IMPLEMENTASI KRIPTOGRAFI MENGGUNAKAN ALGORITMA *ADVANCED ENCRYPTION STANDARD* (AES) DENGAN METODE CBC (*CHIPER BLOCK CHAINING*) DAN PENGECEKAN *ERROR DETECTION CYCLIC REDUNDANCY CHECK*

Warkim, Irvan Lewelusa
Pusat Penelitian Perkembangan Iptek Lembaga Ilmu Pengetahuan Indonesia (PAPPIPTEK-LIPI)
Gedung A PDII-LIPI Jl. Jend. Gatot Subroto 10 Jakarta Selatan
x1syah@gmail.com

Abstrak

Keamanan data atau informasi merupakan aspek yang sangat penting agar terjamin kerasahasiaan dan keaslian suatu data atau informasi. Kriptografi merupakan salah satu solusi dalam pengamanan data atau informasi agar informasi sampai kepada yang berhak menerima. Algoritma kriptografi yang digunakan dalam pengamanan data yang baik yaitu dengan menggunakan metode AES (*Advanced Encryption Standard*) merupakan algoritma kriptografi simetrik yang memproses blok data 128 bit dengan panjang kunci 128 bit (AES 128), 192 bit (AES 192) atau 256 bit (AES 256). Metode yang diterapkan pada algoritma kriptografi penyandian blok AES ini adalah metode CBC (*Chiper Block Chaining*) dengan melakukan pengecekan *error detection. Implementasi* kriptografi AES ini menggunakan *key* berukuran 16 bit sehingga jika dihitung *possible key* kombinasi ada sekitar 65.536 *key* kombinasi yang artinya jika dilakukan *brute force* maka *effort*-nya akan cukup mudah untuk di *crack*. Dalam pengecekan *error detection cyclic reduncancy check* digunakan untuk mengecek *integrity* dari sebuah data, dimana data kemungkinan *corrup* sehingga ada bit-bit yang hilang pada saat di transmisikan pada jaringan.

Kata kunci : kriptografi, AES, *error detection*

Pendahuluan

Kriptografi merupakan salah satu metode untuk menjaga pengamanan data agar terjamin kerahasiaan dan keaslian data serta dapat meningkatkan aspek keamanan suatu data atau informasi. Kriptografi mendukung kebutuhan dua aspek keamanan informasi yaitu perlindungan terhadap kerahasian informasi dan perlindungan terhadap pemalsuan dan pengubahan informasi yang disampaikan dari pengirim kepada penerima informasi. Untuk mengetahui suatu algoritma kriptografi dapat mengamankan data dengan baik, dapat dilihat dari lamanya waktu proses pembobolan untuk memecahkan data yang telah di sandikan. Salah satu algoritma yang dipakai dalam kriptografi adalah metode *Advanced Encryption Standard* (AES) merupakan algoritma kriptografi simetrik yang memproses blok data 128 bit dengan panjang kunci 128 bit (AES 128), 192 bit (AES 192) atau 256 bit (AES 256). Metode yang diterapkan pada algoritma kriptografi penyandi blok AES antara lain *Electronic Code Block* (ECB), *Chiper Block Chaining* (CBC), *Chiper Feedback* (CFB) dan *Output Feedback* (OFB). Dari beberapa metode kriptografi AES tersebut memiliki kelebihan dan kekurangan dalam aspek tingkat keamanan data.

Rumusan Masalah

Rumusan masalah dalam penulisan paper ini adalan sebagai berikut:

1. Bagaimana teknik mengamankan data teks?
2. Bagaimana menerapkan metode kriptografi AES pada pengamanan data teks?
3. Bagaimana merancang perangkat lunak untuk pengamanan data teks dengan menggunakan metode kriptografi AES dengan menggunakan metode CBC dan pengecekan *error detection*.

Ruang Lingkup

Agar pembahasan masalah tidak terlalu luas, cakupan masalah pada penulisan *paper* ini tentang kriptografi dengan menggunakan algoritma AES ini adalah sebagai berikut:

1. Data atau informasi yang diamankan untuk tipe teks dan dilakukan melalui *input keyboard*.
2. Metode kriptografi yang digunakan adalah metode kriptografi AES

Tujuan dan Manfaat

Tujuan penulisan *paper* yaitu menjelaskan proses pengamanan data pada saat pengiriman data teks dengan menggunakan metode kriptografi AES serta merancang perangkat lunak pengamanan data mengunakan metode kriptografi AES dengan pemrograman bahasa C. Sedangkan manfaatnya adalah dapat mempermudah dalam pengolahan data atau informasi untuk pengguna serta mempermudah dalam pengamanan data yang diinginkan.

Kriptografi

Kriptografi berasal dari bahasa Yunani terdiri dari kata *kryptos yang* artinya rahasia dan kata *graphia* yang artinya sesuatu yang terlulis, sehingga kriptografi dapat juga disebut sebagai sesuatu yang tertulis secara rahasia. Kriptografi merupakan ilmu yang mempelajari teknik-teknik matematika yang berhubungan dengan aspek keamanan informasi seperti kerahasiaan, integritas data serta otentikasi (Alfred J. Menezes, 2001). Dengan berkembangan teknologi informasi ilmu penyembunyian informasi dapat digunakan untuk menghindari berbagai serangan dari pihak yang tidak bertanggung jawab, seperti adanya penyadapan dan pengubahan data yang sedang dikirimkan. Kriptografi tidak berarti hanya memberikan keamanan data dan informasi saja melainkan lebih ke arah teknik-tekniknya. Dalam kriptografi, pesat atau informasi yang dibaca disebut sebagai *plaintext* atau *clear text*. Proses yang dilakukan untuk mengubah *plaintext* ke dalam *chipertext* disebut enkripsi. Sedangkan pesan yang tidak dapat terbaca disebut dengan istilah *chipertext*. Proses kebalikan dari enkripsi dinamakan dekripsi yaitu proses yang akan mengembalikan *chipertext* menjadi *plaintext*. Kedua proses enkripsi dan dekripsi membutuhkan penggunaan sejumlah informasi rahasia yang disebut kunci (*key*).

Algoritma AES

Algoritma AES menggunakan substitusi dan permutasi, dan sejumlah putaran (*chiper* berulang) dimana setiap putaran menggunakan kunci (*key*) yang berbeda yang dinamakan *round key*. Algoritma AES menetapkan panjang kuncinya 128, 192, 256 bit. Oleh karena itu dikenal dengan AES-128, AES-192 dan AES-256. Dari ketiga versi AES tersebut terdapat perbedaan pada jumlah key dan putarannya seperti pada tabel 1.

Tabel 1
Perbedaan Tiga Versi Algoritma AES

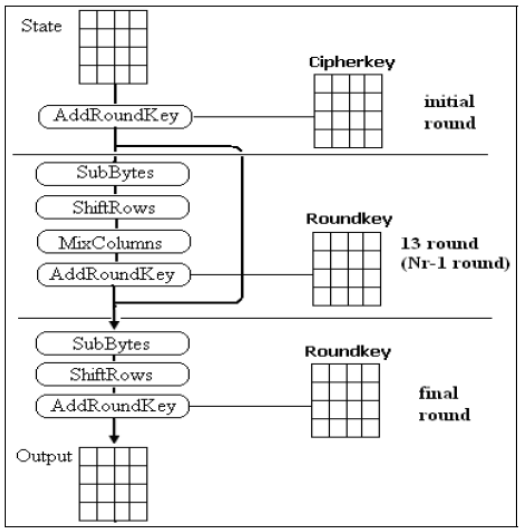
Versi AES	Jumlah Key (*Nk Words*)	Jumlah Blok (*Nb Words*)	Jumlah Putaran (*Nr*)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Sumber : Munir, 2006: 158
Keterangan : 1 Words = 32 bit

Proses Enkripsi

Proses enkripsi pada algoritma AES terdiri dari 4 (empat) jenis transformasi *bytes*, yaitu *SubBytes*, *ShiftRows*, *Mixcolumns* dan *AddRoundKey*. Pada awal proses dilakukan enkripsi, *input* yang telah dikopikan kedalan *state* akan mengalami transformasi *byte AddRoundKey*. Setelah itu, *state* akan mengalami *SubBytes*, *ShiftRows*, *MixColumns* dan *AddRoundKey* secara berulang-ulang sebanyak Nr. Proses ini dalam algoritma dapat disebut dengan *round function*. *Round* yang terakhir agak berbeda dengan *round-round* sebelumnya dimana pada *round* terakhir, *state*

tidak mengalami transformasi *MixColumns* (Rifki Sadikin, 2012). Proses enkripsi adalah proses penyandian pesan terbuka yang disebut dengan *plaintext* akan menjadi pesan rahasia (*chipertext*). Pada chipertext inilah yang nantinya akan dikirimkan oleh pengirim melalui saluran komunikasi terbuka. Pada saat *chipertext* diterima oleh penerima pesan maka pesan rahasia yang dikirim akan diubah menjadi pesan terbuka dengan melalui proses dekripsi sehingga pesan yang tadi dikirim dapat dibaca kembali oleh penerima pesan. Proses enkripsi seperti yang terlihat pada gambar 1.



