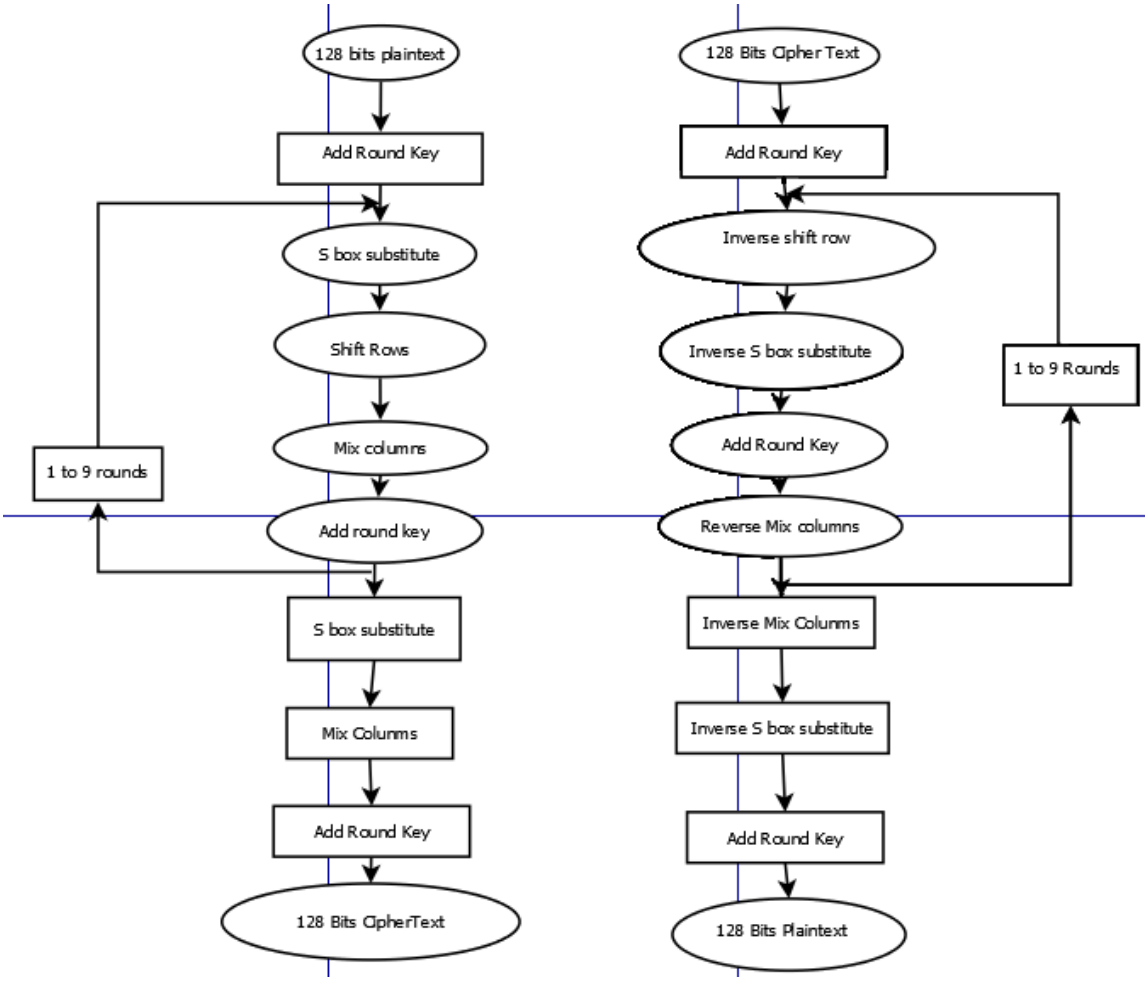

Figure 20. The procedure of AES encryption and decryption.

AES works with three key lengths, thus three different versions of the encryption and decryption scheme have been prepared. The key sizes used for an AES cipher specified the number of repetitions of transformation rounds. The numbers of repetitions of transformation rounds are as follows:

- 10 rounds for 128-bit key;
- 12 rounds for 192-bit key;

- 14 rounds for 256-bit key.

5.4. Result of the implementation

A. CAESAR

After programming the STK 500 with ATMEL studio, Realterm is used for the communication between PC and microcontroller.

A string is sent from the Realterm application (running on the PC) to the microcontroller, where the string will be encrypted. The encrypted string is sent back to the Realterm application, after that the microcontroller decrypts the string again and sends it to the Realterm application.

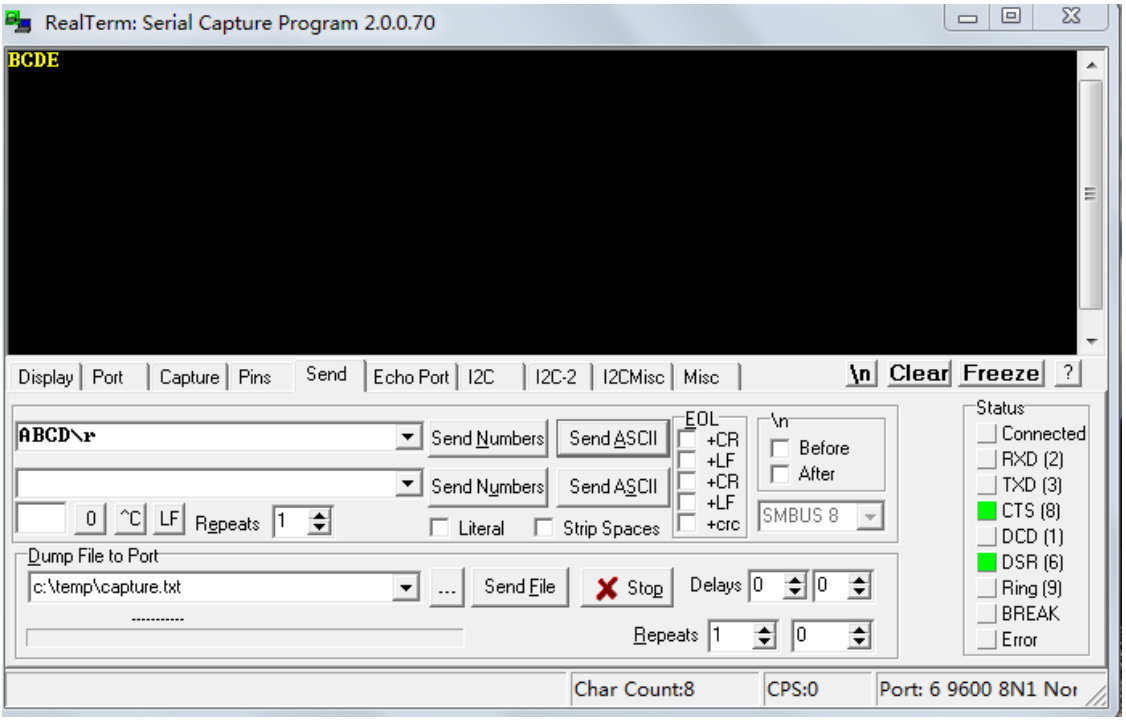

Figure 21. CAESAR algorithm encryption.

Plain message is ABCD and cipher message is BCDE, see **Figure 21**.

The decryption is quite simple here; just reverse the key shift. The encrypted key is 1, so the decrypted key is -1. **Figure 22** illustrates the decryption result.

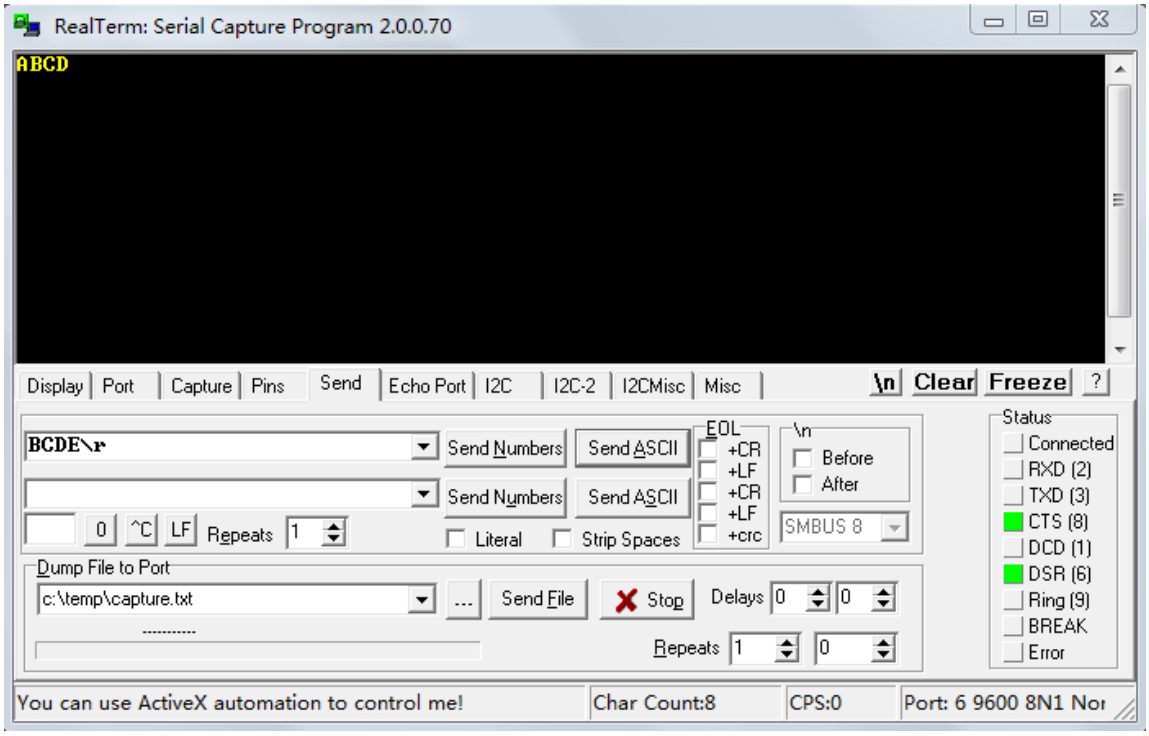

Figure 22. CAESAR algorithm decryption.

