BAB II
TINJAUAN PUSTAKA

2.1 Kriptografi

Kriptografi berasal dari bahasa Yunani. Menurut bahasanya istilah tersebut terdiri dari kata *kripto* dan *graphia*. *Kripto* berarti *secret* (rahasia) dan *graphia* berarti *writing* (tulisan) (Ariyus, 2007). Kriptografi sudah mulai digunakan 4000 tahun yang lalu oleh orang mesir menggunakan *hieroglyf* tidak standar. Kemudian pada zaman romawi kuno, kriptografi juga digunakan oleh Julius Caesar untuk mengirim suatu pesan kepada jenderalnya di medan perang. Agar pesan tersebut tidak diketahui oleh orang lain saat dikirimkan lewat kurir, maka Julius Caesar mengacak susunan *alphabet* dari a, b, c dst, yaitu misalnya a menjadi d, b menjadi e dan seterusnya. Dan tentu saja sang jenderal sudah diberi tahu terlebih dahulu mengenai membaca pesan yang teracak tersebut sehingga dia bisa mengartikannya dengan mudah. Dan kemudian apa yang dilakukan Julius Caesar tersebut terkenal dengan *Caesar Chiper* hingga sekarang.

Pada dasarnya, kriptografi terdiri dari beberapa komponen seperti :

1. Pesan.
   Pesan di sini bisa berupa data atau informasi yang dikirim (melalui kurir, komunikasi data, dan lain-lain) atau yang disimpan di dalam media perekaman.

2. *Plaintext*.
   Plaintext merupakan suatu pesan yang masih asli, masih bermakna dan bisa diartikan dengan jelas. *Plaintext* ini yang nantinya akan diolah oleh algoritma kriptografi.

3. *Ciphertext*.
   Ciphertext merupakan suatu kode-kode yang tidak dapat dimengerti secara langsung karena *ciphertext* ini merupakan *plaintext* yang sudah melalui proses enkripsi.

4. *Key*.
   Key adalah kata kunci yang digunakan untuk melakukan proses enkripsi dan dekripsi.

5. Enkripsi.
   Enkripsi merupakan inti dari kriptografi. Karena pada tahap inilah sebuah *plaintext* akan diubah menjadi kode-kode yang tidak dimengerti yang disebut *ciphertext* dengan menggunakan *key*.
6. Dekripsi.
   Dekripsi merupakan kebalikan dari enkripsi. Dimana pada tahap ini sebuah *ciphertext* akan diubah kembali menjadi *plaintext* dengan menggunakan *key* juga.

2.2 Algoritma Kriptografi

Algoritma kriptografi dibagi menjadi dua bagian berdasarkan *key* yang dipakainya, yaitu :

1. Algoritma Simetri.
   Algoritma ini sering juga disebut algoritma klasik karena memakai *key* yang sama untuk kegiatan enkripsi dan dekripsinya. Pengiriman pesan menggunakan algoritma tersebut mengharuskan si penerima pesan untuk mengetahui *key* dari pesan tersebut agar bisa mendekripsi pesan yang dikirim. Keamanan dari pesan yang menggunakan algoritma itu tergantung pada *key*. Jika *key* tersebut diketahui oleh orang lain maka orang tersebut bisa melakukan enkripsi dan dekripsi terhadap pesan tersebut. Algoritma yang memakai *key* simetri diantaranya adalah :
   - *Data Encryption Standart* (DES)
   - RC2,RC4,RC5,RC6
   - *International Data Encryption Algoritm* (IDEA)
   - *Advanced Encryption Standart* (AES)
   - *One Time Pad* (OTP)
   - A5
   - Dan lain sebagainya.

2. Algoritma Asimetri.
   Algoritma asimetri sering juga disebut algoritma kunci publik. Artinya, *key* yang digunakan untuk melakukan enkripsi dan dekripsinya berbeda. Pada algoritma asimetri, *key* terbagi menjadi dua bagian :
   - Kunci Umum (*public key*) : kunci yang boleh diketahui semua orang (dipublikasikan)

- Kunci pribadi (*private key*) : kunci yang dirahasiakan (hanya boleh oleh satu orang yang bersangkutan)

Key-key tersebut saling berhubungan satu dengan yang lainnya. Penggunaan *public key* memungkinkan seseorang untuk bisa mengenkripsi pesan tetapi tidak bisa mendekripsinya. Hanya orang yang mempunyai *private key* yang bisa mendekripsi pesan tersebut. Algoritma asimetris bisa mengirim pesan dengan lebih aman ketimbang algoritma simetris. Sebagai contoh, Bob mengirim pesan dengan menggunakan algoritma asimetris kepada Alice. Untuk itu hal yang harus dilakukan adalah :

- Bob memberitahu kunci umumnya kepada Alice
- Alice mengenkripsi pesan menggunakan kunci umum Bob
- Bob mendekripsi pesan dari Alice dengan kunci pribadinya.

Algoritma yang memakai *key* asimetri diantaranya adalah : (Ariyus, 2008)

- *Digital Signature Algoritm* (DSA)
- RSA
- *Diffie-Hellman* (DH)
- *Elliptic Curve Cryptography* (ECC)
- *Cryptography Quantum*

2.3 Landasan Matematika Kriptografi

2.3.1 Operasi XOR

Exclusive OR dan kadang ditulis dengan XOR atau simbol adalah salah satu cara untuk melakukan operasi pada *string* binari. Operator xor Ini merupakan suatu penambahan *modulo* 2 dan digambarkan sebagai berikut 0 0 = 0, 0 1 = 1, 1 0 = 1, 1 1 = 0. Dan tabel operator XOR dapat dilihat pada tabel 2.1.