Sumber: Munir, 2006: 159

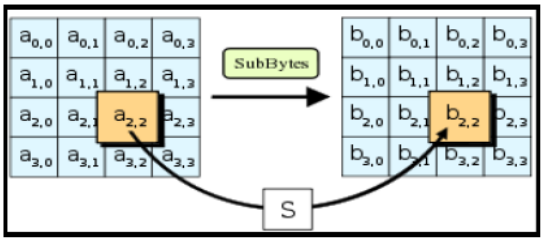
Gambar 1
Diagram Proses Enkripsi

1) *AddRoundKey* : melakukan XOR antara *state* awal (*plaintext*) dengan *chiper key* pada tahapan ini disebut *initial round*. Putaran sebanyak Nr-1 kali dimana proses yang dilakukan pada setiap putaran.
2) *SubBytes* : Substitusi byte dengan menggunakan tabel substitusi (S-Box). Tabel 2 merupakan tabel S-Box SubBytes, untuk setiap byte pada array state, misalkan S[r,c]=xy dimana xy adalah digit hexadesimaldari nilai S[r,c] maka nilai substitusinya dinyatakan dengan S'[r,c] adalah elemen didalam S-Box yang merupakan perpotongan baris x dengan kolom y. Gambar 2 merupakan gambar proses transformasi SubBytes.

Tabel 2
S-Box *SubBytes*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
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

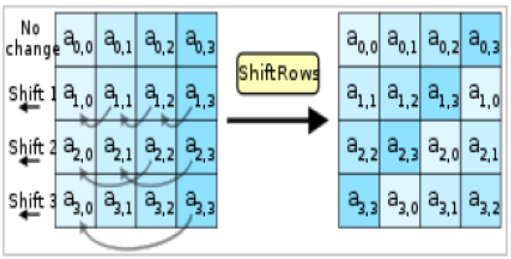
Sumber : Munir, 2006:163



Sumber: Munir, 2006:163

Gambar 2
Transformasi *SubBytes*

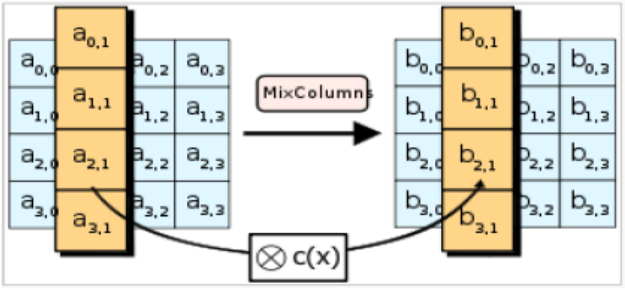
1) *ShiftRows* : pergeseran baris-baris *array state* secara *wrapping* pada ketiga baris terakhir dari *array state*, dimana pada proses ini *bit* paling kiri akan dipindahkan menjadi bit paling kanan. Jumlah pergeseran bergantung pada nilai baris (r). Baris r-1 digeser 1 byte, baris r=2 digeser sejauh 2 byte dan baris r=3 digeser sejauh 3 byte sedangkan baris r=0 tidak digeser seperti yang terlihat pada gambar 3.



Sumber: Munir, 2006:165

Gambar 3
Transformasi *ShiftRows*

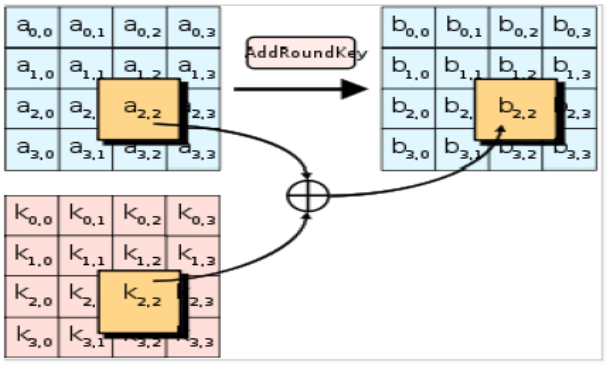
2) *MixColumns* : mengacak-acak data dimasing-masing kolom *array state.* Dalam proses *MixColumns* terdapat beberapa perkalian yaitu *Matrix Multiplication* dan *Galois Field Multiplication.* (gambar 4.)



Sumber: Munir, 2006:166

Gambar 4
Tranformasi *MixColumns*

3) *AddRoundKey* : melakukan XOR antara *state* sekarang dengan *round key*.



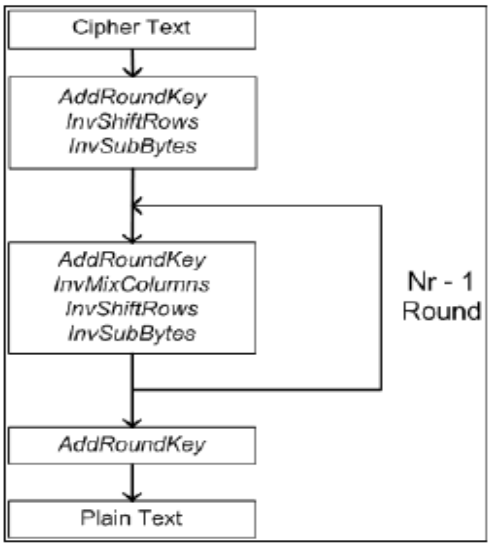
Sumber: Munir, 2006:167

Gambar 5
Transformasi *AddRoundKey*

4) *Final round* : proses untuk putaran terakhir (*SubBytes, ShiftRows, AddRoundKey*).

Proses Dekripsi

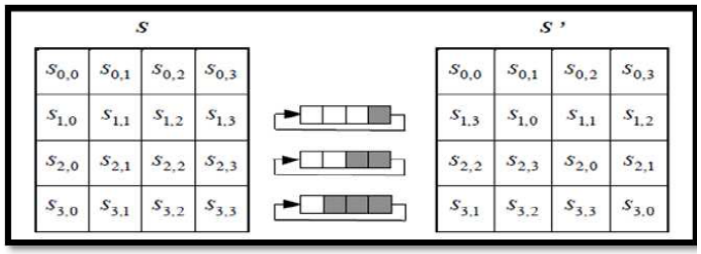
Dekripsi merupakan proses kebalikan dari proses enkripsi, yaitu merubah *chipertext* kembali kedalam bentuk *plaintext*. Untuk menghilangkan penyandianyang telah diberikan pada saat proses enkripsi, membutuhkan peng-gunaan sejumlah informasi rahasia, yang disebut sebagai kunci.



Gambar 6
Proses Dekripsi AES

Transformasi *chiper* dapat dibalikkan dan diimplementasikan dalam arah yang berlawanan untuk menghasilkan *invers chiper* yang mudah dipahami untuk algoritma AES. Transformasi *byte* yang digunakan pada *invers chiper* adalah *InvShiftRows, InvSubBytes, InvMixColumns* dan *AddRoundKey*.

1) *InvShiftRows* : merupakan transformasi bytes yang kebalikan dengan transformasi *ShiftRows*. Pada transformasi *InvShiftRows* dilakukan pergeseran bit ke kanan sedangkan pada *ShiftRows* dilakukan pergeseran bit ke kiri. (Gambar 7.)



Sumber : Munir, 2006

Gambar 7
Transformasi *InvShiftRows*

2) *InvSubBytes* : merupakan transformasi *bytes* yang berkebalikan dengan transformasi *SubBytes*. Pada *InvSubBytes*, tiap elemen pada *state* dipetakan dengan menggunakan tabel *Inverse S-Box*.