B. AES

The result of AES encryption and decryption is shown in **Figure 23**.The block message is 128 bits. AES has 10 rounds for a key length 128 bits; 12 rounds for a 192-bit key and 14 rounds for a 256-bit key. In this experiment, the time and current consumption for the encryption and decryption is measured.

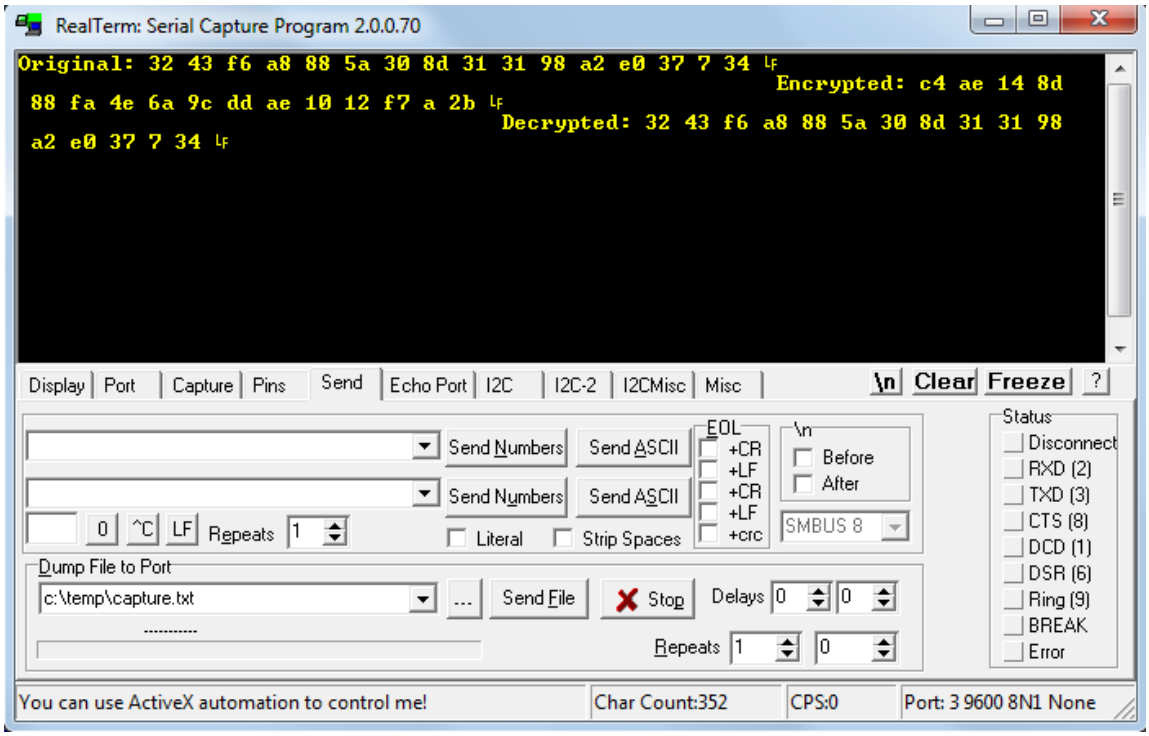

Figure 23. AES Encryption and Decryption result.

5.5 Time Consumption of Different Key Length

A. CAESAR encryption and decryption

An oscilloscope is used for measuring the times. Before and after the encryption/ decryption a selected pin is toggled. Then the encryption time with different number of bytes and different frequencies is measured. **Figure 24** is the screenshot of the CAESAR encryption time measuring result for 50bytes and a beginning of 8 MHz on oscilloscope.

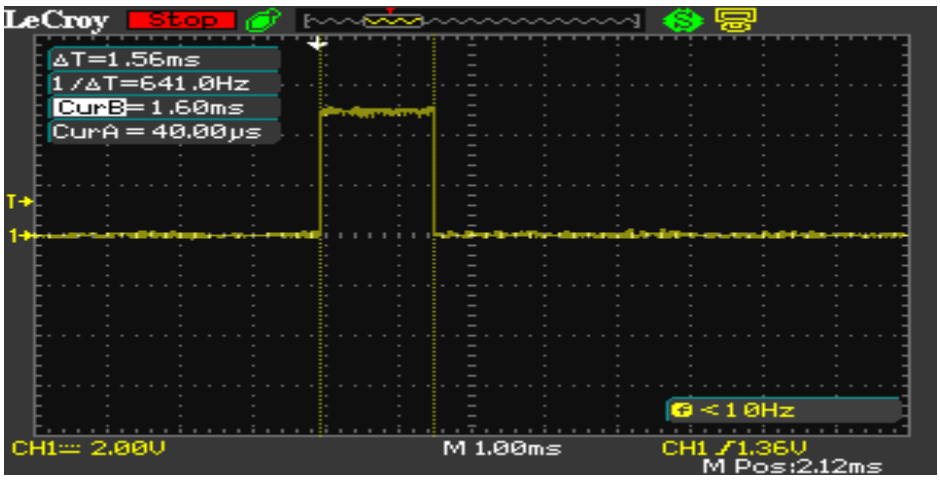

Figure 24. The screenshot of CAESAR encryption time at 8 MHz and 50bytes message on oscilloscope.

The result of time consumption for CAESAR encryption with different bytes and different frequencies is shown in **Table 7**. For CAESAR, decryption is just the reverse procedure of encryption, so the time consumption is just the same as encryption.

Table 7. The result of CAESAR encryption time of different times with different frequencies and messages bytes

Frequency / **NO. of bytes**	1MHz	4MHz	8MHZ
50	12.80 ms	3.12ms	1.56ms
100	25.40ms	6.20ms	3.12ms
150	38.00ms	9.30ms	4.72ms
200	50.08ms	12.30ms	6.28ms

The relationship between the encryption time and the frequency is linear; as well it is linear with the plaintext bytes. The bigger the frequency, the shorter the time consumption is; the more the number of bytes, the higher the time consumption. **Figure 25** represents those relationships.

Generally, decryption time is just the same as encryption in CAESAR.

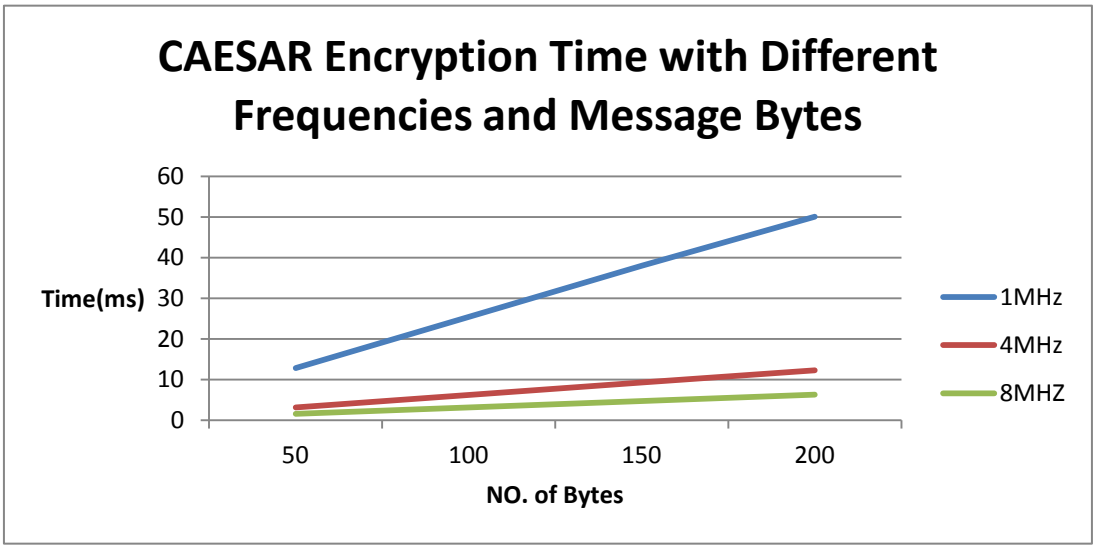

Figure 25. The relationship among CAESAR, frequency and message bytes.

B. AES

In this section, the encryption and decryption times of AES are measured and compared by using different key lengths at different frequencies. An oscilloscope is used for measuring the times. Before and after the encryption/ decryption a selected pin is toggled. **Figure 26** is the screenshot of the AES encryption time measuring result for 50 bytes and a frequency of 8 MHz on oscilloscope.

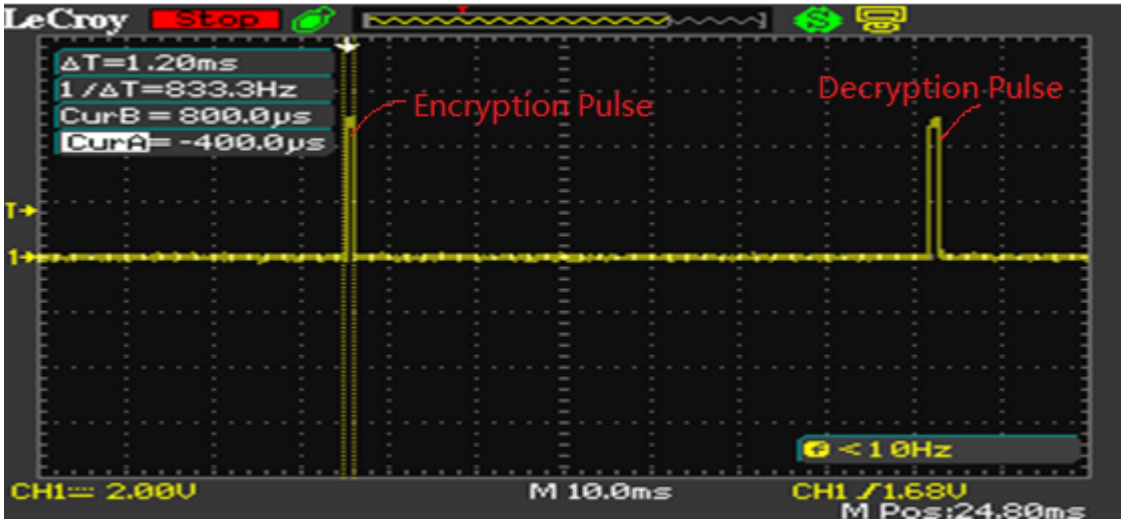

Figure 26. The **s**creenshot of the AES encryption time at 8MHz 128 bit key on oscilloscope.

