STUDI MENGENAI ANUBIS *BLOCK CIPHER*

Tania Krisanty – 13504101

Laboratorium Ilmu Rekayasa dan Komputasi
Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jalan. Ganesha 10, Bandung
e-mail : if14101@students.if.itb.ac.id

Abstrak

Makalah ini membahas studi mengenai salah satu *Advanced Encryption Algorithm* (*AEA*) yaitu Anubis *Block Cipher*. *AEA* merupakan algoritma yang memenuhi suatu *Advanced Encryption Standard* (*AES*). Anubis *Block Cipher* merupakan sebuah algoritma kriptografi kunci simetri rancangan Vincent Rijmen dan Paulo S. L. M. Barreto yang diajukan dalam kompetisi *AEA*. Anubis *Block Cipher* beroperasi pada blok data sebesar 128 bit dan kunci sepanjang 32N bit (N = 4, ..., 10). Nama Anubis diberikan sesuai dengan nama dewa Mesir berwujud anjing yang dipercaya menjaga pintu kubur. Anubis *Block Cipher* diajukan ke kompetisi NESSIE yang mencari rancangan algoritma kriptografi yang memenuhi *AES*, untuk dikirim ke kompetisi berikut yang diselenggarakan oleh *The United States National Institute of Standards and Technology* (*NIST*).

Makalah ini juga memaparkan perkembangan kriptografi dari periode kriptografi klasik sampai kriptografi modern yang terkini, yaitu *AES*. Berbagai algoritma yang diajukan dan memenuhi persyaratan *AES* muncul, antara lain Rijndael, algoritma rancangan Vincent Rijmen dan John Daemen, yang dinilai paling memenuhi *AES*; dan Anubis, yang juga dirancang oleh Vincent Rijmen. Makalah ini juga mengkaji kedua algoritma tersebut dan menyajikan perbandingannya.

Kata kunci: *Advanced Encryption Algorithm, Advanced Encryption Standard*, Anubis *Block Cipher*, *Cipher Block*, *cryptography*, enkripsi, dekripsi.

1. Pendahuluan

Dalam kehidupan yang serba segera seperti sekarang, banyak sekali keperluan pertukaran data dalam waktu yang singkat. Oleh karenanya, manusia mulai meninggalkan segala prosedur pertukaran data secara konvensional seperti surat, telegram, maupun kurir yang memakan banyak waktu dan beralih ke prosedur pertukaran data secara modern yang tentunya lebih cepat prosesnya.

Perkembangan teknologi informasi dan komunikasi pun semakin mendukung pertukaran data secara modern, mulai dari adanya teknologi jaringan internet yang memungkinkan pertukaran data dari dan ke seluruh penjuru dunia, sampai perangkat telekomunikasi yang memungkinkan pertukaran data terjadi di mana dan kapan saja.

Kemajuan teknologi ini tentunya memberikan manfaat yang sangat besar bagi penggunanya, namun di lain pihak kemajuan teknologi ini semakin membuka celah bagi data tersebut untuk terekspos ke depan umum dengan mudah.

Data yang tidak berbahaya bila diketahui banyak pihak mungkin tidak terlalu dipermasalahkan walaupun dengan adanya celah tersebut. Sebaliknya dengan data yang perlu dijaga kerahasiaannya, misalnya data *Personal Identification Number* (*PIN*) dari sebuah kartu Anjungan Tunai Mandiri (ATM) atau kartu kredit, *electronic mail* (*e-mail*) yang berisi informasi penting, percakapan melalui alat komunikasi seluler, dan lain-lain. Jika ada pihak selain pengirim dan penerima data dapat memiliki akses ke data tersebut, ada kemungkinan data tersebut akan disalahgunakan bahkan dimanipulasi dengan cara dihilangkan atau diubah, Misalnya, data *PIN* dari kartu ATM suatu rekening di bank dapat digunakan kembali oleh pihak luar tersebut untuk melakukan berbagai transaksi atas nama pemilik kartu.

Penyalahgunaan data ini bisa memiliki bermacam-macam motif, dari keisengan semata sampai motif politik, bisnis, ekonomi, dan lain-lain. Apapun motifnya, penyalahgunaan data oleh pihak luar yang tidak berkepentingan akan merugikan pihak pengirim

dan penerima data, baik secara moral maupun material.

Oleh karena itu dibutuhkan suatu metode yang dapat menjamin keamanan dan keutuhan data selama proses pengirimannya. Cara termudah mungkin dengan memilih jalur yang sangat aman untuk mengirim data, misalnya dengan memanfaatkan jasa kurir yang sangat profesional dan terpercaya untuk menyampaikan data tersebut, namun cara ini bisa dikatakan kurang praktis karena memerlukan biaya yang relatif besar dan prosedur pengiriman data yang relatif lama. Selain itu, dengan memperbanyak frekuensi pemindahan data ke berbagai pihak yang berbeda akan meningkatkan kemungkinan dan ancaman terhadap keamanan dan keutuhan data.

Mengingat sulitnya mendapatkan jalur pertukaran data yang sangat aman, sebaiknya data tersebut dimanipulasi sedemikian rupa agar hanya bisa dimengerti oleh pihak-pihak yang berkepentingan. Selama dalam proses pengiriman data tersebut akan berwujud data yang telah disandikan sehingga tidak bermakna, setelah sampai di penerima barulah data tersebut dapat diterjemahkan dengan menggunakan 'kunci' yang dimiliki penerima. Jika tanpa sengaja data yang telah termanipulasi dapat diketahui pihak yang tidak berkepentingan, pihak lain tersebut tidak akan mampu menerjemahkan dan memahami makna data.

Untuk menerapkan metode ini diperlukan teknik pengubahan data bermakna (plainteks) menjadi data yang tidak bermakna (cipherteks), yang disebut juga teknik penyandian atau kriptografi.