Tabel 3
Tabel *Invers S-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
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

Sumber : Munir, 2006

InvMicColumn: merupakan setiap kolom state dikalikan dengan matriks perkalian dalam AES. Perkalian dalam matriks dapat dituliskan sebagai berikut:

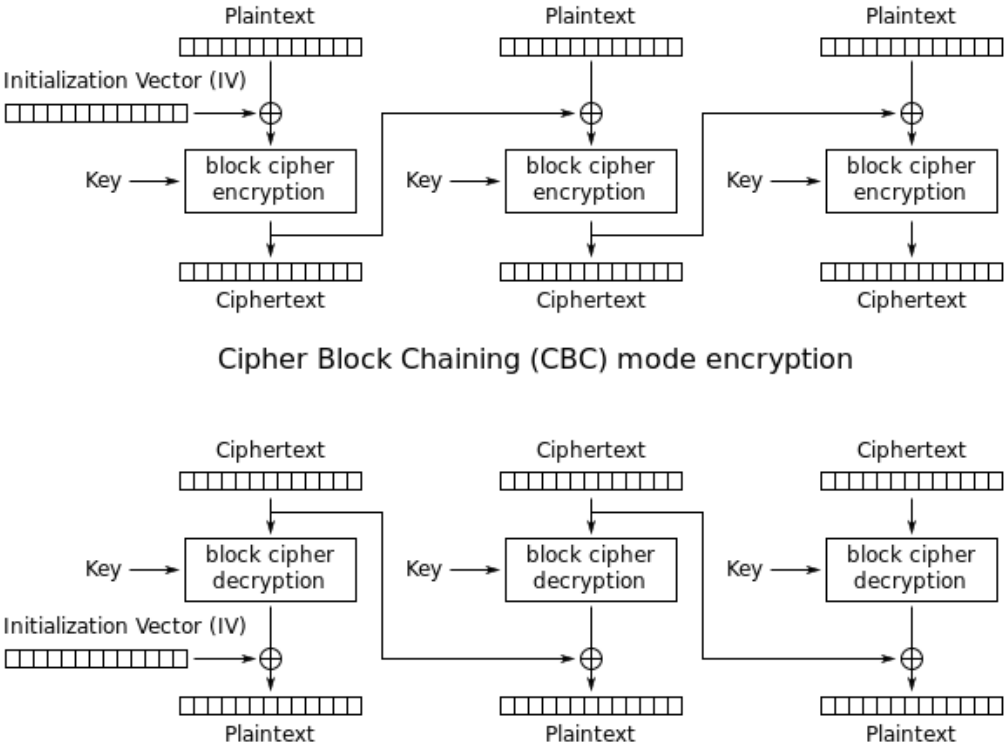
$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Hasil dari perkalian dalam matriks adalah sebagai berikut:

$$s'_{0,c} = (\{0E\} \bullet s_{o,c}) \oplus (\{0B\} \bullet s_{1,c}) \oplus (\{0D\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$
$$s'_{1,c} = (\{09\} \bullet s_{o,c}) \oplus (\{0E\} \bullet s_{1,c}) \oplus (\{0B\} \bullet s_{2,c}) \oplus (\{0D\} \bullet s_{3,c})$$
$$s'_{2,c} = (\{0D\} \bullet s_{o,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0E\} \bullet s_{2,c}) \oplus (\{0B\} \bullet s_{3,c})$$
$$s'_{3,c} = (\{0B\} \bullet s_{o,c}) \oplus (\{0D\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0E\} \bullet s_{3,c})$$

Mode operasi CBC

Mode operasi yang dipergunakan dalam pembuatan program enkripsi ini adalah CBC. Pada tahun 2001 US *National Institute of Standard and Technology* (NIST) merevisi sejumlah mode operasi dan memasukan AES sebagai salah satu blok chiper pada mode operasi (CBC, ECB, OFB, CTR). *Cipher Block Chaining* (CBC), ditemukan oleh IBM pada tahun 1976. Pada mode CBCB, setiap blok *plaintext* dengan chipertex sebelumnnya dibuat sebagai XOR sebelum di enkripsikan. Dengan demikiansetiap blok *ciphretext* sangat bergantung kepada blok *plaintext* yang telah diproses. Untuk membuat setiap mesage adalah unik, maka sebuah *initialization vector* wajib digunakan pada blok pertama, untuk lebih jelasnya dapat dilihat pada ilustrasi gambar dibawah ini.



Gambar 7
Cipher Block Chining (CBC) mode encryption-decryption

Jika pada blok pertama memiliki indeks 1, persamaan matematika untuk mode CBC adalah sebagai berikut

CBC encryption :

$$C_i = E_K(P_i \oplus C_{i-1}), C_0 = IV$$

CBC decryption :

$$P_i = D_K(C_i) \oplus C_{i-1}, C_0 = IV.$$

CBC telah banyak diterapkan, kekurangannya adalah bahwa proses enkripsi dilakukan secara sequensial (tidak dapat dilakukan secara paralel), selain itu message yang akan di enkrip harus di pampatkan ke sebuah *multiple chiper blok size*. Salah satu cara untuk mengatasi masalah tersebut adalah dengan menggunakan sebuah metoda yang dinamakan dengan *ciphertext stealing.* Setiap bit dirubah pada sebuah *plaintext* atau IV mempengaruhi keseluruhan *ciphertext block*. *Decrypting* dengan IV yang salah akan menyebabkan blok pertama dari sebuah *plaintext* akan korup namun *plaintext* yang berurutan akan menjadi benar. Hal ini dikarenakan sebuah *block plaintext* dapat di *recovery* dari dua blok *chipertext* yang saling bersebelahan. Sebagai konsekuensinya, proses dekripsi prosesnya dapat diparalelkan.

***Error Detection* dengan CRC (*Cyclic Redundancy Check*)**

Data dapat menjadi *error* pada saat ditransmisikan, error dapat diakibatkan oleh berubahnya bit-bit. Untuk mendeteksi error pada makalah ini menggunakan CRC, yaitu salah satu skema atau metode pengecekan erro. Metode CRC mengunakan *binary division Finite Algebra Theory.* Dalam implementasinya metode CRC menggunakan konsep *shift register* dan XOR untuk *addition* dan *subtraction.*

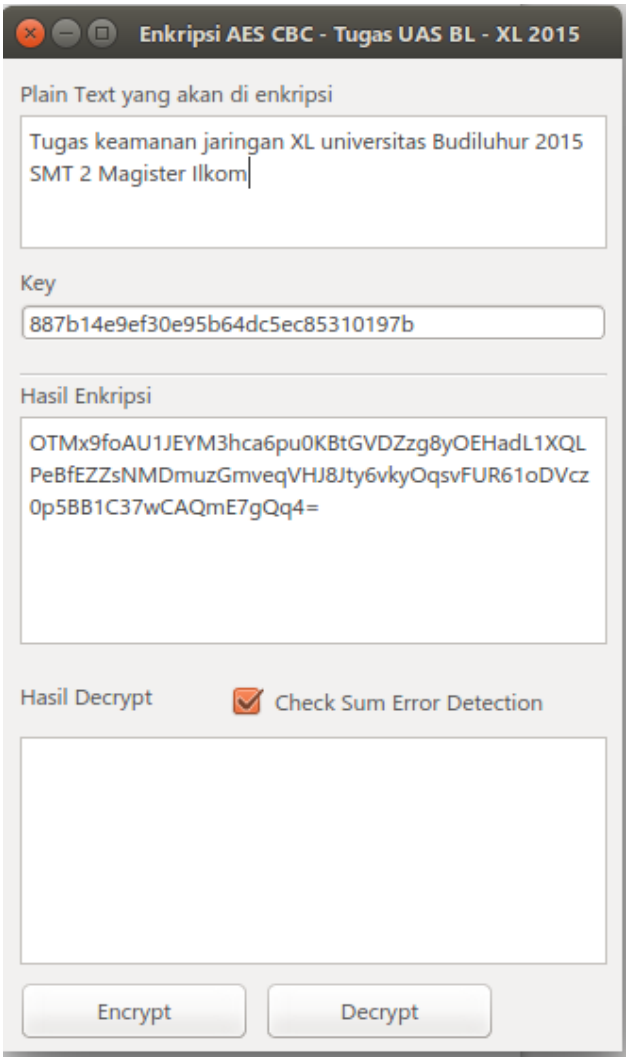
Perhitungan n-bit *binary* CRC yang dilakukan menggunakan 16 bit *message encrypted text* dengan menggunakan 3 bit CRC, sehingga *polynomial* yang digunakan adalah *polynomial* $x^3 + x + 1$. Hasil *division* harus menghasilkan *reminder* = 0, dimana hal ini berarti tidak ada *error*.

Penerapan Algoritma Aes

Proses Enkripsi

Penerapan algoritma AES untuk proses enkripsi terdiri dari beberapa *form* antara lain *form* pengisian *plaintext*, *key*, hasil enkripsi serta hasil dekripsi dengan menggunakan *check sum error detection* seperti yang ditampilkan dalam gambar dibawah ini.

Proses penerapan algoritma AES dengan menyertakan *error detection* pilihan *check box check sum error detection* harus di *checked* dari *key* akan *dicreate header tag* yang akan disisipkan ke dalam file hasil enkripsi. *Header* ini akan dicek pada saat *decrypt*, sehingga apabila teks enkripsi dirubah makan tidak akan dapat di *decrypt* meskipun menggunakan *key* yang sudah benar.



***Source Code* penerapan algoritma AES**

1) *Class* AESBL sebagai *base abstract class constructor* dimana fungsinya akan di *implement* kemudian.