Tabel 2.1 Tabel Operator Xor
(http://www.accs.com/p_and_p/RAID/Definitions.html)

XOR	**Input 1**		
		0	1
Input 2	0	0	1
	1	1	0

Operasi sederhana ini memberikan cara pengkombinasian dua *bit string* dengan panjang yang sama. Sebagai contoh, mengoperasikan 10011 □ 11001, maka pertama kali yang harus dilakukan adalah meng-XOR-kan bit pertama yaitu 1 □ 1= 0, kemudin bit kedua dan seterusnya sampai semua *bit* telah di-XOR-kan. Kelanjutan dari cara ini akan menghasilkan 01010 (Ariyus, 2008).

Sebagai contoh, string "Wiki" jika ditulis dalam format ASCII 8 *bit* menjadi 01010111 01101001 01101011 01101001 dapat diproses dengan suatu operasi XOR misalnya 11110011 adalah sebagai berikut :

01010111	01101001	01101011	01101001
11110011	11110011	11110011	11110011
			(XOR)
10100100	10011010	10011000	10011010 (Hasil)

Dan sebaliknya, jika dilakukan operasi XOR lagi:

10100100	10011010	10011000	10011010
11110011	11110011	11110011	11110011
			(XOR)
01010111	01101001	01101011	01101001 (Hasil)

2.4 Tipe Dan Model Algoritma Kriptografi

2.4.1 Bit *String*

Kriptografi klasik menggunakan sistem substansi dan permutasi karakter dari *plaintext*. Pada kriptografi modern, karakter yang ada dikonversi ke dalam suatu urutan digit biner (bit), yaitu 1 dan 0 yang umum digunakan untuk skema *encoding American Standart Code For Information Interchange* (ASCII). Urutan bit akan mewakili *plaintext* yang kemudian dienkripsi untuk kemudian mendapatkan *ciphertext* dalam bentuk urutan bit.

Algoritma enkripsi bisa menggunakan salah satu dari dua metode, yang pertama "natural", pembagian antara *stream chiper*, dimana urutan bit untuk enkripsi menggunakan metode *bit by bit*. Metode kedua adalah *block cipher*, dimana urutan pembagian dalam bentuk ukuran blok yang diinginkan. ASCII memerlukan 8 bit untuk

mendapatkan satu karakter dan blok kode pada umumnya membutuhkan 64-bit untuk satu blok. Sebagai contoh sekuen 12-*bit* : 100111010110. Jika dipecah menjadi 4 blok didapat 100 111 010 110. Bagaimanapun *bit string* dengan panjang 3 menghadirkan bilangan bulat dari 0 sampai 7 dengan urutan menjadi 4 7 2 6. 000=0, 001=1, 010=2, 011=3, 100=4, 101=5,110=7, 111=7.

Jika diambil urutan yang sama dan dipecah dalam blok berukuran 4-bit maka akan didapat 1001 1101 0110. Bit *string* dengan panjang 4 menghasilkan bilangan *integer* dari 0 sampai 15 dan didapat urutan 9 13 6. Pada umumnya urutan bilangan biner dengan panjang n bisa mewakili suatu bilangan bulat dari 0 sampai 2^n-1 (Ariyus, 2008).

2.4.2 Stream Chiper

Stream chiper mengenkripsi *plaintext* menjadi *ciphertext* bit per-bit (1 bit setiap kali transformasi). Pertama kali diperkenalkan oleh Vernam melalui algoritma yang dikenal dengan nama kode Vernam.

Kode Vernam diadopsi dari *one-time-pad cipher,* yang didalam hal ini karakter diganti oleh bit (0 dan 1). *Ciphertext* diperoleh dari hasil *modulo* (sisa hasil bagi) 2 dari penjumlahan satu bit *plaintext* dengan satu bit *key* seperti pada persamaan 2.1:

$$ci = (pi+ki) \bmod 2 \qquad (2.1)$$

yang dalam hal ini :
pi = bit *plaintext*
ki = bit *key*
ci = bit *ciphertext*

Plaintext diperoleh dari hasil *modulo* (sisa hasil bagi) 2 pengurangan satu bit *ciphertext* dengan satu bit *key* seperti pada persamaan 2.2:

$$pi = (ci-ki) \bmod 2 \qquad (2.2)$$

Oleh karena operasi penjumlahan *modulo* 2 identik dengan operasi bit dengan operator XOR maka persamaan untuk memperoleh *ciphertext* dapat dituliskan seperti pada persamaan 2.3:

$$ci=pi \text{ XOR } ki \qquad (2.3)$$

dan untuk memperoleh *plaintext* dapat dituliskan seperti pada persamaan 2.4:

$$pi=ci \text{ XOR } ki \quad (2.4)$$

Pada *stream chiper*, bit hanya mempunyai dua buah nilai sehingga proses enkripsi hanya menyebabkan dua keadaan pada bit tersebut. Berubah atau tidak berubah. Dua keadaan itu ditentukan oleh kunci enkripsi yang disebut *keystream.*

Keystream dibangkitkan dari sebuah pembangkit yang dinamakan *keystream generator*. *Keystream* di-XOR-kan dengan aliran bit-bit *plaintext* untuk menghasilkan bit-bit *ciphertext*.

Keamanan sistem *stream chiper* bergantung seluruhnya pada *keystream generator*. Jika *keystream generator* mengeluarkan aliran bit kunci yang seluruhnya nol maka *ciphertext* sama dengan *plaintext* dan proses enkripsi menjadi tidak ada artinya. Jika pembangkit mengeluarkan *keystream* dengan pola 16-bit yang berulang maka algoritma enkripsinya menjadi sama dengan enkripsi dengan XOR sederhana yang memiliki tingkat keamanan yang tidak berarti (Ariyus,2008).