Teknik penyandian data telah ada sejak 4000 tahun lalu. Beberapa contoh teknik tersebut adalah *hieroglyph*, simbol yang melambangkan kata-kata yang digunakan bangsa Mesir untuk berkomunikasi secara tulisan; *scytale*, teknik menulis pesan pada sebatang kayu dengan diameter tertentu yang dipakai bangsa Yunani; dan Enigma, nama mesin enkripsi buatan Nazi pada masa perang dunia ke dua. Pada masa-masa tersebut teknologi masih sederhana, belum ada atau belum banyak pemanfaatan komputer untuk membantu penyandian data.

Kriptografi pun masih sangat sederhana, berbasis karakter dan umumnya menggunakan sarana alat tulis saja. Operasi yang banyak dilakukan pada periode kriptografi klasik adalah operasi subtitusi, di mana suatu karakter plainteks disandikan dengan karakter lain sesuai perhitungan tertentu, atau operasi transposisi, di mana cipherteks diperoleh dengan mengubah posisi karakter dalam plainteks.

Operasi substitusi dan transposisi berbasis karakter ini masih sangat sederhana dan relatif mudah dipecahkan, terutama bila telah diketahui bahasa yang digunakan dan konteks atau lingkup dari plainteks. Selain itu umumnya karakter yang disandikan hanya alfabet, sehingga kunci pasti dapat diketahui dengan melakukan *exhaustive* atau *brute force key search*. Metode *exhaustive* atau *brute force key search* ini tidak akan memakan waktu terlalu lama, karena jumlah alfabet dapat dikatakan sedikit.

Seiring dengan perkembangan teknologi komputasi, kriptografi pun semakin berkembang. Operasi tidak lagi dilakukan terhadap karakter, melainkan representasi karakter dalam komputer *digital*, yaitu *binary digit* (bit). Sebuah karakter akan dinyatakan dalam delapan bit atau lebih. Karakter yang direpresentasikan dengan delapan bit atau satu *byte* termasuk dalam *Single-Byte Character Set* (*SBCS*). Karakter yang direpresentasikan dengan lebih dari delapan bit, misalnya enam belas bit atau dua *byte*, termasuk dalam *Multi Byte Character Set* (*MBCS*). Dalam konteks kriptografi modern, umumnya digunakan *SBCS*.

Kriptografi modern masih menggunakan gagasan kriptografi klasik, yaitu substitusi dan transposisi, perbedaannya adalah di sini operasi substitusi dan transposisi diterapkan pada bit-bit karakter. Untuk memperoleh hasil substitusi dan transposisi suatu karakter, banyak dilakukan operasi *exclusive or* (xor) secara *bitwise*, di mana setiap bit yang berkoresponden dari dua buah rangkaian bit di-xor-kan. Operasi xor ini relatif mudah untuk dipecahkan, karenanya operasi ini kerapkali dipadukan dengan operasi lain seperti pergeseran atau pertukaran bit.

Dalam dunia algoritma kriptografi berbasis bit dikenal metode *Cipher* Blok (*Block Cipher*) dan *Cipher* Aliran (*Stream Cipher*). Algoritma *Cipher* Blok beroperasi pada blok bit, enkripsi dan dekripsinya juga dilakukan pada blok bit. *Cipher* Block mudah diterapkan pada data yang utuh atau telah selesai proses pengirimannya, sebaliknya data yang pengirimannya secara bertahap akan sulit sekali dienkripsi atau didekripsi dengan algoritma ini. Untuk itulah dikembangkan algoritma *Cipher* Aliran yang sangat mendukung data *streaming*. Hal ini tentunya sangat efisien pada data berukuran besar yang memerlukan waktu pengiriman yang lama.

Standar algoritma kriptografi yang berlaku sejak 1977 adalah *Data Encryption Standard* (*DES*). *DES* merupakan kriptografi kunci simetri yang beroperasi pada panjang kunci eksternal 64-bit, dengan 8-bit paritas di dalamnya, dan tergolong *Cipher* Blok. Setiap blok mengalami permutasi awal (*IP*), 16 putaran proses enkripsi, dan inversi permutasi awal (IP^{-1}). Dalam setiap putaran digunakan kunci internal

yang dibangkitkan dari pergeseran dan permutasi kunci eksternal.

Dilihat dari prosesnya, *DES* banyak sekali memanipulasi kunci eksternal sehingga semakin kuat ketergantungan terhadap kunci eksternal tersebut. Didukung dengan jumlah bit kunci yang besar, *DES* menciptakan banyak kemungkinan kunci dan makin mempersulit pemecahannya walapun menggunakan teknik *exhaustive* atau *brute force key search*. Dengan panjang kunci 56-bit akan terdapat 2^{56} atau 72.057.594.037.927.936 kemungkinan kunci. Jika dilakukan serangan secara *exhaustive* atau *brute force key search* dengan menggunakan *processor* paralel, dalam satu detik dapat dikerjakan satu juta serangan. diperlukan 1142 tahun untuk menemukan kunci yang benar.

Seiring dengan berkembangnya teknologi perangkat keras dan meluasnya jaringan komputer, *DES* menjadi kurang aman karena kemampuan komputer melakukan komputasi rumit dan banyak dalam waktu singkat semakin meningkat. Panjangnya kunci eksternal tidak terlalu mempengaruhi kekuatan algoritma ini lagi. *DES* yang dikembangkan oleh *International Business Machines* (*IBM*) pada tahun 1972 ini mulai dirasakan perlu diganti dengan standar kriptografi yang baru.

Pada tahun 1997, *the United States National Institute of Standards and Technology* (*NIST*) mengumumkan bahwa sudah saatnya mengembangkan standar algoritma penyandian baru untuk menggantikan *DES* yang kelak diberi nama *Advanced Encryption Standard*. Algoritma *AES* merupakan algoritma kriptografi kunci simetrik yang tergolong *Cipher* Blok dan memproses blok data sebesar 128-bit dengan panjang kunci 128-bit (*AES*-128), 192-bit (*AES*-192), atau 256-bit (*AES*-256). Setelah melalui beberapa tahap seleksi, pada tahun 2000 algoritma Rijndael ditetapkan sebagai algoritma kriptografi *AES*.