2)

```
class AESBL
{
    public:
        AESBL();
        // public functions
        QByteArray Encrypt(QByteArray p_input, QByteArray p_key);
        QByteArray Decrypt(QByteArray p_input, QByteArray p_key);
        QByteArray Encrypt(QByteArray p_input, QByteArray p_key, QByteArray p_iv);
        QByteArray Decrypt(QByteArray p_input, QByteArray p_key, QByteArray p_iv);
        QByteArray HexStringToByte(QString key);
    private:
        typedef struct
        {
            uint_8t ksch[(N_MAX_ROUNDS + 1) * N_BLOCK];
            uint_8t rnd;
        } aes_context;
        // QT helper functions
        void QByteArrayToUCharArray(QByteArray src, unsigned char *dest);
        QByteArray UCharArrayToQByteArray(unsigned char *src, int p_size);
        QByteArray RemovePadding(QByteArray input);
        QByteArray AddPadding(QByteArray input);
        QByteArray GenerateRandomBytes(int length);
        // encryption functions
        aes_result aes_set_key(const unsigned char key[], int keylen, aes_context ctx[1] );
        aes_result aes_encrypt( const unsigned char in[N_BLOCK], unsigned char out[N_BLOCK], const aes_context
ctx[1] );
        aes_result aes_decrypt( const unsigned char in[N_BLOCK], unsigned char out[N_BLOCK], const aes_context
ctx[1] );
        aes_result aes_cbc_encrypt(const unsigned char *in, unsigned char *out, unsigned long size, unsigned
char iv[N_BLOCK],
                                   const aes_context ctx[1] );
        aes_result aes_cbc_decrypt(const unsigned char *in, unsigned char *out, unsigned long size, unsigned
char iv[N_BLOCK],
                                   const aes_context ctx[1] );
        // helper functions
        void xor_block( void *d, const void *s );
        void copy_and_key( void *d, const void *s, const void *k );
        void add_round_key( uint_8t d[N_BLOCK], const uint_8t k[N_BLOCK] );
        void shift_sub_rows( uint_8t st[N_BLOCK] );
        void inv_shift_sub_rows( uint_8t st[N_BLOCK] );
        void mix_sub_columns( uint_8t dt[N_BLOCK] );
        void inv_mix_sub_columns( uint_8t dt[N_BLOCK] );
};
```