The result of time consumption for AES encryption with different key lengths and different frequencies is shown is the **Table 8**.

Table 8. Results of AES Encryption time consumption on Atmel STK 500 with different key lengths and different frequencies.

		Frequency		
Key Length	**Rounds**	**1MHz**	**4MHz**	**8MHz**
128 bit	10	8.80ms	2.20ms	1.20ms
192 bit	12	10.60ms	2.78ms	1.40ms
256 bit	14	13.40ms	3.24ms	1.66ms

The relationship of AES encryption time consumption with different key lengths and different frequencies is shown as a bar chart in **Figure 27**.

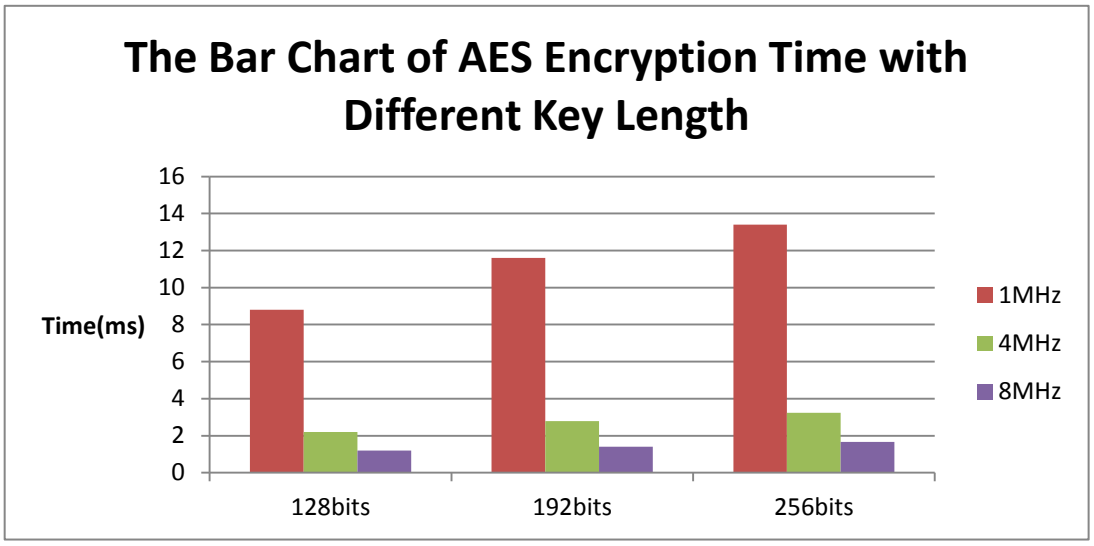

Figure 27. The bar chart of the relationship among AES encryption time consumption, frequency and key lengths.

The comparison to the encryption time obtain in **Table 8**, the decryption time is a little bit longer than the encryption (see **Table 9**).

Table 9. Results of the AES Decryption time consumption on Atmel STK 500 with different key lengths and different frequencies.

		Frequency		
Key Length	**Rounds**	**1MHz**	**4MHz**	**8MHz**
128 bit	10	9.60ms	2.50ms	1.24ms
192 bit	12	11.40ms	2.90ms	1.48ms
256 bit	14	13.60ms	3.30ms	1.72ms

The relationship of AES encryption time consumption with different key lengths and different frequencies is shown as a bar chart in **Figure 28**.

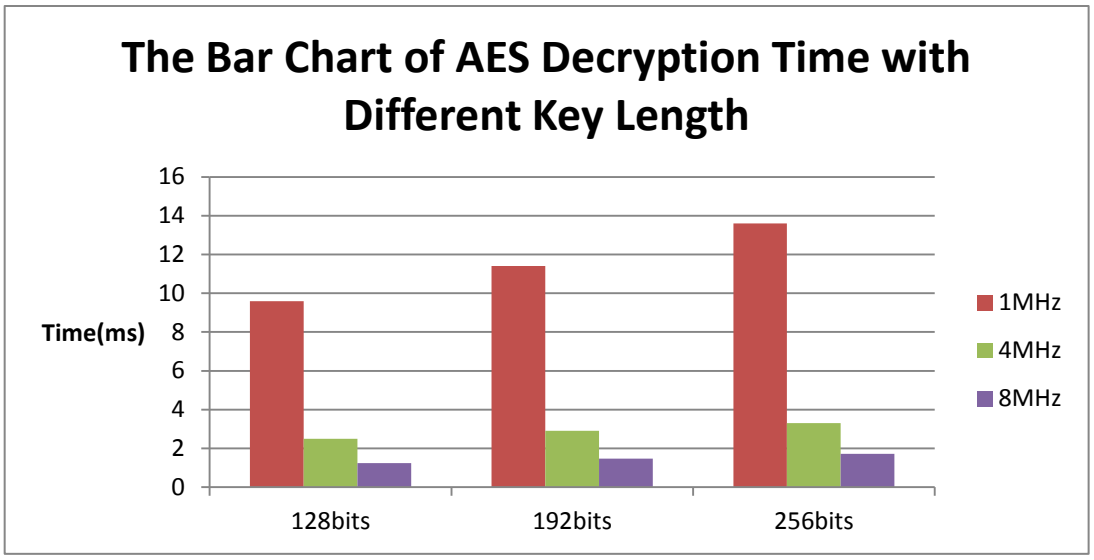

Figure 28. The bar chart of the relationship among AES decryption time consumption, frequency and key lengths.

5.6 Power Consumption of Different Frequencies

A multimeter is used for measuring the current. Idle current is measured when the circuit is idle; maximum current is measured when the circuit is being operated under encryption and decryption process. The circuit model is illustrated in **Figure 29**. The input voltage is 12 Volts. Then the power consumption can be calculated by formula Power= current *voltage.

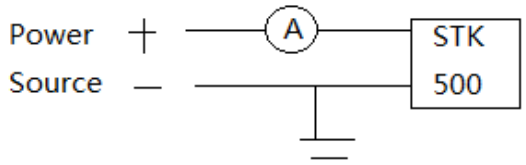

Figure 29. Circuit Model.

The current consumption of CAESAR is measured, and the result is listed in **Table 10.**

Table 10. Results of CAESAR idle and max current measurement

Frequency	1 MHz	4MHz	8MHz
Idle Current	102.5mA	107.2mA	111.9mA
Max Current	110mA	114.5mA	118.6mA

The relationship of CAESAR encryption current consumption with different frequencies is shown as a bar chart in **Figure 30**.

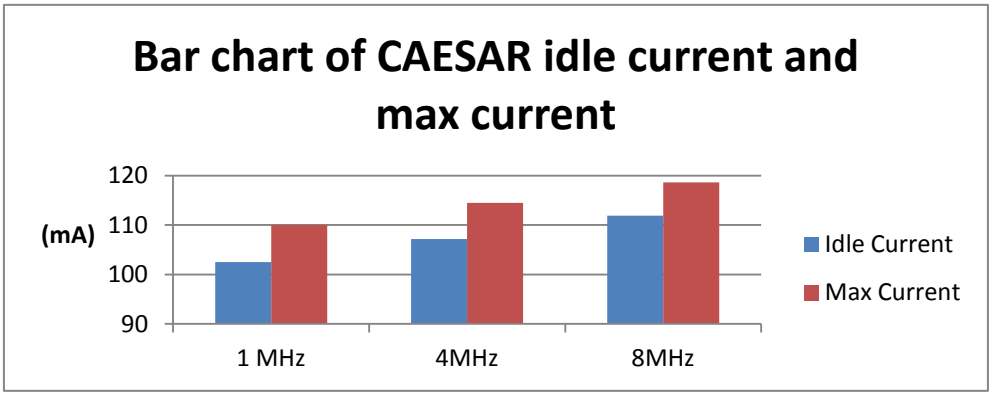

Figure 30. The bar chart shows the relationship between CAESAR encryption current consumption and frequency.

The maximum power consumption at 1MHz is calculated as: P=V*I= 12Volts * 110mA= 1.32 Watt. Then the power/time =1.32watt / 12.8 ms= 103.125 watt/s. When Frequency at 4MHz, P/T = 440.38 watt/s. More power consumption results in less time consumption. This principle is the same for AES cryptography.

The current consumption of AES is measured, and the result is listed in **Table 11.**

Table 11. Results of AES idle and max current measurement.

Frequency	1 MHz	4MHz	8MHz
Idle Current	99.9 mA	103.8mA	106.7mA
Max Current	101.7mA	106.6mA	110.3mA

The relationship of the CAESAR encryption current consumption with different frequencies is shown as a bar chart in **Figure 31**.