Beberapa mode operasi yang dapat diterapkan pada algoritma kriptografi *AES* antara lain *Electronic Code Book* (*ECB*), *Cipher Block Chaining* (*CBC*), *Cipher Feedback* (*CFB*), dan *Output Feedback* (*OFB*). Implementasi *AES* dengan mode operasi *ECB*, *CBC*, *CFB*, dan *OFB* tentu saja memiliki kelebihan dan kekurangan tertentu dalam aspek tingkat keamanan data.

2.1. Panjang Kunci dan Ukuran Blok Rijndael

Rijndael mendukung panjang kunci 128 bit sampai 256 bit dengan jangkauan 32 bit. Panjang kunci dan ukuran blok dapat dipilih secara independen. Setiap blok dienkripsi dalam sejumlah putaran tertentu bergantung pada panjang kuncinya. Untuk panjang kunci 128-bit akan dilakukan sepuluh kali putaran, untuk panjang kunci 192-bit akan dilakukan 12 kali putaran, dan untuk panjang kunci 256-bit akan dilakukan 14 kali putaran.

Panjang kuncinya yang minimal sebesar 128-bit menyebabkan Rijndael tahan terhadap serangan *exhaustive* atau *brute force key search* dengan teknologi saat ini. Untuk panjang kunci 128-bit terdapat $2^{128} \approx 3{,}4 \times 10^{38}$ kemungkinan kunci. Untuk melakukan serangan terhadap algoritma ini tentunya diperlukan waktu yang sangat besar.

2.2. Algoritma Rijndael

Rijndael menggunakan operasi substitusi dan permutasi, diulang sejumlah putaran. Dalam setiap putaran digunakan kunci internal yang berbeda dengan kunci eksternal masukan dari pengguna.

Garis besar algoritma Rijndael yang beroperasi pada blok 128-bit dengan kunci 128-bit adalah sebagai berikut:

1. *AddRoundKey*: melakukan xor antara *state* awal (plainteks) dengan *cipher key*. Tahap ini disebut juga *initial round*.
2. Putaran sebanyak jumlah putaran - 1 kali. Proses yang dilakukan pada setiap putaran adalah:
   a. *ByteSub*: substitusi *byte* dengan menggunakan tabel substitusi (*S-box*).
   b. *ShiftRow:* pergeseran baris-baris *array state* secara *wrapping*.
   c. *MixColumn*: mengacak data di masing-masing kolom *array state*.
   d. *AddRoundKey:* melakukan xor antara *state* sekarang dengan *round key*.
3. *Final round*: proses pada putaran terakhir:
   a. *ByteSub*.
   b. *ShiftRow*.
   c. *AddRoundKey*.

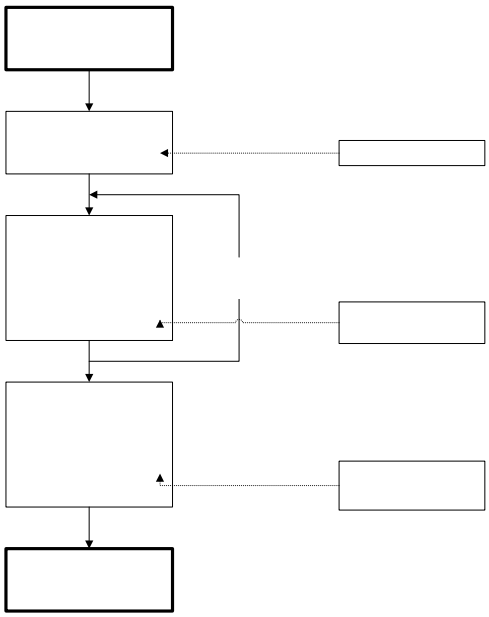
Algoritma Rijndael mempunyai tiga parameter, yaitu:

1. plainteks, berupa *array* berukuran 16 *byte* yang berisi data yang akan dienkripsi.

2. cipherteks, berupa *array* berukuran 16 *byte* yang berisi hasil enkripsi data.
3. key, berupa *array* berukuran 16 *byte*, yang berisi kunci *cipher*.

Selama proses enkripsi plainteks menjadi cipherteks, status data sekarang disimpan di dalam *array of byte* dua dimensi, state, yang berukuran NROWS x NCOLS. Elemen *array* state diakses dengan $S[r,c]$, di mana $0 \le r < 4$ dan $0 \le c < 4$.

Berikut tampilan diagram proses enkripsi Rijndael.



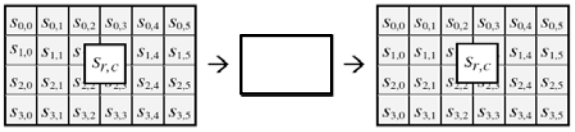
Berikut tabel substitusi (*S-box*) yang digunakan pada proses ByteSub.

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

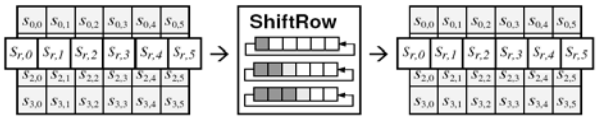
Putaran Standar

1-ByteSub
2-ShiftRow
3-MixColumn
4-AddRoundKey

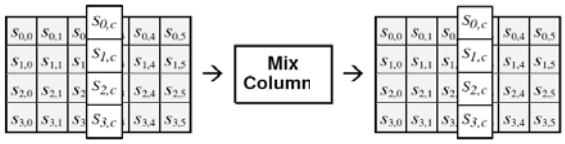
Berikut transformasi ByteSub .