2.4.3 Block Chiper

Block Cipher mengenkripsi *plaintext* dengan ukuran blok n bit (pada umumnya 64-bit). Karena sebagian besar pesan terdiri dari bit-bit yang sangat banyak, maka pendekatan paling sederhana adalah dengan membagi pesan-pesan tersebut ke dalam n-bit blok dan mengenkripsinya secara terpisah (Menezes, 1996).

Misalkan blok *plaintext* (P) yang berukuran *m* bit dinyatakan sebagai vektor seperti pada persamaan 2.5:

$$P=(p_1,p_2,\ldots\ldots,p_m) \quad (2.5)$$

Yang dalam hal ini p_i adalah 0 atau 1 untuk i=1,2,…,*m,* dan blok *ciphertext* (C) adalah seperti pada persamaan 2.6:

$$C=(c_1,c_2,\ldots\ldots,c_m) \quad (2.6)$$

Yang dalam hal ini c_i adalah 0 atau 1 untuk i=1,2,…,*m.*

Bila *plaintext* dibagi menjadi *n* buah blok barisan blok *plaintext* dinyatakan seperti pada persamaan 2.7:

$$(P_1,P_2,\ldots\ldots,P_m) \quad (2.7)$$

Untuk setiap blok *plaintext* P_i, bit-bit penyusunnya dapat dinyatakan sebagai vektor seperti pada persamaan 2.8:

$$P_i = (p_{i1}, p_{i2}, \ldots\ldots, pi_m). \quad (2.8)$$

Enkripsi dan dekripsi dengan kunci K dinyatakan berturut-turut dengan persamaan seperti pada persamaan 2.9:

$$E_k(P) = C \quad (2.9)$$

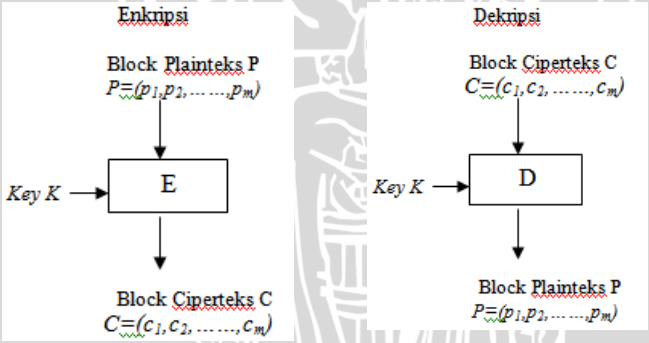
Untuk dekripsinya seperti pada persamaan 2.10:

$$D_k(C) = P \quad (2.10)$$

Fungsi E haruslah fungsi yang berkorespondensi satu ke satu seperti pada persamaan 2.11:

$$E^{-1} = D \quad (2.11)$$

Untuk proses enkripsi dan dekripsi seperti pada gambar 2.1:



Gambar 2.1 Proses Enkripsi dan Dekripsi *block cipher* (Ariyus, 2008).

2.4.4 Jaringan *Feistel*

Kebanyakan algoritma *block cipher* adalah jaringan *feistel*. Ide ini muncul pada awal 1970-an. Jaringan *feistel* ini akan mengambil

sebuah blok dengan panjang n dan membaginya menjadi dua bagian yaitu L dan R. Dan melakukan iterasi, dimana *output* dari putaran ke-i ditentukan dari keluaran putaran sebelumnya seperti pada persamaan 2.12 dan persamaan 2.13:

$$L_i = R_{i+1} \quad (2.12)$$

$$R_i = L_{i-1} \square f(R_{i-1}, K_i) \quad (2.13)$$

Ki adalah subkunci yang digunakan pada putaran ke-i dan *f* adalah fungsi transformasi. Jaringan *feistel* banyak dipakai pada algoritma kriptografi DES, LOKI, GOST, FEAL, *Lucifer*, *CAST-128*, Khufu, Khafre, dan lain-lain, karena model ini bersifat *reversible* untuk proses enkripsi dan dekripsi. Sifat *reversible* ini membuatnya tidak perlu membuat algoritma baru untuk mendekripsi *ciphertext* menjadi *plaintext*. Karena operator XOR mengkombinasikan setengah bagian kiri dengan hasil dari fungsi transformasi *f*, maka persamaan 2.14 berikut pasti benar:

$$L_{i-1} \square f(R_{i-1}, K_i) \square f(R_{i-1}, K_i) = L_{i-1} \quad (2.14)$$

(Schneier,1996).

2.4.5 Padding

Sebagian besar pesan tidak terbagi dalam 64-bit (atau ukuran bit berapa pun). Pada umumnya ada blok yang pendek (tidak sampai 64-bit) di bagian akhir. *Padding* adalah salah satu solusi untuk mengatasi hal tersebut.

Salah satu metode *padding* adalah dengan mengisi blok terakhir yang tidak lengkap dengan string nol agar ukuran blok menjadi lengkap.

Sebagai contoh, ukuran blok untuk enkripsi diasumsikan adalah 64-bit dan blok yang terakhir terdiri dari 3 *bytes* (24 bit). Lima *bytes* lagi diperlukan untuk membuat blok yang terakhir menjadi 64-bit. Caranya adalah dengan menambahkan 4 *bytes* nol dan satu *byte* terakhir dengan 5. Setelah proses dekripsi selesai, maka lima *bytes* terakhir tadi akan terlihat sebagai bit-bit hasil *padding* dan bisa dihapus (Schneier, 1996).