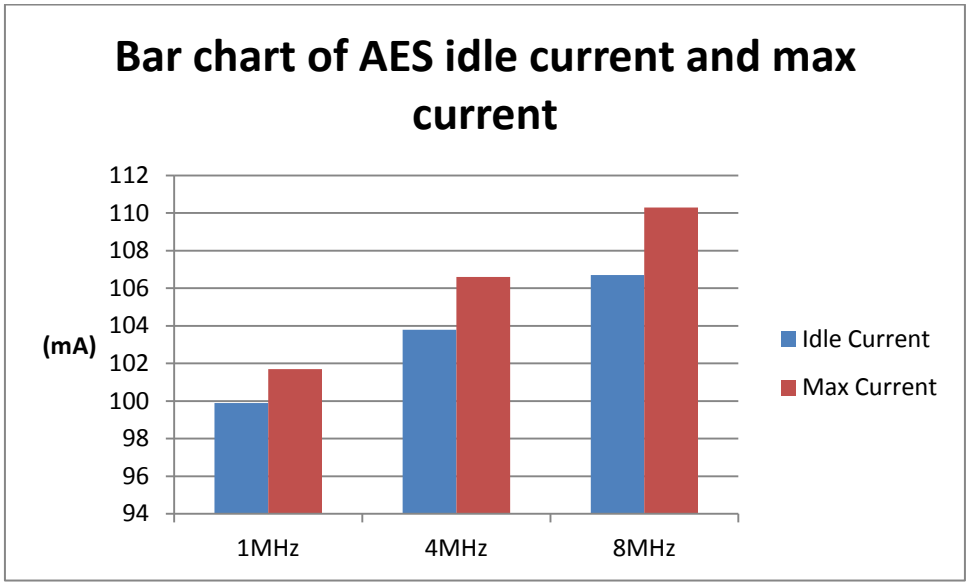

Figure 31. The bar chart of the relationship between AES encryption current consumption and frequency.

The result shows that the higher the frequency, the higher the current consumption. Here voltage is constant; the power consumption is increasing with the rise of the frequency.

6. CONCLUSION AND FUTURE WORK

In this thesis, symmetric cryptography and asymmetric cryptography algorithms were analyzed and researched in the theoretical section. Different algorithms were compared by the key lengths, the length of encryption time and decryption time, as well the security of itself. In the practical section, CAESAR and AES were coded by C language and programmed on an embedded system (Atmel STK 500 board). Time consumption and power consumption of each algorithm were measured. The higher the frequency is, the less the time consumption is. Bigger messages or longer key lengths lead to more time consumptions. Regarding to the power consumption, a higher frequency results in more power consumption. After analyzing those results, it is obviously shown that power consumption decreases when time consumption increases.

ECC is a new secure innovation in the information security field that can be adapted in the future telecommunication and embedded system area. Absolute advantages in computing speed and storage space, it is a research hot spot in current public cryptography systems. As a matter of factor, there are still rooms for improvement on the implementation of ECC with software.

There are still many problems in the research field of ECC, which become the bottleneck of its development and application, such as embedded plaintext algorithm, curve and basis of calculation and the selection of safety curve algorithm. The calculation of ECDLP is the core research of the elliptic curve. Future works on ECC are based on three aspects: how to select the high security level Elliptic Curve, which means the

selected algorithm is easy to be applied and hard to brake. In the Elliptic Curve Cryptosystem, the times of points on the elliptic curve group take up a large proportion of the whole operation. The efficiency is related to the execution of the whole procedure. As far as the application and development of Smart Card and wireless communication field are concerned, what is significant to be researched is how to enhance the defense capability of the chip itself.

REFERENCE

Announcing the Advanced Encryption Standard (AES). Available from the Internet:

< www. Nist.gov>

Avanzi R.M (2005). Side Channel Attacks on Implementations of Curve-Based Cryptographic Primitives (preprint),eprint.

ATMEL, AVR STK500 User Guide

Bailey D & Paar C(1998). Optimal Extension Field for Fast Arithmetic in Public-Key Algorithms , CRYPTO'98, lNCS 1462,pp.472-485.

Bailey D & Paar C (2001). Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography, Journal of Cryptography, Vol 14, pp.153-176

Biryukov, A & D. Khovratovich (2009). "Distinguisher and Realated-Key Attack on the Full AES-256."CRYPTO'09.

Connected: An Internet Encyclopedia."Block Ciphers". April 1997. Available from the Internet: <URL: http://www.freesoft.org/CIE/Topics/143.htm>

I.A. Semaev (1998). Evaluation of discrete logarithms on some elliptic curves. ***Math. Comp***., **67**, 353-356.

Ian B., Gadiel S. and Nigel S (1999). *Elliptic Curves in Cryptography.* ISBN 0-521-65374-6

Ian McCombe April 04, 2007. Available from the Internet:

<URL: http://imps.mcmaster.ca/courses/SE-4C0307/wiki/mccombi/blockciphers.html>.

Information Security. Available from the Internet:

<URL:http://www.javvin.com/networksecurity/dictionary.html>

J.B.Lacy, D.P.Mitchell & W.M.Schell (1984). "CryptoLib: Cryptography in Software," UNIX Security Symposium IV Proceeddings of Crypto 83, Plenum Press, pp.3-23.

Jean-Sebastien Coron. *Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems.*

J.H. Silverman (1986). *The Arithmetic of Elliptic Curves.* Springer-Verlag, GTM 106.

Martti Penttonen (2009). Date Security [online] [cited 13 March 2009]. Available from the Internet: <URL: www.cs.uku.fi/~penttion/secu/>.

N.P.Smart (1998). ***The Algorithmic Resolution of Diophamtine Equations.*** Cambridge University Press.

Ondrej H, Pavel K, Petr H & Petr F (2011). Performance Evaluation of Symmetric Cryptography in Embedded Systems.

Ptiotr B, Wiealaw W & Tomasz A. Implementation of symmetric Cryptography in Embedded Systems for Secure Measurement Systems.

Schneier Bruce (1996). APPLIED CRYPTOGRAPHY. *Protocols, Algorithms, and Source Code in C*. ISBN:0-471-12845-7.

Shammi D., Aaron A & Saurabh B. Optimizing AES for Embedded Devices and Wireless Sensor Network.

S. Chari, C. Jutla, J.R. Rao & P. Rohatgi (1999). A cautionary note regarding evaluation of AES candidates on smart-cards, Proceedings of the second AES Candidate Conference, March, pp. 133-147.

Smart N (1999). The Discrete Logarithm Problem on Elliptic Curves of Trace One. Journal of Cryptography, Vol.12,pp.193-196.

Satoh T. & Araki K (1998). Fermat Quotient and The Polynimial Time Discrete Log Algorithm for Anomalous Elliptic Curves, Commentarii Mathematici Universitatis Sancti Pauli, Vol.47, pp.81-92.

The AES Cipher. Available from the Internet: < http://flylib.com/books/en/3.190.1.55/1/>

Thomas W, Jorge G & Christof P (2003). *Cryptography in Embedded Systems: An Overview*. Pp.735-744, Design & Elektronik, Nuernberg, Germany, Feb. 18-20.

Wang L, Zhao H & Bai Gq. *A Cost-Efficient Implementation of Public-KEY Cryptography on Embedded Systems.*

Wang Qingxian, The application of Elliptic Curves Cryptography in Embedded Systems.

Welschenbach M (2001). *Cryptography in C and C++*. ISBN: 1-893115-95-X.

Wikipedia (2013a). CAESAR cipher. Available from the Internet: <http://en.wikipedia.org/wiki/Caesar_cipher>.

Wikipedia (2012a) . Advanced Encryption Standard. Available from the Internet: <http://en.wikipedia.org/wiki/AES>.

AES 算法自主学习报告. Available from the Internet:<http://wenku.baidu.com/view/e6b01b8671fe910ef12df8d8.html?from=related&hasrec=1>.

杨新国, 基于 AES 的加密技术研究及应用. Available from the Internet: <http://www.doc88.com/p-735479808402.html>.

APPENDIXES

APPENDIX 1. S-box(William Stallings 2011: 181)

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

Inverse S-box

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

APPENDIX 2. CAESAR Encryption

```
#####################################################################
###
### CAESAR encryption code
###
### Copyright (C) 2013, Qian Yang (t94781@student.uwasa.fi)
###
### University of Vaasa
###
#####################################################################
```

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <avr/interrupt.h>

#define ARRAY_SIZE 220

static volatile uint8_t count;

static char buffer[ARRAY_SIZE];

void caesar(char *, const int);

// ISR for USART receive

ISR(USART_RX_vect)

{

        buffer[count] = UDR;

        if(buffer[count] == '\r' || count == (ARRAY_SIZE -1))
```

```
        {
                buffer[count] == '\0';
                // set PB0 to 1
                PORTB |= (1 << PB0);

                caesar(buffer, 10);
                // set PB0 to 0
                PORTB &= ~(1 << PB0);
                int i;
                for(i = 0; i < count; i++)
                {
                        usart_putc(buffer[i]);
                        buffer[i] = '\0';
                }
                count = 0;
        }
        else
                count++;
}
int main(void)
```

```
{
        // define PB0 of PORTB as output
        DDRB |= (1 << PB0);
        // set PB0 to 0
        PORTB &= ~(1 << PB0);
        // initialize USART
        //UBRRL = 103; // 8MHz Baudrate 9600
        //UBRRL = 51; // 4MHz Baudrate 9600
        UBRRL = 12; // 1MHz Baudrate 9600
        UCSRA |= (1 << U2X);
        UCSRB |= (1 << RXCIE) | (1 << RXEN) | (1 << TXEN);
        // initialize count variable
        count = 0;
        // enable interrupt
        sei();
        while(1);
        return 0;
}
inline void caesar(char *str, const int offset)
{
```

```
        for(;*str!='\0';str++)
        {
                if(*str>='A' && *str<='Z')
                *str = 'A' + (*str - 'A' + offset) % 26;
                else if(*str>='a' && *str<='z')
                *str = 'a' + (*str - 'a' + offset) % 26;
        }
}
void usart_putc(unsigned char c)
{
        // wait for an empty transmit buffer
        // UDRE = USART Data Register Empty
        // if UDRE = 1 the buffer is empty
        while(!(UCSRA & (1 << UDRE)));
        // USART I/O Data Register
        UDR = c;
}
```

APPENDIX 3. AES Encryption and Decryption