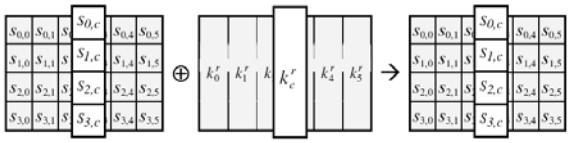
Berikut transformasi ShiftRow.



Berikut transformasi MixColumn.



Berikut transformasi AddRoundKey.



3.1. Panjang Kunci dan Ukuran Blok Anubis

Pada enkripsi dengan Anubis *Block Cipher* panjang kunci dan ukuran blok plainteks dapat dipilih secara independen. Anubis mampu beroperasi pada blok plainteks sebesar 128-bit dan panjang kunci bervariasi dari 128-bit sampai 320-bit dengan jangkauan 32-bit.

Setiap blok plainteks dienkripsi sejumlah putaran tertentu bergantung pada panjang kuncinya, minimal 12 putaran untuk panjang kunci 128-bit, bertambah satu putaran untuk setiap penambahan kunci sebesar 32-bit. Setiap putaran terdiri dari enam belas 8-bit kali 8-bit tabel substitusi (*S-box*), sebuah transformasi linear (transposisi matriks diikuti perkalian dengan konstanta matriks difusi), dan penambahan kunci putaran. *S-box* dan matriks difusi dipilih sedemikian rupa agar enkripsi dan dekripsi hanya perlu memerlukan operasi yang sama, kecuali dalam kunci internalnya.