2.5 Algoritma *CAST*

Algoritma CAST merupakan algoritma yang serupa dengan algoritma Blowfish. Algoritma ini didesain oleh Stafford Adams dan Carlisle Adams, dan nama "*CAST*" merepresentasikan huruf pertama pada nama mereka.

Terdapat dua macam algoritma *CAST*, yaitu *CAST-128* dan *CAST-256*. Algoritma *CAST-128* menggunakan 12 atau 16-*round* jaringan *Feistel* dengan 64-bit ukuran blok dan ukuran kuncinya sekitar 40 hingga128 bit. 16 *round* penuh digunakan ketika kunci yang digunakan lebih panjang dari 80 bit. Komponen-komponennya termasuk didalamnya sebuah 8x32-bit *Sboxes* besar yang berdasarkan *bent-functions*, rotasi kunci independen, *modular addition*, dan subtraksi, serta operasi XOR.

Sementara itu algoritma *CAST-256* adalah sebuah *cipher* simetri yang didesain berdasarkan prosedur mendesain *CAST*. Algoritma ini merupakan ektensi dari *CAST-128* dan telah didaftarkan sebagai salah satu kandidat untuk NIST *Advanced Encryption Standard*(AES). Dalam mendesain algoritma CAST-256 ini, Howard Heys dan Michael Wiener juga turut berkontribusi.

CAST-256 menggunakan komponen-komponen yang sama dengan *CAST-128*, termasuk *S-boxes* (yang diadaptasi dari *block* berukuran 128 bit). Panjang kunci yang dapat diterima adalah 128, 160, 224, atau 256 bit. *CAST-256* menggunakan 48 putaran (*round*) yang sering disebut sebagai 12 "*quad-rounds*".(As'ad. 2010).

2.5.1 Gambaran sederhana Algoritma *CAST-128*

CAST-128 termasuk kelas algoritma enkripsi yang menggunakan jaringan feistel. Secara umum, algoritma ini juga mirip dengan algoritma *Data Encryption Standard*(DES). Berikut ini langkah-langkah secara garis besar dari fungsi enkripsi dalam algoritma *CAST-128*:

Masukan:

1. *Plaintext* p1..p64.
   Blok *plaintext* sepanjang 64 bit.
2. Kunci K=k1..k128.
   Kunci sepanjang 128 bit.

Keluaran: *Ciphertext* c1..c64.
Blok *ciphertext* sepanjang 64 bit.

Langkah-langkah:

1. Penjadwalan kunci.
   Menentukan enam belas pasang kunci (*key*) dari masukan pengguna.
2. Bagi blok menjadi dua bagian.
   64 bit blok plainteks dibagi menjadi dua bagian yang sama, yaitu bagian kiri dan bagian kanan dengan panjang 32 bit.
3. Enam belas putaran *feistel*.
   Lihat bagian 2.3 Jaringan *Feistel* untuk rincian mengenai jaringan tersebut.
4. Konkatenasi untuk membentuk *ciphertext*.
   Tukarkan bagian kiri dengan bagian kanan blok diputaran terakhir. Setelah itu, kedua bagian digabungkan menjadi satu dan menjadi *ciphertext*.

Langkah-langkah dekripsi identik dengan langkah-langkah algoritma enkripsi yang baru saja dibahas. Hanya saja, urutan jadwal kunci yang digunakan pada keenam belas putaran dibalik. Jadi, kunci terakhir akan digunakan pada putaran pertama, dan seterusnya sampai kunci pertama digunakan pada putaran terakhir.(Ariyus, 2008).

2.5.2Kotak-S(*S-box*)

CAST-128 menggunakan delapan 8 x 32 *S-box*: *S-box* S1, S2, S3, dan S4 adalah *S-box* fungsi putaran, sedangkan empat sisanya: S5, S6, S7, dan S8 adalah *S-box* jadwal kunci. Karena *S-box* adalah kotak 8 x 32, kotak ini akan menerima 8 bit masukan dan mengeluarkan 32 bit keluaran.

2.5.3 Pembangkitan Kunci Internal

Karena *CAST-128* menggunakan enambelas putaran *feistel*, dimana setiap putarannya, *CAST-128* membutuhkan sepasang kunci internal, tentunya dibutuhkan kunci sebanyak 32 buah, yaitu enam belas buah (Km1..Km16) dan enam belas buah (Km1..Km16). Enam belas kunci Km1..Km16 digunakan untuk *masking* disetiap putaran,

sedangkan Kr1..Kr16 digunakan untuk kunci rotasi. Kunci-kunci internal ini dibangkitkan dari kunci eksternal yang diberikan oleh pengguna. Jadi, Dari kunci eksternal yang panjangnya 128 bit, dibentuk enam belas pasang kunci internal, yaitu pasangan 32 bit kunci *masking* dan 5 bit kunci rotasi. Algoritma pembangkitan kunci internal diberikan dengan asumsi sebagain berikut:

Dimisalkan Kunci eksternal sepanjang 128 bit dibagi menjadi 16 bytes subkunci, yaitu: x0x1x2x3x4x5x6x7x8x9xAxBxCxDxExF, Dimana x0 menunjukkan *most significant byte* (MSB) dan xF menunjukkan *least significant byte* (LSB). Selain itu, digunakan z0..zF yang merupakan *temporary byte* untuk penyimpanan *byte* sementara. Digunakan juga Si[] yang merepresentasikan *s-box* ke-i. Sebagai catatan, "^" merepresentasikan operasi XOR. Algoritma pembangkitan kunci internal diberikan sebagai berikut:

```
z0z1z2z3=x0x1x2x3^S5[xD]^S6[xF]^S7[xC]^S8[xE]^S7[x8]
z4z5z6z7=x8x9xAxB^S5[z0]^S6[z2]^S7[z1]^S8[z3]^S8[xA]
z8z9zAzB=xCxDxExF^S5[z7]^S6[z6]^S7[z5]^S8[z4]^S5[x9]
zCzDzEzF=x4x5x6x7^S5[zA]^S6[z9]^S7[zB]^S8[z8]^S6[xB]
K1=S5[z8]^S6[z9]^S7[z7]^S8[z6]^S5[z2]
K2=S5[zA]^S6[zB]^S7[z5]^S8[z4]^S6[z6]
K3=S5[zC]^S6[zD]^S7[z3]^S8[z2]^S7[z9]
K4=S5[zE]^S6[zF]^S7[z1]^S8[z0]^S8[zC]
x0x1x2x3=z8z9zAzB^S5[z5]^S6[z7]^S7[z4]^S8[z6]^S7[z0]
x4x5x6x7=z0z1z2z3^S5[x0]^S6[x2]^S7[x1]^S8[x3]^S8[z2]
x8x9xAxB=z4z5z6z7^S5[x7]^S6[x6]^S7[x5]^S8[x4]^S5[z1]
xCxDxExF=zCzDzEzF^S5[xA]^S6[x9]^S7[xB]^S8[x8]^S6[z3]
K5=S5[x3]^S6[x2]^S7[xC]^S8[xD]^ S5[x8]
K6=S5[x1]^S6[x0]^S7[xE]^S8[xF]^ S6[xD]
K7=S5[x7]^S6[x6]^S7[x8]^S8[x9]^ S7[x3]
K8=S5[x5]^S6[x4]^S7[xA]^S8[xB]^ S8[x7]
z0z1z2z3=x0x1x2x3^S5[xD]^S6[xF]^S7[xC]^S8[xE]^S7[x8]
z4z5z6z7=x8x9xAxB^S5[z0]^S6[z2]^S7[z1]^S8[z3]^S8[xA]
z8z9zAzB=xCxDxExF^S5[z7]^S6[z6]^S7[z5]^S8[z4]^S5[x9]
zCzDzEzF=x4x5x6x7^S5[zA]^S6[z9]^S7[zB]^S8[z8]^S6[xB]
K9=S5[z3]^S6[z2]^S7[zC]^S8[zD]^S5[z9]
K10=S5[z1]^S6[z0]^S7[zE]^S8[zF]^S6[zC]
K11=S5[z7]^S6[z6]^S7[z8]^S8[z9]^S7[z2]
K12=S5[z5]^S6[z4]^S7[zA]^S8[zB]^S8[z6]
x0x1x2x3=z8z9zAzB^S5[z5]^S6[z7]^S7[z4]^S8[z6]^S7[z0]
x4x5x6x7=z0z1z2z3^S5[x0]^S6[x2]^S7[x1]^S8[x3]^S8[z2]
x8x9xAxB=z4z5z6z7^S5[x7]^S6[x6]^S7[x5]^S8[x4]^S5[z1]
xCxDxExF=zCzDzEzF^S5[xA]^S6[x9]^S7[xB]^S8[x8]^S6[z3]
K13=S5[x8]^S6[x9]^S7[x7]^S8[x6]^S5[x3]
K14=S5[xA]^S6[xB]^S7[x5]^S8[x4]^S6[x7]
K15=S5[xC]^S6[xD]^S7[x3]^S8[x2]^S7[x8]
```

```
K16=S5[xE]^S6[xF]^S7[x1]^S8[x0]^S8[xD]
```