3) Konstanta *Sbox* dan *Inverse sbox*, *shift* colom dan baris:

```
#define N_ROW              4
#define N_COL                   4
#define N_BLOCK   (N_ROW * N_COL)
#define N_MAX_ROUNDS          14
#define WPOLY   0x011b
#define BPOLY     0x1b
#define DPOLY   0x008d
#define f1(x)   (x)
#define f2(x)   ((x << 1) ^ (((x >> 7) & 1) * WPOLY))
#define f4(x)   ((x << 2) ^ (((x >> 6) & 1) * WPOLY) ^ (((x >> 6) & 2) * WPOLY))
#define f8(x)   ((x << 3) ^ (((x >> 5) & 1) * WPOLY) ^ (((x >> 5) & 2) * WPOLY) ¥
                         ^ (((x >> 5) & 4) * WPOLY))
#define d2(x)   (((x) >> 1) ^ ((x) & 1 ? DPOLY : 0))
```

```
#define f3(x)   (f2(x) ^ x)
#define f9(x)   (f8(x) ^ x)
#define fb(x)   (f8(x) ^ f2(x) ^ x)
#define fd(x)   (f8(x) ^ f4(x) ^ x)
#define fe(x)   (f8(x) ^ f4(x) ^ f2(x))
#define s_box(x)     sbox[(x)]
#define is_box(x)    isbox[(x)]
#define gfm2_sb(x)   gfm2_sbox[(x)]
#define gfm3_sb(x)   gfm3_sbox[(x)]
#define gfm_9(x)     gfmul_9[(x)]
#define gfm_b(x)     gfmul_b[(x)]
#define gfm_d(x)     gfmul_d[(x)]
#define gfm_e(x)     gfmul_e[(x)]
#define block_copy_nn(d, s, l)    memcpy(d, s, l)
#define block_copy(d, s)          memcpy(d, s, N_BLOCK)
#define sb_data(w) {    /* S Box data values */                          ¥
    w(0x63), w(0x7c), w(0x77), w(0x7b), w(0xf2), w(0x6b), w(0x6f), w(0xc5),¥
    w(0x30), w(0x01), w(0x67), w(0x2b), w(0xfe), w(0xd7), w(0xab), w(0x76),¥
    w(0xca), w(0x82), w(0xc9), w(0x7d), w(0xfa), w(0x59), w(0x47), w(0xf0),¥
    w(0xad), w(0xd4), w(0xa2), w(0xaf), w(0x9c), w(0xa4), w(0x72), w(0xc0),¥
    w(0xb7), w(0xfd), w(0x93), w(0x26), w(0x36), w(0x3f), w(0xf7), w(0xcc),¥
    w(0x34), w(0xa5), w(0xe5), w(0xf1), w(0x71), w(0xd8), w(0x31), w(0x15),¥
    w(0x04), w(0xc7), w(0x23), w(0xc3), w(0x18), w(0x96), w(0x05), w(0x9a),¥
    w(0x07), w(0x12), w(0x80), w(0xe2), w(0xeb), w(0x27), w(0xb2), w(0x75),¥
    w(0x09), w(0x83), w(0x2c), w(0x1a), w(0x1b), w(0x6e), w(0x5a), w(0xa0),¥
    w(0x52), w(0x3b), w(0xd6), w(0xb3), w(0x29), w(0xe3), w(0x2f), w(0x84),¥
    w(0x53), w(0xd1), w(0x00), w(0xed), w(0x20), w(0xfc), w(0xb1), w(0x5b),¥
    w(0x6a), w(0xcb), w(0xbe), w(0x39), w(0x4a), w(0x4c), w(0x58), w(0xcf),¥
    w(0xd0), w(0xef), w(0xaa), w(0xfb), w(0x43), w(0x4d), w(0x33), w(0x85),¥
    w(0x45), w(0xf9), w(0x02), w(0x7f), w(0x50), w(0x3c), w(0x9f), w(0xa8),¥
    w(0x51), w(0xa3), w(0x40), w(0x8f), w(0x92), w(0x9d), w(0x38), w(0xf5),¥
    w(0xbc), w(0xb6), w(0xda), w(0x21), w(0x10), w(0xff), w(0xf3), w(0xd2),¥
    w(0xcd), w(0x0c), w(0x13), w(0xec), w(0x5f), w(0x97), w(0x44), w(0x17),¥
    w(0xc4), w(0xa7), w(0x7e), w(0x3d), w(0x64), w(0x5d), w(0x19), w(0x73),¥
    w(0x60), w(0x81), w(0x4f), w(0xdc), w(0x22), w(0x2a), w(0x90), w(0x88),¥
    w(0x46), w(0xee), w(0xb8), w(0x14), w(0xde), w(0x5e), w(0x0b), w(0xdb),¥
    w(0xe0), w(0x32), w(0x3a), w(0x0a), w(0x49), w(0x06), w(0x24), w(0x5c),¥
    w(0xc2), w(0xd3), w(0xac), w(0x62), w(0x91), w(0x95), w(0xe4), w(0x79),¥
    w(0xe7), w(0xc8), w(0x37), w(0x6d), w(0x8d), w(0xd5), w(0x4e), w(0xa9),¥
    w(0x6c), w(0x56), w(0xf4), w(0xea), w(0x65), w(0x7a), w(0xae), w(0x08),¥
    w(0xba), w(0x78), w(0x25), w(0x2e), w(0x1c), w(0xa6), w(0xb4), w(0xc6),¥
    w(0xe8), w(0xdd), w(0x74), w(0x1f), w(0x4b), w(0xbd), w(0x8b), w(0x8a),¥
    w(0x70), w(0x3e), w(0xb5), w(0x66), w(0x48), w(0x03), w(0xf6), w(0x0e),¥
    w(0x61), w(0x35), w(0x57), w(0xb9), w(0x86), w(0xc1), w(0x1d), w(0x9e),¥
    w(0xe1), w(0xf8), w(0x98), w(0x11), w(0x69), w(0xd9), w(0x8e), w(0x94),¥
    w(0x9b), w(0x1e), w(0x87), w(0xe9), w(0xce), w(0x55), w(0x28), w(0xdf),¥
    w(0x8c), w(0xa1), w(0x89), w(0x0d), w(0xbf), w(0xe6), w(0x42), w(0x68),¥
    w(0x41), w(0x99), w(0x2d), w(0x0f), w(0xb0), w(0x54), w(0xbb), w(0x16) }
#define isb_data(w) {   /* inverse S Box data values */                  ¥
    w(0x52), w(0x09), w(0x6a), w(0xd5), w(0x30), w(0x36), w(0xa5), w(0x38),¥
    w(0xbf), w(0x40), w(0xa3), w(0x9e), w(0x81), w(0xf3), w(0xd7), w(0xfb),¥
    w(0x7c), w(0xe3), w(0x39), w(0x82), w(0x9b), w(0x2f), w(0xff), w(0x87),¥
    w(0x34), w(0x8e), w(0x43), w(0x44), w(0xc4), w(0xde), w(0xe9), w(0xcb),¥
    w(0x54), w(0x7b), w(0x94), w(0x32), w(0xa6), w(0xc2), w(0x23), w(0x3d),¥
    w(0xee), w(0x4c), w(0x95), w(0x0b), w(0x42), w(0xfa), w(0xc3), w(0x4e),¥
    w(0x08), w(0x2e), w(0xa1), w(0x66), w(0x28), w(0xd9), w(0x24), w(0xb2),¥
    w(0x76), w(0x5b), w(0xa2), w(0x49), w(0x6d), w(0x8b), w(0xd1), w(0x25),¥
    w(0x72), w(0xf8), w(0xf6), w(0x64), w(0x86), w(0x68), w(0x98), w(0x16),¥
    w(0xd4), w(0xa4), w(0x5c), w(0xcc), w(0x5d), w(0x65), w(0xb6), w(0x92),¥
    w(0x6c), w(0x70), w(0x48), w(0x50), w(0xfd), w(0xed), w(0xb9), w(0xda),¥
    w(0x5e), w(0x15), w(0x46), w(0x57), w(0xa7), w(0x8d), w(0x9d), w(0x84),¥
    w(0x90), w(0xd8), w(0xab), w(0x00), w(0x8c), w(0xbc), w(0xd3), w(0x0a),¥
    w(0xf7), w(0xe4), w(0x58), w(0x05), w(0xb8), w(0xb3), w(0x45), w(0x06),¥
    w(0xd0), w(0x2c), w(0x1e), w(0x8f), w(0xca), w(0x3f), w(0x0f), w(0x02),¥
    w(0xc1), w(0xaf), w(0xbd), w(0x03), w(0x01), w(0x13), w(0x8a), w(0x6b),¥
    w(0x3a), w(0x91), w(0x11), w(0x41), w(0x4f), w(0x67), w(0xdc), w(0xea),¥
    w(0x97), w(0xf2), w(0xcf), w(0xce), w(0xf0), w(0xb4), w(0xe6), w(0x73),¥
```

```
    w(0x96), w(0xac), w(0x74), w(0x22), w(0xe7), w(0xad), w(0x35), w(0x85),¥
    w(0xe2), w(0xf9), w(0x37), w(0xe8), w(0x1c), w(0x75), w(0xdf), w(0x6e),¥
    w(0x47), w(0xf1), w(0x1a), w(0x71), w(0x1d), w(0x29), w(0xc5), w(0x89),¥
    w(0x6f), w(0xb7), w(0x62), w(0x0e), w(0xaa), w(0x18), w(0xbe), w(0x1b),¥
    w(0xfc), w(0x56), w(0x3e), w(0x4b), w(0xc6), w(0xd2), w(0x79), w(0x20),¥
    w(0x9a), w(0xdb), w(0xc0), w(0xfe), w(0x78), w(0xcd), w(0x5a), w(0xf4),¥
    w(0x1f), w(0xdd), w(0xa8), w(0x33), w(0x88), w(0x07), w(0xc7), w(0x31),¥
    w(0xb1), w(0x12), w(0x10), w(0x59), w(0x27), w(0x80), w(0xec), w(0x5f),¥
    w(0x60), w(0x51), w(0x7f), w(0xa9), w(0x19), w(0xb5), w(0x4a), w(0x0d),¥
    w(0x2d), w(0xe5), w(0x7a), w(0x9f), w(0x93), w(0xc9), w(0x9c), w(0xef),¥
    w(0xa0), w(0xe0), w(0x3b), w(0x4d), w(0xae), w(0x2a), w(0xf5), w(0xb0),¥
    w(0xc8), w(0xeb), w(0xbb), w(0x3c), w(0x83), w(0x53), w(0x99), w(0x61),¥
    w(0x17), w(0x2b), w(0x04), w(0x7e), w(0xba), w(0x77), w(0xd6), w(0x26),¥
    w(0xe1), w(0x69), w(0x14), w(0x63), w(0x55), w(0x21), w(0x0c), w(0x7d) }
#define mm_data(w) {    /* basic data for forming finite field tables */   ¥
    w(0x00), w(0x01), w(0x02), w(0x03), w(0x04), w(0x05), w(0x06), w(0x07),¥
    w(0x08), w(0x09), w(0x0a), w(0x0b), w(0x0c), w(0x0d), w(0x0e), w(0x0f),¥
    w(0x10), w(0x11), w(0x12), w(0x13), w(0x14), w(0x15), w(0x16), w(0x17),¥
    w(0x18), w(0x19), w(0x1a), w(0x1b), w(0x1c), w(0x1d), w(0x1e), w(0x1f),¥
    w(0x20), w(0x21), w(0x22), w(0x23), w(0x24), w(0x25), w(0x26), w(0x27),¥
    w(0x28), w(0x29), w(0x2a), w(0x2b), w(0x2c), w(0x2d), w(0x2e), w(0x2f),¥
    w(0x30), w(0x31), w(0x32), w(0x33), w(0x34), w(0x35), w(0x36), w(0x37),¥
    w(0x38), w(0x39), w(0x3a), w(0x3b), w(0x3c), w(0x3d), w(0x3e), w(0x3f),¥
    w(0x40), w(0x41), w(0x42), w(0x43), w(0x44), w(0x45), w(0x46), w(0x47),¥
    w(0x48), w(0x49), w(0x4a), w(0x4b), w(0x4c), w(0x4d), w(0x4e), w(0x4f),¥
    w(0x50), w(0x51), w(0x52), w(0x53), w(0x54), w(0x55), w(0x56), w(0x57),¥
    w(0x58), w(0x59), w(0x5a), w(0x5b), w(0x5c), w(0x5d), w(0x5e), w(0x5f),¥
    w(0x60), w(0x61), w(0x62), w(0x63), w(0x64), w(0x65), w(0x66), w(0x67),¥
    w(0x68), w(0x69), w(0x6a), w(0x6b), w(0x6c), w(0x6d), w(0x6e), w(0x6f),¥
    w(0x70), w(0x71), w(0x72), w(0x73), w(0x74), w(0x75), w(0x76), w(0x77),¥
    w(0x78), w(0x79), w(0x7a), w(0x7b), w(0x7c), w(0x7d), w(0x7e), w(0x7f),¥
    w(0x80), w(0x81), w(0x82), w(0x83), w(0x84), w(0x85), w(0x86), w(0x87),¥
    w(0x88), w(0x89), w(0x8a), w(0x8b), w(0x8c), w(0x8d), w(0x8e), w(0x8f),¥
    w(0x90), w(0x91), w(0x92), w(0x93), w(0x94), w(0x95), w(0x96), w(0x97),¥
    w(0x98), w(0x99), w(0x9a), w(0x9b), w(0x9c), w(0x9d), w(0x9e), w(0x9f),¥
    w(0xa0), w(0xa1), w(0xa2), w(0xa3), w(0xa4), w(0xa5), w(0xa6), w(0xa7),¥
    w(0xa8), w(0xa9), w(0xaa), w(0xab), w(0xac), w(0xad), w(0xae), w(0xaf),¥
    w(0xb0), w(0xb1), w(0xb2), w(0xb3), w(0xb4), w(0xb5), w(0xb6), w(0xb7),¥
    w(0xb8), w(0xb9), w(0xba), w(0xbb), w(0xbc), w(0xbd), w(0xbe), w(0xbf),¥
    w(0xc0), w(0xc1), w(0xc2), w(0xc3), w(0xc4), w(0xc5), w(0xc6), w(0xc7),¥
    w(0xc8), w(0xc9), w(0xca), w(0xcb), w(0xcc), w(0xcd), w(0xce), w(0xcf),¥
    w(0xd0), w(0xd1), w(0xd2), w(0xd3), w(0xd4), w(0xd5), w(0xd6), w(0xd7),¥
    w(0xd8), w(0xd9), w(0xda), w(0xdb), w(0xdc), w(0xdd), w(0xde), w(0xdf),¥
    w(0xe0), w(0xe1), w(0xe2), w(0xe3), w(0xe4), w(0xe5), w(0xe6), w(0xe7),¥
    w(0xe8), w(0xe9), w(0xea), w(0xeb), w(0xec), w(0xed), w(0xee), w(0xef),¥
    w(0xf0), w(0xf1), w(0xf2), w(0xf3), w(0xf4), w(0xf5), w(0xf6), w(0xf7),¥
    w(0xf8), w(0xf9), w(0xfa), w(0xfb), w(0xfc), w(0xfd), w(0xfe), w(0xff) }
typedef unsigned char uint_8t;
typedef unsigned long uint_32t;
typedef uint_8t aes_result;
static const uint_8t sbox[256]  =  sb_data(f1);
static const uint_8t isbox[256] = isb_data(f1);
static const uint_8t gfm2_sbox[256] = sb_data(f2);
static const uint_8t gfm3_sbox[256] = sb_data(f3);
static const uint_8t gfmul_9[256] = mm_data(f9);
static const uint_8t gfmul_b[256] = mm_data(fb);
static const uint_8t gfmul_d[256] = mm_data(fd);
static const uint_8t gfmul_e[256] = mm_data(fe);
```