⊕ Kunci Cipher

$R - 1$
Putaran

⊕ Kunci Putaran ke R

3.2. Algoritma Anubis

Berikut *header* dan implementasi algoritma Anubis *Cipher Block* dalam bahasa C:

```
/* File : nessie.h */


#ifndef PORTABLE_C__
#define PORTABLE_C__

#include <limits.h>


/**
 * Definisi tipe bilangan integer
 * yang akan digunakan
 *
 * u8, bertipe unsigned integer,
 * minimal 8 bit, ekivalen dengan
 * unsigned char.
 * u16, bertipe unsigned integer,
 * minimal 16 bit.
 * u32, bertipe unsigned integer,
 * minimal 32 bit.
 * s8, s16, s32 bertipe
 * signed integer, berkoresponden
 * dengan u8, u16, u32.
 */

typedef signed char s8;
typedef unsigned char u8;

#if UINT_MAX >= 4294967295UL

typedef signed short s16;
typedef signed int s32;
typedef unsigned short u16;
typedef unsigned int u32;

#define ONE32   0xffffffffU

#else

typedef signed int s16;
typedef signed long s32;
typedef unsigned int u16;
typedef unsigned long u32;

#define ONE32   0xffffffffUL

#endif

#define ONE8    0xffU
#define ONE16   0xffffU

#define T8(x)   ((x) & ONE8)
#define T16(x)  ((x) & ONE16)
#define T32(x)  ((x) & ONE32)

/**
 * U8TO32_BIG(c) berfungsi
 * mengembalikan nilai pada array
 * unsigned char yang ditunjuk oleh
 * c ke dalam notasi 32-bit big
 * endian
 */


#define U8TO32_BIG(c)
(((u32)T8(*(c)) << 24) |
((u32)T8(*((c) + 1)) << 16) \

((u32)T8(*((c) + 2)) << 8) |
((u32)T8(*((c) + 3))))

/**
 * U8TO32_LITTLE(c) berfungsi
 * mengembalikan nilai pada array
 * unsigned char yang ditunjuk oleh
 * c ke dalam notasi 32-bit little
 * endian
 */

#define U8TO32_LITTLE(c)
(((u32)T8(*(c))) | ((u32)T8(*((c) +
1)) << 8) \ (u32)T8(*((c) + 2)) <<
16) | ((u32)T8(*((c) + 3)) << 24))

/**
 * U8TO32_BIG(c, v) berfungsi
 * menyimpan nilai 32-bit v dalam
 * notasi big endian ke dalam array
 * unsigned char yang ditunjuk oleh
 * c
 */

#define U32TO8_BIG(c, v) do
{
  u32 x = (v);
  u8 *d = (c);
  d[0] = T8(x >> 24);
  d[1] = T8(x >> 16);
  d[2] = T8(x >> 8);
  d[3] = T8(x);
} while (0)

/**
 * U8TO32_LITTLE(c, v) berfungsi
 * menyimpan nilai 32-bit v dalam
 * notasi little endian ke dalam
```

```
 * array unsigned char yang ditunjuk
 * oleh c
 */

#define U32TO8_LITTLE(c, v) do
{
  u32 x = (v);
  u8 *d = (c);
  d[0] = T8(x);
  d[1] = T8(x >> 8);
  d[2] = T8(x >> 16);
  d[3] = T8(x >> 24);
} while (0)


/**
 * ROTL32(v, n) berfungsi
 * mengembalikan nilai v dalam 32
 * bit unsigned setelah pergeseran n
 * bit ke kiri secara wrapping.
 */

#define ROTL32(v, n)
(T32((v) << (n)) | ((v) >> (32 -
(n))))

/**
 * Definisi spesifik lainnya
 */

#define MIN_N          4
#define MAX_N          10
#define MIN_ROUNDS     (8 + MIN_N)
#define MAX_ROUNDS     (8 + MAX_N)
#define MIN_KEYSIZEB   (4*MIN_N)
#define MAX_KEYSIZEB   (4*MAX_N)
#define BLOCKSIZE      128
#define BLOCKSIZEB     (BLOCKSIZE/8)

/**
 * Macro KEYSIZEB harus didefinisi
 * ulang untuk setiap ukuran kunci.
 * Ukuran yang valid antara lain 16
 * byte, 20 byte, 24 byte, 28 byte,
 * 32 byte, 36 byte, and 40 byte.
 */

#define KEYSIZEB        16

typedef struct NESSIEstruct
{
  int keyBits;
  /**
   * keyBits harus sudah
   * diinisialisasi sebelum
   * pemanggilan NESSIEkeysetup call
   */
  int R;
  u32 roundKeyEnc[MAX_ROUNDS+1][4];
  u32 roundKeyDec[MAX_ROUNDS+1][4];
} NESSIEstruct;

/**
 * Pembangkitan kunci enkripsi dan
 * dekripsi dari kunci masukan
 *
 * @param key
 * key merupakan kunci sepanjang
 * 32N-bit (N = 4, ..., 10)
 *

 * @param structpointer
 * structpointer merupakan
 * pointer ke struktur penyimpan
 * kunci yang telah diekspansi
 */

void NESSIEkeysetup(const unsigned
char * const key, struct
NESSIEstruct * const structpointer);

/**
 * Enkripsi blok data
 *
 * @param structpointer
 * structpointer merupakan kunci
 * yang telah diekspansi
 *
 * @param plaintext
 * plaintext merupakan blok data
 * yang akan dienkripsi
 *
 * @param ciphertext
 * ciphertext merupakan hasil
 * enkripsi blok data
 */

void NESSIEencrypt(const struct
NESSIEstruct * const structpointer,
const unsigned char * const
plaintext, unsigned char * const
ciphertext);

/**
 * Dekripsi blok data
 *
 * @param structpointer
 * structpointer merupakan kunci
 * yang telah diekspansi
 *
 * @param ciphertext
 * ciphertext merupakan blok data
 * yang akan didekripsi
```

```
 *
 * @param plaintext
 * plaintext merupakan hasil
 * dekripsi blok data
 */

void NESSIEdecrypt(const struct
NESSIEstruct * const structpointer,
const unsigned char * const
ciphertext, unsigned char * const
plaintext);

#endif   /* PORTABLE_C__ */

/* End of file : nessie.h */
/* File : Anubis.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nessie.h"

/**
 * Pengisian enam buah tabel
 * enkripsi dengan 256 buah
 * bilangan heksadesimal delapan
 * digit, dalam makalah ini
 * tidak ditampilkan isi
 * tabelnya
 */

static const u32 T0[256] = {};

static const u32 T1[256] = {};

static const u32 T2[256] = {};

static const u32 T3[256] = {};

static const u32 T4[256] = {};

static const u32 T5[256] = {};

/**
 * Pengisian tabel konstanta
 * putaran, dalam makalah ini
 * tidak ditampilkan isinya
 */

static const u32 rc[] = {};

/**
 * Pembangkitan kunci enkripsi
 * dan dekripsi dari kunci
 * masukan
 *
 * @param key
 * key merupakan kunci sepanjang
 * 32N-bit (N = 4, ..., 10)
 *
 * @param structpointer
 * structpointer merupakan
 * pointer ke struktur penyimpan
 * kunci yang telah diekspansi
 */

void NESSIEkeysetup(const unsigned
char * const key, struct
NESSIEstruct * const structpointer)
{

  int N, R, i, pos, r;
  u32 kappa[MAX_N];
  u32 inter[MAX_N];

structpointer->keyBits = KEYSIZEB*8;

  /**
   * Penentuan panjang
   * parameter N, dengan asumsi
   * nilai N valid
   */

N = structpointer->keyBits >> 5;

  /**
   * Penentukan jumlah putaran
   * dari panjang kunci masukan
   */

structpointer->R = R = 8 + N;

  /*
   * Pemetaan kunci enkripsi ke
   * initial key state (mu)
   */

for (i = 0, pos = 0; i < N; i++, pos
+= 4)
{
  kappa[i] = (key[pos    ] <<
  24) ^ (key[pos + 1] << 16) ^
  (key[pos + 2] <<  8) ^
  (key[pos + 3]);
}

  /**
   * Pembangkitan R + 1 kunci
   * putaran
   */

for (r = 0; r <= R; r++)
{
```

```
  u32 K0, K1, K2, K3;

  /**
   * Pembangkitan kunci putaran
   * ke-r K^r
   */

  K0 = T4[(kappa[N - 1] >> 24)];
  K1 = T4[(kappa[N - 1] >> 16) &