Setengah bagian siasanya identik dengan setengah bagian algoritma si atas dengan meneruskan x0..xF yang terakhir untuk membangkitkan K17..32.

```
z0z1z2z3=x0x1x2x3^S5[xD]^S6[xF]^S7[xC]^S8[xE]^S7[x8]
z4z5z6z7=x8x9xAxB^S5[z0]^S6[z2]^S7[z1]^S8[z3]^S8[xA]
z8z9zAzB=xCxDxExF^S5[z7]^S6[z6]^S7[z5]^S8[z4]^S5[x9]
zCzDzEzF=x4x5x6x7^S5[zA]^S6[z9]^S7[zB]^S8[z8]^S6[xB]
K17=S5[z8]^S6[z9]^S7[z7]^S8[z6]^S5[z2]
K18=S5[zA]^S6[zB]^S7[z5]^S8[z4]^S6[z6]
K19=S5[zC]^S6[zD]^S7[z3]^S8[z2]^S7[z9]
K20=S5[zE]^S6[zF]^S7[z1]^S8[z0]^S8[zC]
x0x1x2x3=z8z9zAzB^S5[z5]^S6[z7]^S7[z4]^S8[z6]^S7[z0]
x4x5x6x7=z0z1z2z3^S5[x0]^S6[x2]^S7[x1]^S8[x3]^S8[z2]
x8x9xAxB=z4z5z6z7^S5[x7]^S6[x6]^S7[x5]^S8[x4]^S5[z1]
xCxDxExF=zCzDzEzF^S5[xA]^S6[x9]^S7[xB]^S8[x8]^S6[z3]
K21=S5[x3]^S6[x2]^S7[xC]^S8[xD]^S5[x8]
K22=S5[x1]^S6[x0]^S7[xE]^S8[xF]^S6[xD]
K23=S5[x7]^S6[x6]^S7[x8]^S8[x9]^S7[x3]
K24=S5[x5]^S6[x4]^S7[xA]^S8[xB]^S8[x7]
z0z1z2z3=x0x1x2x3^S5[xD]^S6[xF]^S7[xC]^S8[xE]^S7[x8]
z4z5z6z7=x8x9xAxB^S5[z0]^S6[z2]^S7[z1]^S8[z3]^S8[xA]
z8z9zAzB=xCxDxExF^S5[z7]^S6[z6]^S7[z5]^S8[z4]^S5[x9]
zCzDzEzF=x4x5x6x7^S5[zA]^S6[z9]^S7[zB]^S8[z8]^S6[xB]
K25=S5[z3]^S6[z2]^S7[zC]^S8[zD]^S5[z9]
K26=S5[z1]^S6[z0]^S7[zE]^S8[zF]^S6[zC]
K27=S5[z7]^S6[z6]^S7[z8]^S8[z9]^S7[z2]
K28=S5[z5]^S6[z4]^S7[zA]^S8[zB]^S8[z6]
x0x1x2x3=z8z9zAzB^S5[z5]^S6[z7]^S7[z4]^S8[z6]^S7[z0]
x4x5x6x7=z0z1z2z3^S5[x0]^S6[x2]^S7[x1]^S8[x3]^S8[z2]
x8x9xAxB=z4z5z6z7^S5[x7]^S6[x6]^S7[x5]^S8[x4]^S5[z1]
xCxDxExF=zCzDzEzF^S5[xA]^S6[x9]^S7[xB]^S8[x8]^S6[z3]
K29=S5[x8]^S6[x9]^S7[x7]^S8[x6]^S5[x3]
K30=S5[xA]^S6[xB]^S7[x5]^S8[x4]^S6[x7]
K31=S5[xC]^S6[xD]^S7[x3]^S8[x2]^S7[x8]
K32=S5[xE]^S6[xF]^S7[x1]^S8[x0]^S 8[xD]
```

Dapat dilihat dari algoritma di atas bahwa. Pembangkitan kunci internal menggunakan operator XOR dan empat buah *S-box* yaitu S5, S6, S7, dan S8. Dengan algoritma ini didapat K1..K32, dimana penjadwalannya diberikan dibagian selanjutnya.

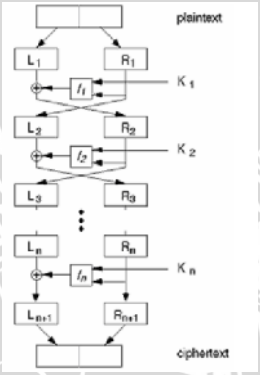
2.5.4 Penjadwalan Kunci

Dari K1..K32 yang dibangkitkan sebelumnya, enam belas kunci pertama akan digunakan untuk kunci *masking* (satu kunci per

putaran) dan enam belas sisanya digunakan untuk kunci rotasi (satu kunci per putaran pula). Untuk kunci rotasi, hanya 5 bit dari *least significant byte* yang digunakan. Algoritma penjadwalan kunci diberikan sebagai berikut:

```
for (i=1; i<=16; i++)
{
  Kmi = Ki;
  Kri = K16+i;
}
```

Digambarkan seperti gambar 2.2:

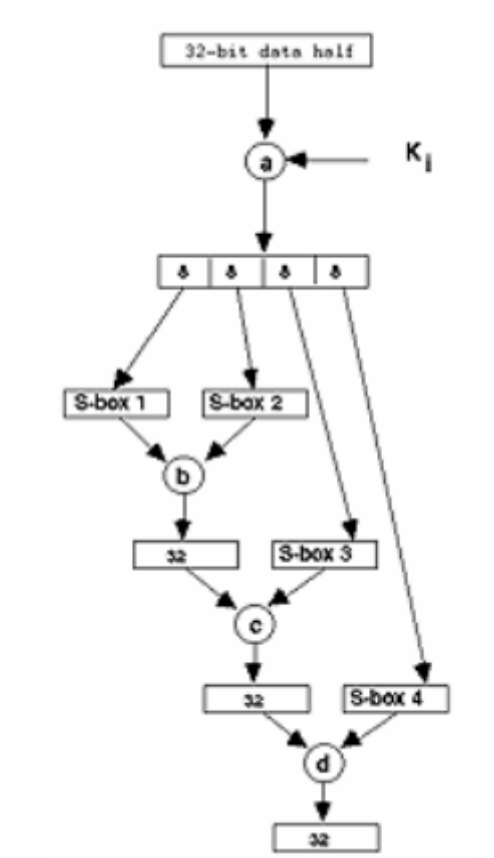


Gambar 2.2 Putaran *Feistel* n putaran.

Sesuai gambar 10 di halaman sebelumnya, jika n=16, yang berarti ada enam belas putaran feistel, akan diperlukan enam belas kunci, yaitu K1..K16. Dalam *CAST-128* keenambelas kunci tersebut diisi dengan keenambelas pasang kunci yang telah dibangkitkan. Contohnya: K1 adalah sepasang kunci Km1 dan Kr1.

2.5.5 Fungsi Enkripsi

Dari enam belas putaran feistel yang digunakan, CAST-128 menggunakan tiga jenis putaran yang berbeda. Tiga jenis putaran tersebut dibedakan menurut tiga tipe fungsi enkripsi yang berbeda. Skema fungsi enkripsi seperti pada gambar 2.3:



Gambar 2.3Fungsi *CAST*

Gambar 2.3 menunjukkan bahwa fungsi enkripsi *CAST-128* menerima input 32 bit data *half*, yang didapatkan dari 64 bit blok yang telah dibagi dua pada saat masuk jaringan *feistel*. Lalu operasi a dilakukan pada 32 bit data masukan ini. Setelah operasi a dilakukan, hasilnya dibagi menjadi empat bagian dengan panjang yang sama (delapan bit). Delapan bit pertama akan menjadi input dari *S-box* 1 dan delapan bit kedua akan menjadi input dari *S-box* 2. Ingat bahwa karena *S-box* yang dipakai adalah kotak 8 x 32, dari 8 bit masukan akan dihasilkan 32 bit keluaran. Selanjutnya, Hasil dari kedua *S-box* tadi akan digabung dengan menggunakan operasi b. Setelah itu, hasil dari operasi b akan digabung dengan hasil dari *S-box* 3 dengan masukan delapan bit ketiga. Operasi yang digunakan kali ini adalah

operasi c. Delapan bit terkahir akan menjadi masukan *S-box* keempat dan hasilnya akan digabung dengan hasil operasi c menggunakan operasi d. Hasil yang diperoleh dari seluruh fungsi ini adalah sepanjang 32 bit, sesuai dengan 32 bit masukan. Tiga puluh dua bit hasil ini akan kembali masuk ke putaran *feistel*.

Tentunya, dari penjelasan di atas, yang menjadi pertanyaan adalah apa operasi a, b, c , dan d. Keempat operasi inilah yang berbeda di tiga jenis fungsi *CAST*.

1. Tipe 1.
   - Operasi a adalah penjumlahan bit modulo 2^{32} dan penggeseran bit ke kiri (*circular left-shift operation*). Bit-bit masukan akan ditambahkan dengan Kmi (sesuai putaran ke-i) dan akan digeser ke kiri sebanyak Kri (sesuai putaran ke-i pula).
   - Operasi b adalah XOR.
   - Operasi c adalah pengurangan bit modulo 2^{32}.
   - Operasi d adalah penjumlahan bit modulo 2^{32}.

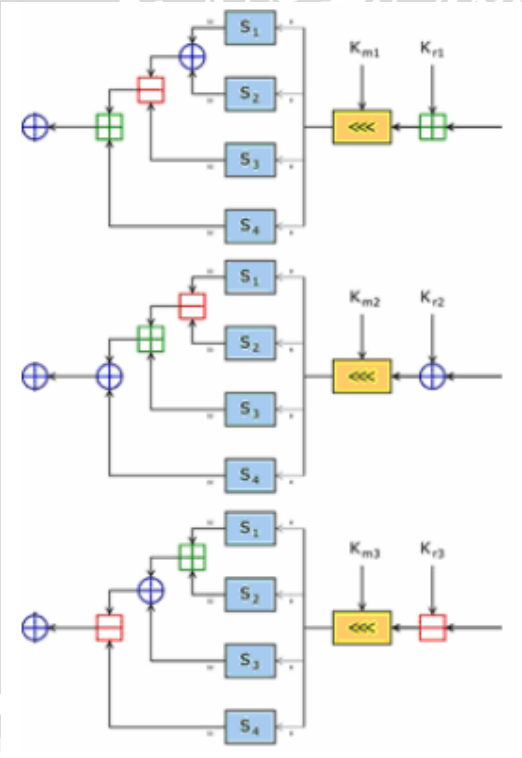
2. Tipe 2.
   - Operasi a adalah XOR dan penggeseran bit ke kiri (*circular left-shift operation*). Bit-bit masukan akan di-XOR-kan dengan Kmi (sesuai putaran ke-i) dan akan digeser ke kiri sebanyak Kri (sesuai putaran ke-i pula).
   - Operasi b adalah pengurangan bit modulo 2^{32}.
   - Operasi c adalah penjumlahan bit modulo 2^{32}.
   - Operasi d adalah XOR.

3. Tipe 3.
   - Operasi a adalah pengurangan bit modulo 2^{32} dan penggeseran bit ke kiri (*circular left-shift operation*). Kmi (sesuai putaran ke-i) akan dikurangkan dengan bit-bit masukan dan akan digeser ke kiri sebanyak Kri (sesuai putaran ke-i pula).
   - Operasi b adalah penjumlahan bit modulo 2^{32}.
   - Operasi c adalah XOR.
   - Operasi d adalah pengurangan bit modulo 2^{32}.

Secara matematis ketiga tipe tersebut dinyatakan dengan persamaan 2.15.

Type 1: I = ((Kmi + D) <<< Kri)
f = ((S1[Ia] ^ S2[Ib]) - S3[Ic]) + S4[Id]
Type 2: I = ((Kmi ^ D) <<< Kri)
f = ((S1[Ia] - S2[Ib]) + S3[Ic]) ^ S4[Id]
Type 3: I = ((Kmi - D) <<< Kri)
f = ((S1[Ia] + S2[Ib]) ^ S3[Ic]) - S4[Id] (2.15)

Dimana D adalah 32 bit data input, I adalah hasil operasi a terhadap D. I dibagi menjadi empat bagian sepanjang 8 bit la, lb, lc, dan ld terurut mulai *dari most significant byte* (MSB) sampai *least significant byte*(LSB). f adalah hasil fungsi enkripsi. Sebagai catatan, "+" dan "-"adalah penjumlahan dan pengurangan modulo 2^{32}, "^"adalah XOR, dan "<<<" adalah penggeseran bit ke kiri (*circular left-shift operation*). Untuk lebih jelasnya, diberikan gambar 12 dimana empat operasi a, b, c, dan d sudah diganti sesuai tipenya seperti pada gambar 2.4.



Gambar 2.4 Tiga Fungsi Enkripsi *CAST-128*

Jadwal pemakaian ketiga tipe fungsi tersebutdalam jaringan *feistel* diberikan di bawah ini:

- Putaran 1, 4, 7, 10, 13, and 16 menggunakan fungsi tipe 1.
- Putaran 2, 5, 8, 11, and 14 menggunakan fungsi tipe 2.
- Putaran 3, 6, 9, 12, and 15 menggunakan fungsi tipe 3.

2.5.6 Fungsi Dekripsi

Algoritma *CAST-128* memiliki keunikan dalam hal proses dekripsi, yaitu proses dekripsi identik dengan langkah-langkah proses enkripsi, hanya saja urutan jadwal kunci yang digunakan pada keenam belas putaran dibalik. Jadi kunci terakhir akan digunakan pada putaran pertama dan seterusnya sampai kunci pertama digunakan pada putaran terakhir. (Ariyus,2008).

2.5.7 Panjang Kunci dan Pengaruhnya

Algoritma *CAST-128* didesain untuk mampu menerima berbagai macam panjang kunci yangn berbeda mulai dari 40 bit sampai dengan 128 bit, dimana perbedaan antara nilai tersebut harus dalam kelipatan delapan. Jadi panjang kunci yang valid adalah: 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 bit. Berdasarkan panjang kunci tersebut, ada perbedaan cara enkripsi pada *CAST-128*, yaitu:

1. Jika panjang kunci 40 bit sampai dengan 80 bit, algoritma yang digunakan sama persis, hanya saja digunakan hanya dua belas putaran *feistel* dari enam belas yang seharusnya.

2. Untuk panjang kunci lebih dari 80 bit, digunakan penuh algoritma yang telah diutarakan dengan enam belas putaran.

3. Untuk Kunci kurang dari 128 bit, kunci akan di*padding* dengan bit 0 di bagian kanan atau di bagian *least significant byte* sampai dicapai 128 bit kunci.

2.6 *Avalanche Effect*

Salah satu karakteristik untuk menentukan baik atau tidaknya suatu algoritma kriptografi adalah dengan melihat *avalanche effect*-nya. yaitu perubahan minimum pada masukan mampu menghasilkan perubahan drastis pada keluaran, analog dengan terjadinya *avalanche* (longsor salju) dimana satu gerakan kecil di puncak dapat menimbulkan longsor yang semakin membesar dan merusak di lembah (Ariesanda, 2006). Efek ini sejalan dengan konsep *diffusion* dalam Teori Informasi oleh Shannon yang menjadi landasan umum bagi kriptografi (Ariesanda, 2006).

Prinsip *diffusion* ini menyebarkan pengaruh satu bit *plaintext* atau kunci ke sebanyak mungkin *ciphertext*. Sebagai contoh, pengubahan kecil pada *plaintext* sebanyak satu atau dua bit menghasilkan perubahan pada *ciphertext* yang tidak dapat diprediksi (Munir, 2004).

Prinisp *diffusion* juga menyembunyikan hubungan statistik antara *plaintext, ciphertext,* dan kunci dan membuat kriptanalisis menjadi sulit (Munir, 2004).

Suatu *avalanche effect* dikatakan baik jika perubahan bit yang dihasilkan berkisar antara 45-60% (sekitar separuhnya, 50 % adalah hasil yang sangat baik). Hal ini dikarenakan perubahan tersebut berarti membuat perbedaan yang cukup sulit untuk kriptanalis melakukan serangan (Rudianto,2004).

Persamaan *Avalance Effect* dapat dilihat seperti pada persamaan 2.16.

$$\textit{Avalanche Effect}=\left(\frac{\text{jumlah bit berbeda}}{\text{panjang bit total}}\right).100\%\text{(Andini,2011)(2.16)}$$

Sebagai contoh perhitungan *avalanche effect* adalah sebagai berikut :

Perubahan *plaintext* 1 bit ditunjukkan sebagai berikut :

Tabel *Avallanche Effect* perubahan *plaintext* 1 bit
Plaintext 1 : 00000000000000000000000000000000 (*hex*)
Plaintext 2 : 80000000000000000000000000000000 (*hex*)

Dengan kunci yang digunakan adalah :
Key : 00000000000000000000000000000000 (*hex*)

Ciphertext yang dihasilkan adalah :
*Ciphertext*1:DBAD348CF30BBF8E64B5E5D3065D6898 (*hex*)
*Ciphertext*2:D2734057410AE10710C4922DCC9B34FA (*hex*)

Perbedaan bit pada *ciphertext* 1 dengan *ciphertext* 2 adalah sebanyak 67 bit lebih dari separuh besar blok (128 bit) sekitar 52%.

Sedangkan untuk perubahan kunci 1 bit kunci ditunjukkan sebagai berikut :

Tabel *Avalanche Effect* perubahan kunci 1 bit
Plaintext : 00000000000000000000000000000000 (*hex*)

Dengan kunci yang digunakan adalah :
*Key*1 : 00000000000000000000000000000000 (*hex*)
*Key*2 : 80000000000000000000000000000000 (*hex*)

Chipertext yang dihasilkan adalah :
*Chipertext*1:DBAD348CF30BBF8E64B5E5D3065D6898(*hex*)
*Chipertext*2:48F00DFF8C90822417D12ECAD682B014 (*hex*)

Perbedaan bit pada *chipertext* 1 dengan *chipertext* 2 adalah sebanyak 72 bit atau sekitar 56% dari besar blok (128 bit). (Budiyono, 2004) .


UNIVERSITAS BRAWIJAYA