```
################################################################
### AES Encryption and Decryption code
###
### Original author: Karl Malbrain, malbrain@yahoo.com
### Modified and ported to Atmel AVR by: Yang Qian
### (t94781@student.uwasa.fi)
### University of Vaasa
################################################################
```

```
// AES only supports Nb=4
#define Nb 4
// The number of columns comprising a state in AES. This is a constant in AES.
Value=4
#define Nk 4                    // number of columns in a key
#define Nr 10                   // number of rounds in encryption

#define Sbox(i) (pgm_read_byte(&P_Sbox[i]))
const unsigned char P_Sbox[256] __attribute__ ((__progmem__)) = {
// forward s-box
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7,
0xab, 0x76,
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4,
0x72, 0xc0,
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8,
0x31, 0x15,
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27,
0xb2, 0x75,
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
0x2f, 0x84,
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c,
0x58, 0xcf,
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
```

```
0x9f, 0xa8,
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff,
0xf3, 0xd2,
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d,
0x19, 0x73,
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e,
0x0b, 0xdb,
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95,
0xe4, 0x79,
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a,
0xae, 0x08,
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd,
0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1,
0x1d, 0x9e,
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,
0x28, 0xdf,
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,
0xbb, 0x16};

#define InvSbox(i) (pgm_read_byte(&P_InvSbox[i]))
const unsigned char P_InvSbox[256] __attribute__ ((__progmem__)) = {   // inverse s-
box
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3,
0xd7, 0xfb,
0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde,
0xe9, 0xcb,
0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa,
0xc3, 0x4e,
0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b,
0xd1, 0x25,
0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65,
0xb6, 0x92,
```

```
0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d,
0x9d, 0x84,
0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3,
0x45, 0x06,
0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13,
0x8a, 0x6b,
0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4,
0xe6, 0x73,
0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75,
0xdf, 0x6e,
0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18,
0xbe, 0x1b,
0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd,
0x5a, 0xf4,
0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80,
0xec, 0x5f,
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9,
0x9c, 0xef,
0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53,
0x99, 0x61,
0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21,
0x0c, 0x7d};

// combined Xtimes2[Sbox[]]
#define Xtime2Sbox(i) (pgm_read_byte(&P_Xtime2Sbox[i]))
const unsigned char P_Xtime2Sbox[256] __attribute__ ((__progmem__)) = {
0xc6, 0xf8, 0xee, 0xf6, 0xff, 0xd6, 0xde, 0x91, 0x60, 0x02, 0xce, 0x56, 0xe7, 0xb5,
0x4d, 0xec,
0x8f, 0x1f, 0x89, 0xfa, 0xef, 0xb2, 0x8e, 0xfb, 0x41, 0xb3, 0x5f, 0x45, 0x23, 0x53,
0xe4, 0x9b,
0x75, 0xe1, 0x3d, 0x4c, 0x6c, 0x7e, 0xf5, 0x83, 0x68, 0x51, 0xd1, 0xf9, 0xe2, 0xab,
0x62, 0x2a,
0x08, 0x95, 0x46, 0x9d, 0x30, 0x37, 0x0a, 0x2f, 0x0e, 0x24, 0x1b, 0xdf, 0xcd, 0x4e,
```

```
0x7f, 0xea,
0x12, 0x1d, 0x58, 0x34, 0x36, 0xdc, 0xb4, 0x5b, 0xa4, 0x76, 0xb7, 0x7d, 0x52, 0xdd,
0x5e, 0x13,
0xa6, 0xb9, 0x00, 0xc1, 0x40, 0xe3, 0x79, 0xb6, 0xd4, 0x8d, 0x67, 0x72, 0x94, 0x98,
0xb0, 0x85,
0xbb, 0xc5, 0x4f, 0xed, 0x86, 0x9a, 0x66, 0x11, 0x8a, 0xe9, 0x04, 0xfe, 0xa0, 0x78,
0x25, 0x4b,
0xa2, 0x5d, 0x80, 0x05, 0x3f, 0x21, 0x70, 0xf1, 0x63, 0x77, 0xaf, 0x42, 0x20, 0xe5,
0xfd, 0xbf,
0x81, 0x18, 0x26, 0xc3, 0xbe, 0x35, 0x88, 0x2e, 0x93, 0x55, 0xfc, 0x7a, 0xc8, 0xba,
0x32, 0xe6,
0xc0, 0x19, 0x9e, 0xa3, 0x44, 0x54, 0x3b, 0x0b, 0x8c, 0xc7, 0x6b, 0x28, 0xa7, 0xbc,
0x16, 0xad,
0xdb, 0x64, 0x74, 0x14, 0x92, 0x0c, 0x48, 0xb8, 0x9f, 0xbd, 0x43, 0xc4, 0x39, 0x31,
0xd3, 0xf2,
0xd5, 0x8b, 0x6e, 0xda, 0x01, 0xb1, 0x9c, 0x49, 0xd8, 0xac, 0xf3, 0xcf, 0xca, 0xf4,
0x47, 0x10,
0x6f, 0xf0, 0x4a, 0x5c, 0x38, 0x57, 0x73, 0x97, 0xcb, 0xa1, 0xe8, 0x3e, 0x96, 0x61,
0x0d, 0x0f,
0xe0, 0x7c, 0x71, 0xcc, 0x90, 0x06, 0xf7, 0x1c, 0xc2, 0x6a, 0xae, 0x69, 0x17, 0x99,
0x3a, 0x27,
0xd9, 0xeb, 0x2b, 0x22, 0xd2, 0xa9, 0x07, 0x33, 0x2d, 0x3c, 0x15, 0xc9, 0x87, 0xaa,
0x50, 0xa5,
0x03, 0x59, 0x09, 0x1a, 0x65, 0xd7, 0x84, 0xd0, 0x82, 0x29, 0x5a, 0x1e, 0x7b, 0xa8,
0x6d, 0x2c
};

// combined Xtimes3[Sbox[]]
#define Xtime3Sbox(i) (pgm_read_byte(&P_Xtime3Sbox[i]))
const unsigned char P_Xtime3Sbox[256] __attribute__ ((__progmem__)) = {
0xa5, 0x84, 0x99, 0x8d, 0x0d, 0xbd, 0xb1, 0x54, 0x50, 0x03, 0xa9, 0x7d, 0x19, 0x62,
0xe6, 0x9a,
0x45, 0x9d, 0x40, 0x87, 0x15, 0xeb, 0xc9, 0x0b, 0xec, 0x67, 0xfd, 0xea, 0xbf, 0xf7,
```

```
0x96, 0x5b,
0xc2, 0x1c, 0xae, 0x6a, 0x5a, 0x41, 0x02, 0x4f, 0x5c, 0xf4, 0x34, 0x08, 0x93, 0x73,
0x53, 0x3f,
0x0c, 0x52, 0x65, 0x5e, 0x28, 0xa1, 0x0f, 0xb5, 0x09, 0x36, 0x9b, 0x3d, 0x26, 0x69,
0xcd, 0x9f,
0x1b, 0x9e, 0x74, 0x2e, 0x2d, 0xb2, 0xee, 0xfb, 0xf6, 0x4d, 0x61, 0xce, 0x7b, 0x3e,
0x71, 0x97,
0xf5, 0x68, 0x00, 0x2c, 0x60, 0x1f, 0xc8, 0xed, 0xbe, 0x46, 0xd9, 0x4b, 0xde, 0xd4,
0xe8, 0x4a,
0x6b, 0x2a, 0xe5, 0x16, 0xc5, 0xd7, 0x55, 0x94, 0xcf, 0x10, 0x06, 0x81, 0xf0, 0x44,
0xba, 0xe3,
0xf3, 0xfe, 0xc0, 0x8a, 0xad, 0xbc, 0x48, 0x04, 0xdf, 0xc1, 0x75, 0x63, 0x30, 0x1a,
0x0e, 0x6d,
0x4c, 0x14, 0x35, 0x2f, 0xe1, 0xa2, 0xcc, 0x39, 0x57, 0xf2, 0x82, 0x47, 0xac, 0xe7,
0x2b, 0x95,
0xa0, 0x98, 0xd1, 0x7f, 0x66, 0x7e, 0xab, 0x83, 0xca, 0x29, 0xd3, 0x3c, 0x79, 0xe2,
0x1d, 0x76,
0x3b, 0x56, 0x4e, 0x1e, 0xdb, 0x0a, 0x6c, 0xe4, 0x5d, 0x6e, 0xef, 0xa6, 0xa8, 0xa4,
0x37, 0x8b,
0x32, 0x43, 0x59, 0xb7, 0x8c, 0x64, 0xd2, 0xe0, 0xb4, 0xfa, 0x07, 0x25, 0xaf, 0x8e,
0xe9, 0x18,
0xd5, 0x88, 0x6f, 0x72, 0x24, 0xf1, 0xc7, 0x51, 0x23, 0x7c, 0x9c, 0x21, 0xdd, 0xdc,
0x86, 0x85,
0x90, 0x42, 0xc4, 0xaa, 0xd8, 0x05, 0x01, 0x12, 0xa3, 0x5f, 0xf9, 0xd0, 0x91, 0x58,
0x27, 0xb9,
0x38, 0x13, 0xb3, 0x33, 0xbb, 0x70, 0x89, 0xa7, 0xb6, 0x22, 0x92, 0x20, 0x49, 0xff,
0x78, 0x7a,
0x8f, 0xf8, 0x80, 0x17, 0xda, 0x31, 0xc6, 0xb8, 0xc3, 0xb0, 0x77, 0x11, 0xcb, 0xfc,
0xd6, 0x3a
};

// modular multiplication tables
// based on:
```

```
// Xtime2[x] = (x & 0x80 ? 0x1b : 0) ^ (x + x)
// Xtime3[x] = x^Xtime2[x];

#define Xtime2(i) (pgm_read_byte(&P_Xtime2[i]))
const unsigned char P_Xtime2[256] __attribute__ ((__progmem__)) = {
0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0x10, 0x12, 0x14, 0x16, 0x18, 0x1a,
0x1c, 0x1e,
0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0x30, 0x32, 0x34, 0x36, 0x38, 0x3a,
0x3c, 0x3e,
0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0x50, 0x52, 0x54, 0x56, 0x58, 0x5a,
0x5c, 0x5e,
0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0x70, 0x72, 0x74, 0x76, 0x78, 0x7a,
0x7c, 0x7e,
0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0x90, 0x92, 0x94, 0x96, 0x98, 0x9a,
0x9c, 0x9e,
0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0xb0, 0xb2, 0xb4, 0xb6, 0xb8, 0xba,
0xbc, 0xbe,
0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0xd0, 0xd2, 0xd4, 0xd6, 0xd8, 0xda,
0xdc, 0xde,
0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0xf0, 0xf2, 0xf4, 0xf6, 0xf8, 0xfa, 0xfc,
0xfe,
0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15, 0x0b, 0x09, 0x0f, 0x0d, 0x03, 0x01,
0x07, 0x05,
0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35, 0x2b, 0x29, 0x2f, 0x2d, 0x23, 0x21,
0x27, 0x25,
0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55, 0x4b, 0x49, 0x4f, 0x4d, 0x43, 0x41,
0x47, 0x45,
0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75, 0x6b, 0x69, 0x6f, 0x6d, 0x63, 0x61,
0x67, 0x65,
0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95, 0x8b, 0x89, 0x8f, 0x8d, 0x83, 0x81,
0x87, 0x85,
0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5, 0xab, 0xa9, 0xaf, 0xad, 0xa3, 0xa1,
```

```
0xa7, 0xa5,
0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5, 0xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1,
0xc7, 0xc5,
0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5, 0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7,
0xe5};

#define Xtime9(i) (pgm_read_byte(&P_Xtime9[i]))
const unsigned char P_Xtime9[256] __attribute__ ((__progmem__)) = {
0x00, 0x09, 0x12, 0x1b, 0x24, 0x2d, 0x36, 0x3f, 0x48, 0x41, 0x5a, 0x53, 0x6c, 0x65,
0x7e, 0x77,
0x90, 0x99, 0x82, 0x8b, 0xb4, 0xbd, 0xa6, 0xaf, 0xd8, 0xd1, 0xca, 0xc3, 0xfc, 0xf5,
0xee, 0xe7,
0x3b, 0x32, 0x29, 0x20, 0x1f, 0x16, 0x0d, 0x04, 0x73, 0x7a, 0x61, 0x68, 0x57, 0x5e,
0x45, 0x4c,
0xab, 0xa2, 0xb9, 0xb0, 0x8f, 0x86, 0x9d, 0x94, 0xe3, 0xea, 0xf1, 0xf8, 0xc7, 0xce,
0xd5, 0xdc,
0x76, 0x7f, 0x64, 0x6d, 0x52, 0x5b, 0x40, 0x49, 0x3e, 0x37, 0x2c, 0x25, 0x1a, 0x13,
0x08, 0x01,
0xe6, 0xef, 0xf4, 0xfd, 0xc2, 0xcb, 0xd0, 0xd9, 0xae, 0xa7, 0xbc, 0xb5, 0x8a, 0x83,
0x98, 0x91,
0x4d, 0x44, 0x5f, 0x56, 0x69, 0x60, 0x7b, 0x72, 0x05, 0x0c, 0x17, 0x1e, 0x21, 0x28,
0x33, 0x3a,
0xdd, 0xd4, 0xcf, 0xc6, 0xf9, 0xf0, 0xeb, 0xe2, 0x95, 0x9c, 0x87, 0x8e, 0xb1, 0xb8,
0xa3, 0xaa,
0xec, 0xe5, 0xfe, 0xf7, 0xc8, 0xc1, 0xda, 0xd3, 0xa4, 0xad, 0xb6, 0xbf, 0x80, 0x89,
0x92, 0x9b,
0x7c, 0x75, 0x6e, 0x67, 0x58, 0x51, 0x4a, 0x43, 0x34, 0x3d, 0x26, 0x2f, 0x10, 0x19,
0x02, 0x0b,
0xd7, 0xde, 0xc5, 0xcc, 0xf3, 0xfa, 0xe1, 0xe8, 0x9f, 0x96, 0x8d, 0x84, 0xbb, 0xb2,
0xa9, 0xa0,
0x47, 0x4e, 0x55, 0x5c, 0x63, 0x6a, 0x71, 0x78, 0x0f, 0x06, 0x1d, 0x14, 0x2b, 0x22,
0x39, 0x30,
0x9a, 0x93, 0x88, 0x81, 0xbe, 0xb7, 0xac, 0xa5, 0xd2, 0xdb, 0xc0, 0xc9, 0xf6, 0xff,
```

```
0xe4, 0xed,
0x0a, 0x03, 0x18, 0x11, 0x2e, 0x27, 0x3c, 0x35, 0x42, 0x4b, 0x50, 0x59, 0x66, 0x6f,
0x74, 0x7d,
0xa1, 0xa8, 0xb3, 0xba, 0x85, 0x8c, 0x97, 0x9e, 0xe9, 0xe0, 0xfb, 0xf2, 0xcd, 0xc4,
0xdf, 0xd6,
0x31, 0x38, 0x23, 0x2a, 0x15, 0x1c, 0x07, 0x0e, 0x79, 0x70, 0x6b, 0x62, 0x5d, 0x54,
0x4f, 0x46};

#define XtimeB(i) (pgm_read_byte(&P_XtimeB[i]))
const unsigned char P_XtimeB[256] __attribute__ ((__progmem__)) = {
0x00, 0x0b, 0x16, 0x1d, 0x2c, 0x27, 0x3a, 0x31, 0x58, 0x53, 0x4e, 0x45, 0x74, 0x7f,
0x62, 0x69,
0xb0, 0xbb, 0xa6, 0xad, 0x9c, 0x97, 0x8a, 0x81, 0xe8, 0xe3, 0xfe, 0xf5, 0xc4, 0xcf,
0xd2, 0xd9,
0x7b, 0x70, 0x6d, 0x66, 0x57, 0x5c, 0x41, 0x4a, 0x23, 0x28, 0x35, 0x3e, 0x0f, 0x04,
0x19, 0x12,
0xcb, 0xc0, 0xdd, 0xd6, 0xe7, 0xec, 0xf1, 0xfa, 0x93, 0x98, 0x85, 0x8e, 0xbf, 0xb4,
0xa9, 0xa2,
0xf6, 0xfd, 0xe0, 0xeb, 0xda, 0xd1, 0xcc, 0xc7, 0xae, 0xa5, 0xb8, 0xb3, 0x82, 0x89,
0x94, 0x9f,
0x46, 0x4d, 0x50, 0x5b, 0x6a, 0x61, 0x7c, 0x77, 0x1e, 0x15, 0x08, 0x03, 0x32, 0x39,
0x24, 0x2f,
0x8d, 0x86, 0x9b, 0x90, 0xa1, 0xaa, 0xb7, 0xbc, 0xd5, 0xde, 0xc3, 0xc8, 0xf9, 0xf2,
0xef, 0xe4,
0x3d, 0x36, 0x2b, 0x20, 0x11, 0x1a, 0x07, 0x0c, 0x65, 0x6e, 0x73, 0x78, 0x49, 0x42,
0x5f, 0x54,
0xf7, 0xfc, 0xe1, 0xea, 0xdb, 0xd0, 0xcd, 0xc6, 0xaf, 0xa4, 0xb9, 0xb2, 0x83, 0x88,
0x95, 0x9e,
0x47, 0x4c, 0x51, 0x5a, 0x6b, 0x60, 0x7d, 0x76, 0x1f, 0x14, 0x09, 0x02, 0x33, 0x38,
0x25, 0x2e,
0x8c, 0x87, 0x9a, 0x91, 0xa0, 0xab, 0xb6, 0xbd, 0xd4, 0xdf, 0xc2, 0xc9, 0xf8, 0xf3,
0xee, 0xe5,
0x3c, 0x37, 0x2a, 0x21, 0x10, 0x1b, 0x06, 0x0d, 0x64, 0x6f, 0x72, 0x79, 0x48, 0x43,
```

```
0x5e, 0x55,
0x01, 0x0a, 0x17, 0x1c, 0x2d, 0x26, 0x3b, 0x30, 0x59, 0x52, 0x4f, 0x44, 0x75, 0x7e,
0x63, 0x68,
0xb1, 0xba, 0xa7, 0xac, 0x9d, 0x96, 0x8b, 0x80, 0xe9, 0xe2, 0xff, 0xf4, 0xc5, 0xce,
0xd3, 0xd8,
0x7a, 0x71, 0x6c, 0x67, 0x56, 0x5d, 0x40, 0x4b, 0x22, 0x29, 0x34, 0x3f, 0x0e, 0x05,
0x18, 0x13,
0xca, 0xc1, 0xdc, 0xd7, 0xe6, 0xed, 0xf0, 0xfb, 0x92, 0x99, 0x84, 0x8f, 0xbe, 0xb5,
0xa8, 0xa3};