  0xff];
  K2 = T4[(kappa[N - 1] >>  8) &
  0xff];
  K3 = T4[(kappa[N - 1]      ) &
  0xff];

for (i = N - 2; i >= 0; i--)
{
  K0 = T4[(kappa[i] >> 24)] ^
  (T5[(K0 >> 24)] &0xff000000U)
  ^ (T5[(K0 >> 16) & 0xff] &
  0x00ff0000U) ^ (T5[(K0 >>  8)
  & 0xff] & 0x0000ff00U) ^
  (T5[(K0      ) & 0xff] &
  0x000000ffU);

  K1 = T4[(kappa[i] >> 16) &
  0xff] ^ (T5[(K1 >> 24)] &
  0xff000000U) ^ (T5[(K1 >> 16)
  & 0xff] & 0x00ff0000U) ^
  (T5[(K1 >>  8) & 0xff] &
  0x0000ff00U) ^ (T5[(K1) &
  0xff] & 0x000000ffU);

  K2 = T4[(kappa[i] >>  8) &
  0xff] ^ (T5[(K2 >> 24)] &
  0xff000000U) ^ (T5[(K2 >> 16)
  & 0xff] & 0x00ff0000U) ^
  (T5[(K2 >> 8) & 0xff] &
  0x0000ff00U) ^
  (T5[(K2) & 0xff] &
  0x000000ffU);

  K3 = T4[(kappa[i]) & 0xff] ^
  (T5[(K3 >> 24)] &0xff000000U)
  ^ (T5[(K3 >> 16) & 0xff] &
  0x00ff0000U) ^ (T5[(K3 >>  8)
  & 0xff] & 0x0000ff00U) ^
  (T5[(K3      ) & 0xff] &
  0x000000ffU);
}

  /* Berikut kode yang digunakan
     untuk tabel U yang
     berukuran besar:

     K0 = K1 = K2 = K3 = 0;
     for (i = 0; i < N; i++)
     {
       K0 ^= U[i][(kappa[i] >>
       24)];

       K1 ^= U[i][(kappa[i] >>
       16) & 0xff];

       K2 ^= U[i][(kappa[i] >>
       8) & 0xff];

       K3 ^= U[i][(kappa[i]) &
       0xff];
     }
   */
structpointer->roundKeyEnc[r][0] =
K0;

structpointer->roundKeyEnc[r][1] =
K1;

structpointer->roundKeyEnc[r][2] =
K2;

structpointer->roundKeyEnc[r][3] =
K3;

/**
 * Penghitungan nilai
 * kappa^{r+1} dari kappa^r:
 */

if (r == R)
{
  break;
}

for (i = 0; i < N; i++)
{
  int j = i;
  inter[i]  = T0[(kappa[j--] >>
  24)];
  if (j < 0)
  {
    j = N - 1;
  }
  inter[i] ^= T1[(kappa[j--] >>
  16) & 0xff];
  if (j < 0)
  {
    j = N - 1;
  }
  inter[i] ^= T2[(kappa[j--] >>
  8) & 0xff];
  if (j < 0)
  {
    j = N - 1;
```

```
  }
  inter[i] ^= T3[(kappa[j]) &
  0xff];
}

kappa[0] = inter[0] ^ rc[r];

for (i = 1; i < N; i++)
{
  kappa[i] = inter[i];
}
}

  /**
   * Pembangkitan kunci inverse:
   * K'^0 = K^R, K'^R
   * = K^0, K'^r = theta(K^{R
   * r}):
   */

for (i = 0; i < 4; i++)
{
  structpointer->
  roundKeyDec[0][i] =
  structpointer->
  roundKeyEnc[R][i];

  structpointer->
  roundKeyDec[R][i] =
  structpointer->
  roundKeyEnc[0][i];
}

for (r = 1; r < R; r++)
{
  for (i = 0; i < 4; i++)
  {
    u32 v = structpointer->
    roundKeyEnc[R - r][i];
    structpointer->
    roundKeyDec[r][i] =
    T0[T4[(v >> 24)] & 0xff] ^
    T1[T4[(v >> 16) & 0xff] &
    0xff] ^ T2[T4[(v >> 8) &
    0xff] & 0xff] ^ T3[T4[(v) &
    0xff] & 0xff];
  }
}
}

/**
 * Enkripsi atau dekripsi blok data
 *
 * @param block
 * block merupakan blok data yang
 * akan dienkripsi atau didekripsi
 *
 * @param roundKey
 * roundkey merupakan kunci yang
 * akan digunakan
 *
 * @param R
 * R merupakan banyaknya putaran
 */

static void crypt(const u8
plaintext[/*16*/], u8
ciphertext[/*16*/], const u32
roundKey[MAX_ROUNDS + 1][4], int R)
{
  int i, pos, r;
  u32 state[4];
  u32 inter[4];

  /**
   * Pemetaan blok plainteks ke
   * state cipher (mu) dan
   * penambahan kunci putaran awal
   * (sigma[K^0])
   */

  for (i = 0, pos = 0; i < 4; i++,
  pos += 4)
  {
    state[i] = (plaintext[pos] <<
    24) ^ (plaintext[pos + 1] << 16)
    ^ (plaintext[pos + 2] <<  8) ^
    (plaintext[pos + 3]) ^
    roundKey[0][i];
  }

  /**
   * Putaran dari 1 sampai R - 1
   */

  for (r = 1; r < R; r++)
  {
    inter[0] = T0[(state[0] >> 24)]
    ^ T1[(state[1] >> 24)] ^
    T2[(state[2] >> 24)] ^
    T3[(state[3] >> 24)] ^
    roundKey[r][0];
    inter[1] = T0[(state[0] >> 16) &
    0xff] ^ T1[(state[1] >> 16) &
    0xff] ^ T2[(state[2] >> 16) &
    0xff] ^ T3[(state[3] >> 16) &
    0xff] ^ roundKey[r][1];
    inter[2] = T0[(state[0] >>  8) &
    0xff] ^ T1[(state[1] >>  8) &
    0xff] ^ T2[(state[2] >>  8) &
    0xff] ^ T3[(state[3] >>  8) &
    0xff] ^ roundKey[r][2];
    inter[3] = T0[(state[0]) & 0xff]
    ^ T1[(state[1]) & 0xff] ^
```

```
  T2[(state[2]) & 0xff] ^
  T3[(state[3]      ) & 0xff] ^
  roundKey[r][3];
  state[0] = inter[0];
  state[1] = inter[1];
  state[2] = inter[2];
  state[3] = inter[3];
}

/**
 * Putaran ke R atau terakhir
 */

inter[0] = (T0[(state[0] >> 24)] &
0xff000000U) ^ (T1[(state[1] >>
24)] & 0x00ff0000U) ^
(T2[(state[2] >> 24)] &
0x0000ff00U) ^ (T3[(state[3] >>
24)] & 0x000000ffU) ^
roundKey[R][0];
inter[1] = (T0[(state[0] >> 16) &
0xff] & 0xff000000U) ^
(T1[(state[1] >> 16) & 0xff] &
0x00ff0000U) ^
(T2[(state[2] >> 16) & 0xff] &
0x0000ff00U) ^ (T3[(state[3] >>
16) & 0xff] & 0x000000ffU) ^
roundKey[R][1];
inter[2] = (T0[(state[0] >>  8) &
0xff] & 0xff000000U) ^
(T1[(state[1] >>  8) & 0xff] &
0x00ff0000U) ^
(T2[(state[2] >>  8) & 0xff] &
0x0000ff00U) ^
(T3[(state[3] >>  8) & 0xff] &
0x000000ffU) ^
roundKey[R][2];
inter[3] = (T0[(state[0]) & 0xff]
& 0xff000000U) ^ (T1[(state[1]) &
0xff] & 0x00ff0000U) ^
(T2[(state[2]) & 0xff] &
0x0000ff00U) ^ (T3[(state[3]) &
0xff] & 0x000000ffU) ^
roundKey[R][3];

/**
 * Pemetaan state cipher ke blok
 * cipherteks(mu^{-1})
 */

for (i = 0, pos = 0; i < 4; i++,
pos += 4)
{
  u32 w = inter[i];
  ciphertext[pos] = (u8)(w >> 24);
  ciphertext[pos + 1] = (u8)(w >>
  16);
    ciphertext[pos + 2] = (u8)(w >>
    8);
    ciphertext[pos + 3] = (u8)(w);
  }
}

/**
 * Enkripsi blok data
 *
 * @param structpointer
 * structpointer merupakan kunci
 * yang telah diekspansi
 *
 * @param plaintext
 * plaintext merupakan blok data
 * yang akan dienkripsi
 *
 * @param ciphertext
 * ciphertext merupakan hasil
 * enkripsi blok data
 */

void NESSIEencrypt(const struct
NESSIEstruct * const structpointer,
const unsigned char * const
plaintext, unsigned char * const
ciphertext)
{
  crypt(plaintext, ciphertext,
  structpointer->roundKeyEnc,
  structpointer->R);
}

/**
 * Dekripsi blok data
 *
 * @param structpointer
 * structpointer merupakan kunci
 * yang telah diekspansi
 *
 * @param ciphertext
 * ciphertext merupakan blok data
 * yang akan didekripsi
 *
 * @param plaintext
 * plaintext merupakan hasil
 * dekripsi blok data
 */

void NESSIEdecrypt(const struct
NESSIEstruct * const structpointer,
const unsigned char * const
ciphertext, unsigned char * const
plaintext)
{
  crypt(ciphertext, plaintext,
  structpointer->roundKeyDec,
```

```
  structpointer->R);
}

/* End of file : Anubis.c */
```