c. *Method* Enkripsi

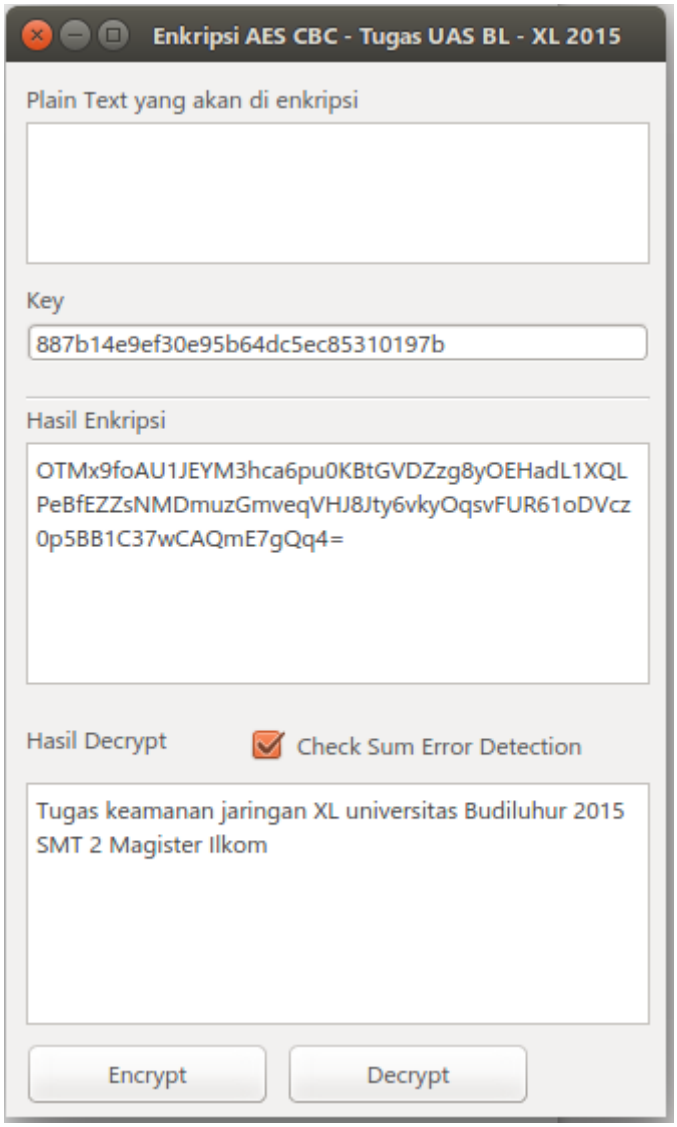
```
#include "aesbl.h"
AESBL::AESBL()
{
}
// encryption with IV
QByteArray AESBL::Encrypt(QByteArray p_input, QByteArray p_key)
{
    QByteArray iv = QUuid::createUuid().toRfc4122();
    QByteArray input = p_input.prepend(iv);
    return Encrypt(input, p_key, iv);
}
QByteArray AESBL::Decrypt(QByteArray p_input, QByteArray p_key)
{
    QByteArray iv = p_input.left(16);
    QByteArray input = p_input.remove(0, 16);
    return Decrypt(input, p_key, iv);
}
// basic encryption
QByteArray AESBL::Encrypt(QByteArray p_input, QByteArray p_key, QByteArray p_iv)
{
    int keySize = p_key.size();
    int ivSize = p_iv.size();
    if (keySize != 16 && keySize != 24 && keySize != 32)
        return QByteArray();
    if (ivSize != 16)
        return QByteArray();
    // add padding
    QByteArray input = AddPadding(p_input);
    int inputSize = input.size();
    unsigned char key[keySize];
    QByteArrayToUCharArray(p_key, key);
    unsigned char iv[ivSize];
    QByteArrayToUCharArray(p_iv, iv);

    unsigned char decrypted[inputSize];
    QByteArrayToUCharArray(input, decrypted);
    unsigned char encrypted[inputSize]; // encrypted text
    aes_context context;
    aes_set_key(key, keySize * 8, &context);
    aes_cbc_encrypt(decrypted, encrypted, inputSize, iv, &context);
    QByteArray result = UCharArrayToQByteArray(encrypted, inputSize);
    return result;
}

// helper functions
QByteArray AESBL::HexStringToByte(QString key)
{
    return QByteArray::fromHex(QString(key).toLatin1());
}
void AESBL::QByteArrayToUCharArray(QByteArray src, unsigned char *dest)
{
    for (int i = 0; i < src.size(); i++)
    {
        dest[i] = src.at(i);
    }
}
QByteArray AESBL::UCharArrayToQByteArray(unsigned char *src, int p_size)
{
    QByteArray array((char*)src, p_size);
    return array;
}
// pkcs#7 padding
QByteArray AESBL::RemovePadding(QByteArray input)
{
    int padding = input.at(input.size() - 1);
    for(int i = 0; i < padding; i++)
    {
```

```
        if (input.at(input.size() - 1) == padding)
        {
            input.chop(1);
        }
    }
    return input;
}
QByteArray AESBL::AddPadding(QByteArray input)
{
    int size = input.size();
    int padding = 16 - (size % 16);
    for(int i = 0; i < padding; i++)
    {
        input.append(padding);
    }
    return input;
}
// algorithm
void AESBL::xor_block( void *d, const void *s )
{
    ((uint_32t*)d)[ 0] ^= ((uint_32t*)s)[ 0];
    ((uint_32t*)d)[ 1] ^= ((uint_32t*)s)[ 1];
    ((uint_32t*)d)[ 2] ^= ((uint_32t*)s)[ 2];
    ((uint_32t*)d)[ 3] ^= ((uint_32t*)s)[ 3];
}
void AESBL::copy_and_key( void *d, const void *s, const void *k )
{
    ((uint_32t*)d)[ 0] = ((uint_32t*)s)[ 0] ^ ((uint_32t*)k)[ 0];
    ((uint_32t*)d)[ 1] = ((uint_32t*)s)[ 1] ^ ((uint_32t*)k)[ 1];
    ((uint_32t*)d)[ 2] = ((uint_32t*)s)[ 2] ^ ((uint_32t*)k)[ 2];
    ((uint_32t*)d)[ 3] = ((uint_32t*)s)[ 3] ^ ((uint_32t*)k)[ 3];
}
void AESBL::add_round_key( uint_8t d[N_BLOCK], const uint_8t k[N_BLOCK] )
{
    xor_block(d, k);
}
void AESBL::shift_sub_rows( uint_8t st[N_BLOCK] )
{
    uint_8t tt;
    st[ 0] = s_box(st[ 0]); st[ 4] = s_box(st[ 4]);
    st[ 8] = s_box(st[ 8]); st[12] = s_box(st[12]);
    tt = st[1]; st[ 1] = s_box(st[ 5]); st[ 5] = s_box(st[ 9]);
    st[ 9] = s_box(st[13]); st[13] = s_box( tt );
    tt = st[2]; st[ 2] = s_box(st[10]); st[10] = s_box( tt );
    tt = st[6]; st[ 6] = s_box(st[14]); st[14] = s_box( tt );
    tt = st[15]; st[15] = s_box(st[11]); st[11] = s_box(st[ 7]);
    st[ 7] = s_box(st[ 3]); st[ 3] = s_box( tt );
}
void AESBL::inv_shift_sub_rows( uint_8t st[N_BLOCK] )
{
    uint_8t tt;
    st[ 0] = is_box(st[ 0]); st[ 4] = is_box(st[ 4]);
    st[ 8] = is_box(st[ 8]); st[12] = is_box(st[12]);
    tt = st[13]; st[13] = is_box(st[9]); st[ 9] = is_box(st[5]);
    st[ 5] = is_box(st[1]); st[ 1] = is_box( tt );
    tt = st[2]; st[ 2] = is_box(st[10]); st[10] = is_box( tt );
    tt = st[6]; st[ 6] = is_box(st[14]); st[14] = is_box( tt );
    tt = st[3]; st[ 3] = is_box(st[ 7]); st[ 7] = is_box(st[11]);
    st[11] = is_box(st[15]); st[15] = is_box( tt );
}
void AESBL::mix_sub_columns( uint_8t dt[N_BLOCK] )
{
    uint_8t st[N_BLOCK];
    block_copy(st, dt);
    dt[ 0] = gfm2_sb(st[0]) ^ gfm3_sb(st[5]) ^ s_box(st[10]) ^ s_box(st[15]);
    dt[ 1] = s_box(st[0]) ^ gfm2_sb(st[5]) ^ gfm3_sb(st[10]) ^ s_box(st[15]);
    dt[ 2] = s_box(st[0]) ^ s_box(st[5]) ^ gfm2_sb(st[10]) ^ gfm3_sb(st[15]);
```

```
    dt[ 3] = gfm3_sb(st[0]) ^ s_box(st[5]) ^ s_box(st[10]) ^ gfm2_sb(st[15]);
    dt[ 4] = gfm2_sb(st[4]) ^ gfm3_sb(st[9]) ^ s_box(st[14]) ^ s_box(st[3]);
    dt[ 5] = s_box(st[4]) ^ gfm2_sb(st[9]) ^ gfm3_sb(st[14]) ^ s_box(st[3]);
    dt[ 6] = s_box(st[4]) ^ s_box(st[9]) ^ gfm2_sb(st[14]) ^ gfm3_sb(st[3]);
    dt[ 7] = gfm3_sb(st[4]) ^ s_box(st[9]) ^ s_box(st[14]) ^ gfm2_sb(st[3]);
    dt[ 8] = gfm2_sb(st[8]) ^ gfm3_sb(st[13]) ^ s_box(st[2]) ^ s_box(st[7]);
    dt[ 9] = s_box(st[8]) ^ gfm2_sb(st[13]) ^ gfm3_sb(st[2]) ^ s_box(st[7]);
    dt[10] = s_box(st[8]) ^ s_box(st[13]) ^ gfm2_sb(st[2]) ^ gfm3_sb(st[7]);
    dt[11] = gfm3_sb(st[8]) ^ s_box(st[13]) ^ s_box(st[2]) ^ gfm2_sb(st[7]);
    dt[12] = gfm2_sb(st[12]) ^ gfm3_sb(st[1]) ^ s_box(st[6]) ^ s_box(st[11]);
    dt[13] = s_box(st[12]) ^ gfm2_sb(st[1]) ^ gfm3_sb(st[6]) ^ s_box(st[11]);
    dt[14] = s_box(st[12]) ^ s_box(st[1]) ^ gfm2_sb(st[6]) ^ gfm3_sb(st[11]);
    dt[15] = gfm3_sb(st[12]) ^ s_box(st[1]) ^ s_box(st[6]) ^ gfm2_sb(st[11]);
  }
void AESBL::inv_mix_sub_columns( uint_8t dt[N_BLOCK] )
{
    uint_8t st[N_BLOCK];
    block_copy(st, dt);
    dt[ 0] = is_box(gfm_e(st[ 0]) ^ gfm_b(st[ 1]) ^ gfm_d(st[ 2]) ^ gfm_9(st[ 3]));
    dt[ 5] = is_box(gfm_9(st[ 0]) ^ gfm_e(st[ 1]) ^ gfm_b(st[ 2]) ^ gfm_d(st[ 3]));
    dt[10] = is_box(gfm_d(st[ 0]) ^ gfm_9(st[ 1]) ^ gfm_e(st[ 2]) ^ gfm_b(st[ 3]));
    dt[15] = is_box(gfm_b(st[ 0]) ^ gfm_d(st[ 1]) ^ gfm_9(st[ 2]) ^ gfm_e(st[ 3]));
    dt[ 4] = is_box(gfm_e(st[ 4]) ^ gfm_b(st[ 5]) ^ gfm_d(st[ 6]) ^ gfm_9(st[ 7]));
    dt[ 9] = is_box(gfm_9(st[ 4]) ^ gfm_e(st[ 5]) ^ gfm_b(st[ 6]) ^ gfm_d(st[ 7]));
    dt[14] = is_box(gfm_d(st[ 4]) ^ gfm_9(st[ 5]) ^ gfm_e(st[ 6]) ^ gfm_b(st[ 7]));
    dt[ 3] = is_box(gfm_b(st[ 4]) ^ gfm_d(st[ 5]) ^ gfm_9(st[ 6]) ^ gfm_e(st[ 7]));
    dt[ 8] = is_box(gfm_e(st[ 8]) ^ gfm_b(st[ 9]) ^ gfm_d(st[10]) ^ gfm_9(st[11]));
    dt[13] = is_box(gfm_9(st[ 8]) ^ gfm_e(st[ 9]) ^ gfm_b(st[10]) ^ gfm_d(st[11]));
    dt[ 2] = is_box(gfm_d(st[ 8]) ^ gfm_9(st[ 9]) ^ gfm_e(st[10]) ^ gfm_b(st[11]));
    dt[ 7] = is_box(gfm_b(st[ 8]) ^ gfm_d(st[ 9]) ^ gfm_9(st[10]) ^ gfm_e(st[11]));
    dt[12] = is_box(gfm_e(st[12]) ^ gfm_b(st[13]) ^ gfm_d(st[14]) ^ gfm_9(st[15]));
    dt[ 1] = is_box(gfm_9(st[12]) ^ gfm_e(st[13]) ^ gfm_b(st[14]) ^ gfm_d(st[15]));
    dt[ 6] = is_box(gfm_d(st[12]) ^ gfm_9(st[13]) ^ gfm_e(st[14]) ^ gfm_b(st[15]));
    dt[11] = is_box(gfm_b(st[12]) ^ gfm_d(st[13]) ^ gfm_9(st[14]) ^ gfm_e(st[15]));
  }
// Set the cipher key for the pre-keyed version
aes_result AESBL::aes_set_key( const unsigned char key[], int keylen, aes_context ctx[1] )
{
    uint_8t cc, rc, hi;
    switch( keylen )
    {
    case 128:
        keylen = 16;
        break;
    case 192:
        keylen = 24;
        break;
    case 256:
        keylen = 32;
        break;
    default:
        ctx->rnd = 0;
        return -1;
    }
    block_copy_nn(ctx->ksch, key, keylen);
    hi = (keylen + 28) << 2;
    ctx->rnd = (hi >> 4) - 1;
    for( cc = keylen, rc = 1; cc < hi; cc += 4 )
    {   uint_8t tt, t0, t1, t2, t3;
        t0 = ctx->ksch[cc - 4];
        t1 = ctx->ksch[cc - 3];
        t2 = ctx->ksch[cc - 2];
        t3 = ctx->ksch[cc - 1];
        if( cc % keylen == 0 )
        {
            tt = t0;
            t0 = s_box(t1) ^ rc;
```

```
            t1 = s_box(t2);
            t2 = s_box(t3);
            t3 = s_box(tt);
            rc = f2(rc);
        }
        else if( keylen > 24 && cc % keylen == 16 )
        {
            t0 = s_box(t0);
            t1 = s_box(t1);
            t2 = s_box(t2);
            t3 = s_box(t3);
        }
        tt = cc - keylen;
        ctx->ksch[cc + 0] = ctx->ksch[tt + 0] ^ t0;
        ctx->ksch[cc + 1] = ctx->ksch[tt + 1] ^ t1;
        ctx->ksch[cc + 2] = ctx->ksch[tt + 2] ^ t2;
        ctx->ksch[cc + 3] = ctx->ksch[tt + 3] ^ t3;
    }
    return 0;
}
// Encrypt a single block of 16 bytes
aes_result AESBL::aes_encrypt( const unsigned char in[N_BLOCK], unsigned char out[N_BLOCK], const aes_context
ctx[1] )
{
    if( ctx->rnd )
    {
        uint_8t s1[N_BLOCK], r;
        copy_and_key( s1, in, ctx->ksch );
        for( r = 1 ; r < ctx->rnd ; ++r )
        {
            mix_sub_columns( s1 );
            add_round_key( s1, ctx->ksch + r * N_BLOCK);
        }
        shift_sub_rows( s1 );
        copy_and_key( out, s1, ctx->ksch + r * N_BLOCK );
    }
    else
        return -1;
    return 0;
}
// CBC encrypt a number of blocks (input and return an IV)
aes_result AESBL::aes_cbc_encrypt(const unsigned char *in, unsigned char *out, unsigned long size, unsigned
char iv[N_BLOCK], const aes_context ctx[1] )
{
    if (size % 16 != 0)
        return EXIT_FAILURE;
    unsigned long n_block = size / 16;
    while(n_block--)
    {
        xor_block(iv, in);
        if(aes_encrypt(iv, iv, ctx) != EXIT_SUCCESS)
            return EXIT_FAILURE;
        memcpy(out, iv, N_BLOCK);
        in += N_BLOCK;
        out += N_BLOCK;
    }
    return EXIT_SUCCESS;
}
```

Proses Dekripsi




```
QByteArray AESBL::Decrypt(QByteArray p_input, QByteArray p_key, QByteArray p_iv)
{
    int inputSize = p_input.size();
    int keySize = p_key.size();
    int ivSize = p_iv.size();
    if (keySize != 16 && keySize != 24 && keySize != 32)
        return QByteArray();
    if (ivSize != 16)
        return QByteArray();
    unsigned char key[keySize];
    QByteArrayToUCharArray(p_key, key);
    unsigned char iv[ivSize];
    QByteArrayToUCharArray(p_iv, iv);
    unsigned char encrypted[inputSize];
    QByteArrayToUCharArray(p_input, encrypted);
    unsigned char decrypted[inputSize]; // decrypted text
    aes_context context;
    aes_set_key(key, keySize * 8, &context);
    aes_cbc_decrypt(encrypted, decrypted, inputSize, iv, &context);
    QByteArray result = RemovePadding(UCharArrayToQByteArray(decrypted, inputSize));
    return result;
```

```
}
// Decrypt a single block of 16 bytes
aes_result AESBL::aes_decrypt( const unsigned char in[N_BLOCK], unsigned char out[N_BLOCK], const
aes_context ctx[1] )
{
    if( ctx->rnd )
    {
        uint_8t s1[N_BLOCK], r;
        copy_and_key( s1, in, ctx->ksch + ctx->rnd * N_BLOCK );
        inv_shift_sub_rows( s1 );
        for( r = ctx->rnd ; --r ; )
        {
            add_round_key( s1, ctx->ksch + r * N_BLOCK );
            inv_mix_sub_columns( s1 );
        }
        copy_and_key( out, s1, ctx->ksch );
    }
    else
        return -1;
    return 0;
}
// CBC decrypt a number of blocks (input and return an IV)
aes_result AESBL::aes_cbc_decrypt( const unsigned char *in, unsigned char *out, unsigned long size,
unsigned char iv[N_BLOCK], const aes_context ctx[1] )
{
    if (size % 16 != 0)
        return EXIT_FAILURE;
    unsigned long n_block = size / 16;
    while (n_block--)
    {
        uint_8t tmp[N_BLOCK];
        memcpy(tmp, in, N_BLOCK);
        if(aes_decrypt(in, out, ctx) != EXIT_SUCCESS)
            return EXIT_FAILURE;
        xor_block(out, iv);
        memcpy(iv, tmp, N_BLOCK);
        in += N_BLOCK;
        out += N_BLOCK;
    }
    return EXIT_SUCCESS;
}
```

3.3. Pengecekan *Checksum Error Detection*

1) Konstanta CRC *polynomial*, *shift register* dan XOR untuk penguranan dan penambahan

```
#ifndef CRC
#define CRC
#define FALSE   0
#define TRUE    !FALSE
/*
 * Select the CRC standard from the list that follows.
 */
#define CRC_CCITT
#if defined(CRC_CCITT)
typedef unsigned short  crc;
#define CRC_NAME                        "CRC-CCITT"
#define POLYNOMIAL                      0x1021
#define INITIAL_REMAINDER       0xFFFF
#define FINAL_XOR_VALUE         0x0000
#define REFLECT_DATA            FALSE
#define REFLECT_REMAINDER       FALSE
```

```
#define CHECK_VALUE                     0x29B1
#elif defined(CRC16)
typedef unsigned short  crc;
#define CRC_NAME                        "CRC-16"
#define POLYNOMIAL                      0x8005
#define INITIAL_REMAINDER       0x0000
#define FINAL_XOR_VALUE         0x0000
#define REFLECT_DATA            TRUE
#define REFLECT_REMAINDER       TRUE
#define CHECK_VALUE                     0xBB3D
#elif defined(CRC32)
typedef unsigned long  crc;
#define CRC_NAME                        "CRC-32"
#define POLYNOMIAL                      0x04C11DB7
#define INITIAL_REMAINDER       0xFFFFFFFF
#define FINAL_XOR_VALUE         0xFFFFFFFF
#define REFLECT_DATA            TRUE
#define REFLECT_REMAINDER       TRUE
#define CHECK_VALUE                     0xCBF43926
#else
#error "One of CRC_CCITT, CRC16, or CRC32 must be #define'd."
#endif
void  crcInit(void);
crc   crcSlow(unsigned char const message[], int nBytes);
crc   crcFast(unsigned char const message[], int nBytes);
#endif // CRC
```

2) *Error detection method*

```
#include "crc.h"
#define WIDTH    (8 * sizeof(crc))
#define TOPBIT   (1 << (WIDTH - 1))
#if (REFLECT_DATA == TRUE)
#undef  REFLECT_DATA
#define REFLECT_DATA(X)                 ((unsigned char) reflect((X), 8))
#else
#undef  REFLECT_DATA
#define REFLECT_DATA(X)                 (X)
#endif
#if (REFLECT_REMAINDER == TRUE)
#undef  REFLECT_REMAINDER
#define REFLECT_REMAINDER(X)    ((crc) reflect((X), WIDTH))
#else
#undef  REFLECT_REMAINDER
#define REFLECT_REMAINDER(X)    (X)
#endif
//Reorder the bits of a binary sequence, by reflecting
static unsigned long
reflect(unsigned long data, unsigned char nBits)
{
    unsigned long  reflection = 0x00000000;
    unsigned char  bit;
    /*
     * Reflect the data about the center bit.
     */
    for (bit = 0; bit < nBits; ++bit)
    {
        /*
         * If the LSB bit is set, set the reflection of it.
         */
        if (data & 0x01)
        {
            reflection |= (1 << ((nBits - 1) - bit));
        }
        data = (data >> 1);
    }
```

```
    return (reflection);
}       /* reflect() */
crc  crcTable[256];
//Populate the partial CRC lookup table.
void
crcInit(void)
{
    crc                     remainder;
    int                     dividend;
    unsigned char  bit;
    /*
     * Compute the remainder of each possible dividend.
     */
    for (dividend = 0; dividend < 256; ++dividend)
    {
        /*
         * Start with the dividend followed by zeros.
         */
        remainder = dividend << (WIDTH - 8);
        /*
         * Perform modulo-2 division, a bit at a time.
         */
        for (bit = 8; bit > 0; --bit)
        {
            /*
             * Try to divide the current data bit.
             */
            if (remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
        /*
         * Store the result into the table.
         */
        crcTable[dividend] = remainder;
    }
}
// Compute the CRC string
crc
crcFast(unsigned char const message[], int nBytes)
{
    crc             remainder = INITIAL_REMAINDER;
    unsigned char  data;
    int             byte;
    /*
     * Divide the message by the polynomial, a byte at a time.
     */
    for (byte = 0; byte < nBytes; ++byte)
    {
        data = REFLECT_DATA(message[byte]) ^ (remainder >> (WIDTH - 8));
        remainder = crcTable[data] ^ (remainder << 8);
    }
    /*
     * The final remainder is the CRC.
     */
    return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
}
```

Kesimpulan

Dari hasil pembuatan aplikasi enkripsi menggunakan AES *encryption* dengan penambahan *error detection* ini dapat diambil kesimpulan sebagai berikut:

1. *Key* yang digunakan pada aplikasi ini adalah berukuran 16 bit sehingga jika dihitung *possible key* kombinasi ada sekitar 65536 *key* kombinasi yang artinya jika dilakukan *brute force* maka *effort*-nya akan cukup mudah untuk di *crack*.
2. AES digunakan untuk melakukan enkripsi *message string* yang relatif pendek untuk enkripsi *string message* yang cukup panjang, maka aplikasi perlu di sesuaikan kembali karena akan mengakibatkan *memory overflow*.
3. Pengecekan *error detection* CRC digunakan untuk mengecek *integrity* dari sebuah data, dimana data kemungkinan *corrup* sehingga ada bit-bit yang hilang pada saat di transmisikan pada jaringan.

Daftar Pustaka

Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, "*Handbook of Applied Cryptography (5th Printing)*" CRC Press, 2001.

Arius, Dhony. "Kriptografi Keamanan Data dan Komuikasi", Graha Ilmu, Yogyakarta, 2008.

Munir, Rinaldi. "Kriptografi", Informatika, Bandung, 2006.

Rifki, Sadikin. "Kriptografi dan Keamanan Jaringan", Edisi Pertama, Penerbit Andi, Yogyakarta, 2012