#define XtimeD(i) (pgm_read_byte(&P_XtimeD[i]))
const unsigned char P_XtimeD[256] __attribute__ ((__progmem__)) = {
0x00, 0x0d, 0x1a, 0x17, 0x34, 0x39, 0x2e, 0x23, 0x68, 0x65, 0x72, 0x7f, 0x5c, 0x51,
0x46, 0x4b,
0xd0, 0xdd, 0xca, 0xc7, 0xe4, 0xe9, 0xfe, 0xf3, 0xb8, 0xb5, 0xa2, 0xaf, 0x8c, 0x81,
0x96, 0x9b,
0xbb, 0xb6, 0xa1, 0xac, 0x8f, 0x82, 0x95, 0x98, 0xd3, 0xde, 0xc9, 0xc4, 0xe7, 0xea,
0xfd, 0xf0,
0x6b, 0x66, 0x71, 0x7c, 0x5f, 0x52, 0x45, 0x48, 0x03, 0x0e, 0x19, 0x14, 0x37, 0x3a,
0x2d, 0x20,
0x6d, 0x60, 0x77, 0x7a, 0x59, 0x54, 0x43, 0x4e, 0x05, 0x08, 0x1f, 0x12, 0x31, 0x3c,
0x2b, 0x26,
0xbd, 0xb0, 0xa7, 0xaa, 0x89, 0x84, 0x93, 0x9e, 0xd5, 0xd8, 0xcf, 0xc2, 0xe1, 0xec,
0xfb, 0xf6,
0xd6, 0xdb, 0xcc, 0xc1, 0xe2, 0xef, 0xf8, 0xf5, 0xbe, 0xb3, 0xa4, 0xa9, 0x8a, 0x87,
0x90, 0x9d,
0x06, 0x0b, 0x1c, 0x11, 0x32, 0x3f, 0x28, 0x25, 0x6e, 0x63, 0x74, 0x79, 0x5a, 0x57,
0x40, 0x4d,
0xda, 0xd7, 0xc0, 0xcd, 0xee, 0xe3, 0xf4, 0xf9, 0xb2, 0xbf, 0xa8, 0xa5, 0x86, 0x8b,
0x9c, 0x91,
0x0a, 0x07, 0x10, 0x1d, 0x3e, 0x33, 0x24, 0x29, 0x62, 0x6f, 0x78, 0x75, 0x56, 0x5b,
0x4c, 0x41,
0x61, 0x6c, 0x7b, 0x76, 0x55, 0x58, 0x4f, 0x42, 0x09, 0x04, 0x13, 0x1e, 0x3d, 0x30,
```

```
0x27, 0x2a,
0xb1, 0xbc, 0xab, 0xa6, 0x85, 0x88, 0x9f, 0x92, 0xd9, 0xd4, 0xc3, 0xce, 0xed, 0xe0,
0xf7, 0xfa,
0xb7, 0xba, 0xad, 0xa0, 0x83, 0x8e, 0x99, 0x94, 0xdf, 0xd2, 0xc5, 0xc8, 0xeb, 0xe6,
0xf1, 0xfc,
0x67, 0x6a, 0x7d, 0x70, 0x53, 0x5e, 0x49, 0x44, 0x0f, 0x02, 0x15, 0x18, 0x3b, 0x36,
0x21, 0x2c,
0x0c, 0x01, 0x16, 0x1b, 0x38, 0x35, 0x22, 0x2f, 0x64, 0x69, 0x7e, 0x73, 0x50, 0x5d,
0x4a, 0x47,
0xdc, 0xd1, 0xc6, 0xcb, 0xe8, 0xe5, 0xf2, 0xff, 0xb4, 0xb9, 0xae, 0xa3, 0x80, 0x8d,
0x9a, 0x97};

#define XtimeE(i) (pgm_read_byte(&P_XtimeE[i]))
const unsigned char P_XtimeE[256] __attribute__ ((__progmem__)) = {
0x00, 0x0e, 0x1c, 0x12, 0x38, 0x36, 0x24, 0x2a, 0x70, 0x7e, 0x6c, 0x62, 0x48, 0x46,
0x54, 0x5a,
0xe0, 0xee, 0xfc, 0xf2, 0xd8, 0xd6, 0xc4, 0xca, 0x90, 0x9e, 0x8c, 0x82, 0xa8, 0xa6,
0xb4, 0xba,
0xdb, 0xd5, 0xc7, 0xc9, 0xe3, 0xed, 0xff, 0xf1, 0xab, 0xa5, 0xb7, 0xb9, 0x93, 0x9d,
0x8f, 0x81,
0x3b, 0x35, 0x27, 0x29, 0x03, 0x0d, 0x1f, 0x11, 0x4b, 0x45, 0x57, 0x59, 0x73, 0x7d,
0x6f, 0x61,
0xad, 0xa3, 0xb1, 0xbf, 0x95, 0x9b, 0x89, 0x87, 0xdd, 0xd3, 0xc1, 0xcf, 0xe5, 0xeb,
0xf9, 0xf7,
0x4d, 0x43, 0x51, 0x5f, 0x75, 0x7b, 0x69, 0x67, 0x3d, 0x33, 0x21, 0x2f, 0x05, 0x0b,
0x19, 0x17,
0x76, 0x78, 0x6a, 0x64, 0x4e, 0x40, 0x52, 0x5c, 0x06, 0x08, 0x1a, 0x14, 0x3e, 0x30,
0x22, 0x2c,
0x96, 0x98, 0x8a, 0x84, 0xae, 0xa0, 0xb2, 0xbc, 0xe6, 0xe8, 0xfa, 0xf4, 0xde, 0xd0,
0xc2, 0xcc,
0x41, 0x4f, 0x5d, 0x53, 0x79, 0x77, 0x65, 0x6b, 0x31, 0x3f, 0x2d, 0x23, 0x09, 0x07,
0x15, 0x1b,
0xa1, 0xaf, 0xbd, 0xb3, 0x99, 0x97, 0x85, 0x8b, 0xd1, 0xdf, 0xcd, 0xc3, 0xe9, 0xe7,
```

```
0xf5, 0xfb,
0x9a, 0x94, 0x86, 0x88, 0xa2, 0xac, 0xbe, 0xb0, 0xea, 0xe4, 0xf6, 0xf8, 0xd2, 0xdc,
0xce, 0xc0,
0x7a, 0x74, 0x66, 0x68, 0x42, 0x4c, 0x5e, 0x50, 0x0a, 0x04, 0x16, 0x18, 0x32, 0x3c,
0x2e, 0x20,
0xec, 0xe2, 0xf0, 0xfe, 0xd4, 0xda, 0xc8, 0xc6, 0x9c, 0x92, 0x80, 0x8e, 0xa4, 0xaa,
0xb8, 0xb6,
0x0c, 0x02, 0x10, 0x1e, 0x34, 0x3a, 0x28, 0x26, 0x7c, 0x72, 0x60, 0x6e, 0x44, 0x4a,
0x58, 0x56,
0x37, 0x39, 0x2b, 0x25, 0x0f, 0x01, 0x13, 0x1d, 0x47, 0x49, 0x5b, 0x55, 0x7f, 0x71,
0x63, 0x6d,
0xd7, 0xd9, 0xcb, 0xc5, 0xef, 0xe1, 0xf3, 0xfd, 0xa7, 0xa9, 0xbb, 0xb5, 0x9f, 0x91,
0x83, 0x8d};

// exchanges columns in each of 4 rows
// row0 - unchanged, row1- shifted left 1,
// row2 - shifted left 2 and row3 - shifted left 3
void ShiftRows (unsigned char *state)
{
        unsigned char tmp;

        // just substitute row 0
        state[0] = Sbox(state[0]), state[4] = Sbox(state[4]);
        state[8] = Sbox(state[8]), state[12] = Sbox(state[12]);

        // rotate row 1
        tmp = Sbox(state[1]), state[1] = Sbox(state[5]);
        state[5] = Sbox(state[9]), state[9] = Sbox(state[13]), state[13] = tmp;

        // rotate row 2
        tmp = Sbox(state[2]), state[2] = Sbox(state[10]), state[10] = tmp;
        tmp = Sbox(state[6]), state[6] = Sbox(state[14]), state[14] = tmp;
```

```
        // rotate row 3
        tmp = Sbox(state[15]), state[15] = Sbox(state[11]);
        state[11] = Sbox(state[7]), state[7] = Sbox(state[3]), state[3] = tmp;
}

// restores columns in each of 4 rows
// row0 - unchanged, row1- shifted right 1,
// row2 - shifted right 2 and row3 - shifted right 3
void InvShiftRows (unsigned char *state)
{
        unsigned char tmp;

        // restore row 0
        state[0] = InvSbox(state[0]), state[4] = InvSbox(state[4]);
        state[8] = InvSbox(state[8]), state[12] = InvSbox(state[12]);

        // restore row 1
        tmp = InvSbox(state[13]), state[13] = InvSbox(state[9]);
        state[9] = InvSbox(state[5]), state[5] = InvSbox(state[1]), state[1] = tmp;

        // restore row 2
        tmp = InvSbox(state[2]), state[2] = InvSbox(state[10]), state[10] = tmp;
        tmp = InvSbox(state[6]), state[6] = InvSbox(state[14]), state[14] = tmp;

        // restore row 3
        tmp = InvSbox(state[3]), state[3] = InvSbox(state[7]);
        state[7] = InvSbox(state[11]), state[11] = InvSbox(state[15]), state[15] = tmp;
}

// recombine and mix each row in a column
void MixSubColumns (unsigned char *state)
{
unsigned char tmp[4 * Nb];
```

```
        // mixing column 0
        tmp[0] = Xtime2Sbox(state[0]) ^ Xtime3Sbox(state[5]) ^ Sbox(state[10]) ^
Sbox(state[15]);
        tmp[1] = Sbox(state[0]) ^ Xtime2Sbox(state[5]) ^ Xtime3Sbox(state[10]) ^
Sbox(state[15]);
        tmp[2] = Sbox(state[0]) ^ Sbox(state[5]) ^ Xtime2Sbox(state[10]) ^
Xtime3Sbox(state[15]);
        tmp[3] = Xtime3Sbox(state[0]) ^ Sbox(state[5]) ^ Sbox(state[10]) ^
Xtime2Sbox(state[15]);

        // mixing column 1
        tmp[4] = Xtime2Sbox(state[4]) ^ Xtime3Sbox(state[9]) ^ Sbox(state[14]) ^
Sbox(state[3]);
        tmp[5] = Sbox(state[4]) ^ Xtime2Sbox(state[9]) ^ Xtime3Sbox(state[14]) ^
Sbox(state[3]);
        tmp[6] = Sbox(state[4]) ^ Sbox(state[9]) ^ Xtime2Sbox(state[14]) ^
Xtime3Sbox(state[3]);
        tmp[7] = Xtime3Sbox(state[4]) ^ Sbox(state[9]) ^ Sbox(state[14]) ^
Xtime2Sbox(state[3]);

        // mixing column 2
        tmp[8] = Xtime2Sbox(state[8]) ^ Xtime3Sbox(state[13]) ^ Sbox(state[2]) ^
Sbox(state[7]);
        tmp[9] = Sbox(state[8]) ^ Xtime2Sbox(state[13]) ^ Xtime3Sbox(state[2]) ^
Sbox(state[7]);
        tmp[10]  = Sbox(state[8]) ^ Sbox(state[13]) ^ Xtime2Sbox(state[2]) ^
Xtime3Sbox(state[7]);
        tmp[11]  = Xtime3Sbox(state[8]) ^ Sbox(state[13]) ^ Sbox(state[2]) ^
Xtime2Sbox(state[7]);

        // mixing column 3
        tmp[12] = Xtime2Sbox(state[12]) ^ Xtime3Sbox(state[1]) ^ Sbox(state[6]) ^
```

```
Sbox(state[11]);
        tmp[13] = Sbox(state[12]) ^ Xtime2Sbox(state[1]) ^ Xtime3Sbox(state[6]) ^
Sbox(state[11]);
        tmp[14] = Sbox(state[12]) ^ Sbox(state[1]) ^ Xtime2Sbox(state[6]) ^
Xtime3Sbox(state[11]);
        tmp[15] = Xtime3Sbox(state[12]) ^ Sbox(state[1]) ^ Sbox(state[6]) ^
Xtime2Sbox(state[11]);

        memcpy (state, tmp, sizeof(tmp));
}

// restore and un-mix each row in a column
void InvMixSubColumns (unsigned char *state)
{
unsigned char tmp[4 * Nb];
int i;

        // restore column 0
        tmp[0] = XtimeE(state[0]) ^ XtimeB(state[1]) ^ XtimeD(state[2]) ^
Xtime9(state[3]);
        tmp[5] = Xtime9(state[0]) ^ XtimeE(state[1]) ^ XtimeB(state[2]) ^
XtimeD(state[3]);
        tmp[10] = XtimeD(state[0]) ^ Xtime9(state[1]) ^ XtimeE(state[2]) ^
XtimeB(state[3]);
        tmp[15] = XtimeB(state[0]) ^ XtimeD(state[1]) ^ Xtime9(state[2]) ^
XtimeE(state[3]);

        // restore column 1
        tmp[4] = XtimeE(state[4]) ^ XtimeB(state[5]) ^ XtimeD(state[6]) ^
Xtime9(state[7]);
        tmp[9] = Xtime9(state[4]) ^ XtimeE(state[5]) ^ XtimeB(state[6]) ^
XtimeD(state[7]);
        tmp[14] = XtimeD(state[4]) ^ Xtime9(state[5]) ^ XtimeE(state[6]) ^
```

```
XtimeB(state[7]);
        tmp[3] = XtimeB(state[4]) ^ XtimeD(state[5]) ^ Xtime9(state[6]) ^
XtimeE(state[7]);

        // restore column 2
        tmp[8] = XtimeE(state[8]) ^ XtimeB(state[9]) ^ XtimeD(state[10]) ^
Xtime9(state[11]);
        tmp[13] = Xtime9(state[8]) ^ XtimeE(state[9]) ^ XtimeB(state[10]) ^
XtimeD(state[11]);
        tmp[2]  = XtimeD(state[8]) ^ Xtime9(state[9]) ^ XtimeE(state[10]) ^
XtimeB(state[11]);
        tmp[7]  = XtimeB(state[8]) ^ XtimeD(state[9]) ^ Xtime9(state[10]) ^
XtimeE(state[11]);

        // restore column 3
        tmp[12] = XtimeE(state[12]) ^ XtimeB(state[13]) ^ XtimeD(state[14]) ^
Xtime9(state[15]);
        tmp[1] = Xtime9(state[12]) ^ XtimeE(state[13]) ^ XtimeB(state[14]) ^
XtimeD(state[15]);
        tmp[6] = XtimeD(state[12]) ^ Xtime9(state[13]) ^ XtimeE(state[14]) ^
XtimeB(state[15]);
        tmp[11] = XtimeB(state[12]) ^ XtimeD(state[13]) ^ Xtime9(state[14]) ^
XtimeE(state[15]);

        for( i=0; i < 4 * Nb; i++ )
                state[i] = InvSbox(tmp[i]);
}

// encrypt/decrypt columns of the key
// n.b. you can replace this with
//      byte-wise xor if you wish.

void AddRoundKey (unsigned *state, unsigned *key)
```

```
{
        int idx;

        for( idx = 0; idx < 4; idx++ )
                state[idx] ^= key[idx];
}

unsigned char Rcon[11] = {0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b, 0x36};

// produce Nb bytes for each round
void ExpandKey (unsigned char *key, unsigned char *expkey)
{
        unsigned char tmp0, tmp1, tmp2, tmp3, tmp4;
        unsigned idx;

        memcpy (expkey, key, Nk * 4);

        for( idx = Nk; idx < Nb * (Nr + 1); idx++ ) {
                tmp0 = expkey[4*idx - 4];
                tmp1 = expkey[4*idx - 3];
                tmp2 = expkey[4*idx - 2];
                tmp3 = expkey[4*idx - 1];
                if( !(idx % Nk) ) {
                        tmp4 = tmp3;
                        tmp3 = Sbox(tmp0);
                        tmp0 = Sbox(tmp1) ^ Rcon[idx/Nk];
                        tmp1 = Sbox(tmp2);
                        tmp2 = Sbox(tmp4);
                } else if( Nk > 6 && idx % Nk == 4 ) {
                        tmp0 = Sbox(tmp0);
                        tmp1 = Sbox(tmp1);
                        tmp2 = Sbox(tmp2);
```

```
                tmp3 = Sbox(tmp3);
            }

            expkey[4*idx+0] = expkey[4*idx - 4*Nk + 0] ^ tmp0;
            expkey[4*idx+1] = expkey[4*idx - 4*Nk + 1] ^ tmp1;
            expkey[4*idx+2] = expkey[4*idx - 4*Nk + 2] ^ tmp2;
            expkey[4*idx+3] = expkey[4*idx - 4*Nk + 3] ^ tmp3;
        }
}

// encrypt one 128 bit block
void Encrypt (unsigned char *in, unsigned char *expkey, unsigned char *out)
{
        unsigned char state[Nb * 4];
        unsigned round;

        memcpy (state, in, Nb * 4);
        AddRoundKey ((unsigned *)state, (unsigned *)expkey);

        for( round = 1; round < Nr + 1; round++ ) {
            if( round < Nr )
                MixSubColumns (state);
            else
                ShiftRows (state);

            AddRoundKey ((unsigned *)state, (unsigned *)expkey + round * Nb);
        }

        memcpy (out, state, sizeof(state));
}

void Decrypt (unsigned char *in, unsigned char *expkey, unsigned char *out)
{
```

```
        unsigned char state[Nb * 4];
        unsigned round;

        memcpy (state, in, sizeof(state));

        AddRoundKey ((unsigned *)state, (unsigned *)expkey + Nr * Nb);
        InvShiftRows(state);

        for( round = Nr; round--; )
        {
                AddRoundKey ((unsigned *)state, (unsigned *)expkey + round * Nb);
                if( round )
                        InvMixSubColumns (state);
        }

        memcpy (out, state, sizeof(state));
}

#define USR UCSRA

void printP (PGM_P string){
      char c;
      c=pgm_read_byte(string);
            while (c) {
            loop_until_bit_is_set(USR, UDRE);
            UDR = c;
            c=pgm_read_byte(++string);
            }
            return;
           }
```

```
void print (const char *string){
                while (*string) {
                loop_until_bit_is_set(USR, UDRE);
                UDR = *string++;
                }
                return;
               }

void scan(char *string){
char c;
      do     {
            do {
                  loop_until_bit_is_set(USR, RXC);
                  c =UDR;
                  } while bit_is_set(USR, FE);
            *string++ = c;
            //echo the character
            loop_until_bit_is_set(USR, UDRE);
            UDR = c;
            } while ( c != '\r' );
      loop_until_bit_is_set(USR, UDRE);
      UDR = '\n';
      string[-1]=0;
      }

void UART_init(void)      // initialize USART
{
         UBRRH = 0;
         UBRRL = 103; // 8MHz, Baudrate: 9600
         UCSRA = (1<<U2X);
         UCSRB = (1<<TXEN)|(1<<RXEN);
         UCSRC = (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1)|(1<<USBS);
}
```

```
#define itoa10(N,S) itoa(N,S,10)
#define itoa16(N,S) itoa(N,S,16)

//DEMO

unsigned char sampleout[16];

unsigned char samplekey[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab,
0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};

unsigned char samplein[] = {0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d, 0x31,
0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34};

int main(void)
{
        UART_init();

        unsigned char expkey[4 * Nb * (Nr + 1)];
        unsigned char i;
        char c[8];

        printP(PSTR("Original: "));
        for( i = 0; i < 16; i++ ) {itoa16(samplein[i],c);print(c); print(" ");}
        printP(PSTR("\n"));

        ExpandKey (samplekey, expkey);
        Encrypt (samplein, expkey, sampleout);

        printP(PSTR("Encrypted: "));
        for( i = 0; i < 16; i++ ) {itoa16(sampleout[i],c);print(c); print(" ");}
        printP(PSTR("\n"));
```

```
        Decrypt (sampleout, expkey, samplein);

        printP(PSTR("Decrypted: "));

        for( i = 0; i < 16; i++ ) {itoa16(samplein[i],c);print(c); print(" ");}
        printP(PSTR("\n"));
}
```