5. Analisis

Dari algoritma di atas terlihat langkah-langkah enkripsi dengan Anubis *Block Cipher*.

Diawali dengan pengisian tabel-tabel enkripsi yang masing-masing berisi 256 buah bilangan heksadesimal sepanjang delapan digit, lalu mengisi konstanta putaran.

Pada prosedur `NESSIEkeysetup` dibangkitkan kunci internal dari kunci eksternal yang dimasukkan pengguna, dilakukan dengan tahap-tahap berikut, menentukan `N` dari kunci sepanjang 32N-bit, menentukan banyak putaran (`R`) dengan menjumlahkan delapan dan `N`, memetakan kunci *cipher* ke *initial key state* (`mu`), membangkitkan kunci putaran sebanyak `R` + 1, lalu membangkitkan kunci *inverse* dengan rumus K'^0 = K^R, K'^R = K^0, K'^r = theta(K^{R-r}).

Untuk mengenkripsi atau mendekripsi blok data digunakan prosedur yang sama, yaitu `crypt`. Prosedur inilah yang akan menentukan apakah blok data tersebut akan dienkripsi atau didekripsi berdasarkan `roundkey` dari parameter masukan prosedur ini. Prosedur `crypt` ini diawali dengan memetakan blok plainteks ke `mu` dan menambahkan kunci putaran awal, lalu melakukan proses sebanyak `R` – 1 putaran. Pada putaran terakhir, yaitu putaran ke R, operasi melibatkan suatu bilangan baru yang bervariasi dari 255, 65280, 16711680, sampai 4278190080. Langkah terakhir adalah memetakan *state cipher* ke blok cipherteks.

Operasi yang banyak digunakan dalam algoritma Anubis antara lain perpangkatan dan *bitwise and*.

Algoritma Anubis secara garis besar hampir sama dengan algoritma Rijndael, hanya saja terdapat perbedaan dalam beberapa hal.

Berikut tabel perbandingan algoritma Rijndael dan algoritma Anubis.

	Rijndael	Anubis
Ukuran blok plainteks (dalam bit)	128, 192, 256	128
Panjang kunci (dalam bit)	128, 192, 256	128, 160, 192, 224, 256, 288, 320
Jumlah putaran	10, 12, 14	12, 13, 14, 15, 16, 17, 18
Pembangkitan kunci	algoritma	evolusi kunci (variasi dari fungsi putaran) dan pemilihan kunci (proyeksi)
GF(2^8) *reduction polynomial*	$x^8 + x^4 + x^3 + x + 1$ (0x11B)	$x^8 + x^4 + x^3 + x^2 + x + 1$ (0x11D)
Asal S-box	pemetaan u → u^{-1} melalui GF (2^8), ditambah dengan transformasi *affine*	involusi semi acak
Asal konstanta jumlah putaran	polinomial s x^i dari GF (2^8)	*succesive entries* dari S-*box*

Dalam pengujian, sesuai hasil tes vektor yang ditampilkan oleh *NIST*, Anubis dan Rijndael sama-sama menunjukkan kemampuan mengenkripsi dan mendekripsi kembali data dengan baik dan benar. Serangan paling efisien atas kedua jenis algoritma ini hanyalah *exhaustive* atau *brute force key search*, walaupun algoritmanya diketahui secara publik.

Berikut ini contoh beberapa vektor tes algoritma Rijndael yang beroperasi pada blok plainteks sebesar 128-bit dan panjang kunci 128-bit.

vektor ke 0:
kunci =
80000000000000000000000000000000
cipherteks =
0EDD33D3C621E546455BD8BA1418BEC8
plainteks =
00000000000000000000000000000000
cipherteks terdekripsi =
00000000000000000000000000000000

vektor ke 1:
kunci =
64646464646464646464646464646464
plainteks =
A22377C4945F399225BABDE72B4C50A2
cipherteks =

64646464646464646464646464646464
plainteks terenkripsi =
64646464646464646464646464646464

vektor ke 2:
kunci =
000102030405060708090A0B0C0D0E0F
plainteks =
762A5AB50929189CEFDB99434790AAD8
cipherteks =
00112233445566778899AABBCCDDEEFF
plainteks terenkripsi =
00112233445566778899AABBCCDDEEFF

Berikut ini contoh beberapa vektor tes algoritma Anubis yang beroperasi pada blok plainteks sebesar 128-bit dan panjang kunci 128-bit.

vektor ke 0:
kunci =
80000000000000000000000000000000
cipherteks =
F06860FC6730E818F132C78AF4132AFE
plainteks =
00000000000000000000000000000000
cipherteks terdekripsi =
00000000000000000000000000000000

vektor ke 1:
kunci =
64646464646464646464646464646464
plainteks =
B97D356D7DD92BC2666B3941A0216C8C
cipherteks =
64646464646464646464646464646464
plainteks terenkripsi =
64646464646464646464646464646464

vektor ke 2:
kunci =
000102030405060708090A0B0C0D0E0F
plainteks =
82AA04A54EF9B29A7F0D6B80F5D24089
cipherteks =
00112233445566778899AABBCCDDEEFF
plainteks terenkripsi =
00112233445566778899AABBCCDDEEFF

Berikut ini contoh beberapa vektor tes algoritma Anubis yang beroperasi pada blok plainteks sebesar 128-bit dan panjang kunci 256-bit.

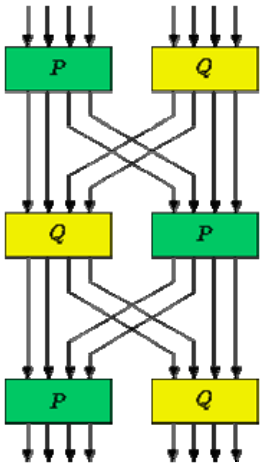
vektor ke 0:
kunci =
8000000000000000000000000000000000000000000000000000000000000000
cipherteks =
E086AC456B3CE513EDF5DFDDD63B7193
plainteks =
00000000000000000000000000000000
cipherteks terdekripsi =
00000000000000000000000000000000

vektor ke 1:
kunci =
6464646464646464646464646464646464646464646464646464646464646464
cipherteks =
B4F659AFD508171D4F06D2C26B57198D
plainteks =
6464646464646464646464646464646464646464646464646464646464646464
cipherteks terenkripsi =
6464646464646464646464646464646464

vektor ke 2:
kunci =
000102030405060708090A0B0C0D0E0F
101112131415161718191A1B1C1D1E1F
plainteks =
E8BB515890F2B4DEEA97ECD43D76BC10
cipherteks =
00112233445566778899AABBCCDDEEFF
plainteks terenkripsi =
00112233445566778899AABBCCDDEEFF

Untuk meningkatkan performansi Anubis *Block Cipher* dan memudahkan implemantasinya dalam perangkat keras, perancangnya menambahkan struktur *tweak* yang menggantikan *S-box*. Struktur *tweak* di sini dibangkitkan secara rekursif dan semi acak, menghasilkan 4-bit kali 4-bit mini*box*.

Berikut tampilan struktur *tweak* pada Anubis *Block Cipher*.



Berikut tampilan mini-*box* P dan mini-*box* Q

Mini-*box* P

u	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
P(u)	3	F	E	0	5	4	B	C	D	A	9	6	7	8	2	1

Mini-*box* Q

u	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Q(u)	9	E	5	6	A	2	3	C	F	0	4	D	7	B	1	8

6. Kesimpulan

Kesimpulan yang dapat diambil dari studi tentang Anubis *Block Cipher* ini adalah:

1. Anubis *Block Cipher* memiliki struktur perulangan proses yang bergantung pada panjang kunci masukan dari pengguna. Hal ini akan mempersulit penyerangan karena kesalahan sedikit pada kunci masukan akan menyebabkan banyak perubahan pada proses dan hasil akhir.
2. Serangan yang paling efisien terhadap Anubis *Block Cipher* adalah teknik *exhaustive* atau *brute force key search*.
3. Satu-satunya serangan yang paling efisien terhadap Anubis *Block Cipher* jika diketahui plainteks dan cipherteks yang berkoresponden adalah mencari kuncinya menggunakan teknik *exhaustive* atau *brute force key search*.
4. Besarnya usaha yang diperlukan untuk memecahkan kunci dari suatu cipherteks terenkripsi dengan Anubis *Block Cipher* sangat bergantung pada panjang kunci. Untuk mencoba segala kemungkinan kunci, diperlukan 2^{m-1} kali aplikasi of Anubis *Cipher Block* untuk kunci sepanjang *m*-bit. Ketergantungan yang sangat besar terhadap kunci ini yang menjadikan Anubis masih memiliki kemungkinan yang cukup besar untuk diserang secara *exhaustive* atau *brute force key search*.
5. Teknik enkripsi dengan Rijndael dan Anubis memiliki bebapa kemiripan, antara lain adanya struktur perulangan, tabel substitusi, pembangkitan kunci internal. Namun kemiripan ini hanya struktural, karena operasi matematika yang dilakukan di dalamnya berbeda.

DAFTAR PUSTAKA

[1] http://planeta.terra.com.br/informatica.
Tanggal akses: 18 September 2006.

[2] http://www.esat.kuleuven.ac.be.
Tanggal akses: 26 September 2006.

[4] http://www.nist.gov.
Tanggal akses: 18 September 2006.

[4] Munir, Rinaldi. (2004). Diktat Kuliah IF5054 Kriptografi. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung,