Studi *Block Cipher* Serpent dan Rijndael

Aulia Rahma Amin
13503009
Email : aulia@students.itb.ac.id
Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Abstrak

Cipher blok Serpent dan Rijndael, keduanya adalah algoritma kriptografi berbasis blok yang menggunakan panjang blok 128 bit. Keduanya adalah finalis pada kompetisi AES (Advance Encryption Standard). *Cipher* Rijndael memenangkan kompetisi tersebut dengan meraih 86 suara sedangkan *Cipher* Serpent menduduki peringkat kedua dengan 59 suara [1].

Makalah ini akan membahas cara kerja kedua algoritma tersebut. Pada makalah ini juga akan dipaparkan kelebihan dan kekurangan masing-masing algoritma.

Kata Kunci : *Rijndael, Serpent, Block Cipher*

1. Latar Belakang

Pada tanggal 2 Januari 1997, *National Institute of Standard Technology* (NIST), sebuah badan penelitian milik pemerintah Amerika Serikat, mengumumkan dimulainya usaha pembuatan algoritma ***Advanced Encryption Standard*** **(AES)** sebagai pengganti DES. Tujuan keseluruhan dari usaha ini adalah untuk membangun suatu *Federal Information Processing Standards* (FIPS) yang menspesifikasikan sebuah algoritma (atau algoritma-algoritma) yang dapat melindungi informasi penting milik pemerintah.

Pada 20 Agustus 1998, NIST mengumumkan lima belas algoritma kandidat. Algoritma-algoritma ini dikirimkan oleh komunitas kriptografi di seluruh dunia. Setelah menyelenggarakan konferensi dan memeriksa analisis/komentar dari komunitas kriptografi, pada 15 April 1999 NIST mengumumkan lima algoritma finalis: **Mars, RC6, Rijndael, Serpent,** dan **Twofish**.

Setelah melalui proses pemeriksaan oleh NIST dan masukan serta komentar dari komunitas kriptografi, pada tanggal 2 Oktober 2000 Rijndael terpilih sebagai proposal AES (FIPS-197). Algoritma Rijndael diciptakan dan dikirimkan oleh dua *cryptographers* dari Belgia: Dr. Joan Daemen dari Proton World International dan Dr. Vincent Rijmen, seorang peneliti *post-doctoral* pada Electrical Engineering Department (ESAT) Katholieke Universiteit Leuven.

Kedua algoritma Serpent dan Rijndael terbukti cukup kuat dan menarik untuk dipelajari.

2. *Block Cipher* Serpent

2.1. Pendahuluan

Serpent adalah sebuah *Block Cipher* dengan panjang blok 128 bit. *Cipher* ini didesain oleh Ross Anderson, Eli Biham dan Lars Knudsen. *Cipher* ini adalah kandidat Advance Encryption Standard (AES). *Cipher* Serpent menduduki peringkat kedua pada kompetisi AES dengan 59 suara. Serpent dirancang untuk menyediakan *Cipher* yang sangat aman dan nyaris tidak ada jalan pintas bagi para kriptanalis untuk menyerang *Cipher* ini. Untuk itu, perancang membatasi *Cipher* ini dengan hanya menggunakan mekanisme kriptografi yang telah umum dipahami. Serpent menggunakan putaran yang lebih banyak dibandingkan Rijndael, sehingga Serpent kalah cepat apabila dibandingkan dengan Rijndael. Meski demikian, Serpent terbukti lebih cepat daripada DES dan lebih aman dibanding Triple DES, karena dirancang untuk mendukung implementasi *bitslice* yang sangat efektif. Versi paling cepat dari *Cipher* serpent mampu berjalan pada kecepatan 45 Mbit/sec pada komputer 200 MHz Pentium. Ini lebih cepat dibanding DES yang hanya mampu menembus angka 15 Mbit/sec pada komputer yang sama.[2]

Serpent menggunakan S-box dari DES yang telah banyak dipelajari selama bertahun-tahun dengan properti-properti yang dapat dipahami dengan baik, dengan struktu baru

yang telah dioptimasi untuk implementasi yang lebih efisien pada *Processor* modern. Rancangan serpent tahan terhadap semua jenis serangan yang telah diketahui, termasuk serangan-serangan yang menggunakan teknik diferensial dan linier.

Ada beberapa varian *Cipher* serpent. Varian utama adalah *Cipher* 32 putaran yang diyakini seaman *Cipher* tiga kunci triple DES. *Cipher* serpent 32 putaran ini sedikit lebih lambat daripada DES. *Cipher* ini beroperasi pada empat *word* berukuran masing-masing 32 bit sehingga ukuran blok menjadi 128 bit.

Varian lainnya dibuat dengan meningkatkan ukuran blok. Ukuran blok dapat ditingkatkan menjadi dua kali lipat menjadi 256 bit baik dengan meningkatkan ukuran *word* dari 32 bit menjadi 64 bit, maupun dengan menggunakan fungsi putaran pada jaringan Feistel. Dua varian *Cipher* serpent ini dapat dikombinasikan untuk menghasilkan *Cipher* dengan panjang blok 512 bit.

Semua nilai yang digunakan pada *Cipher* direpresentasikan dalam litle-endian, termasuk urutan bit (0-31 dalam *word* berukuran 32 bit, atau 0-127 dalam blok 128 bit), dan urutan *word* dalam blok. Bit 0 adalah bit paling tidak berarti (*Least Significant Bit*), dan *word* 0 adalah *word* paling tidak berarti *(Least Significant Word*). Notasi penulisan sangat penting karena ada dia representasi serpent yang ekuivalen yaitu representasi standard dan representasi *bitslice*.

2.2. Deskripsi *Block Cipher* Serpent

Varian utama dari *Cipher* serpent mengenkripsi 128 bit plainteks P menjadi 128 bit *Cipher*teks C dalam r putaran menggunakan r+1 kunci internal (K_0, K_1, ..., K_r) yang masing-masing panjangnya 128 bit. Secara default, jumlah putaran r adalah sebanyak 32.

*Cipher*nya sendiri adalah sebuah jaringan SP yang terdiri atas:

a) Sebuah initial permutation IP.
b) 32 putaran, dimana tiap putaran mengandung operasi *key*-mixing, operasi terhadap S-box, dan (untuk semua putaran kecuali putaran terakhir) sebuah transformasi linier. Pada putaran terakhir, transformasi linier ini diganti dengan operasi *key*-mixing tambahan.
c) Final permutation FP.

Initial permutation dan final permutation tidak memiliki nilai kriptografik yang signifikan, melainkan hanya digunakan untuk mengoptimasi dan menyederhanakan implementasi *Cipher*, dan untuk meningkatkan efisiensi komputasional.

Notasi yang digunakan adalah sebagai berikut. Initial permutation IP dioperasikan terhadap plainteks P menghasilkan B_0 yang merupakan input putaran pertama. Putaran diberi nomor 0 sampai 31, dimana putana pertama adalah putaran 0, dan putaran terakhir adalah putaran 31. Output putaran pertama (putaran 0) adalah B_1, output putaran kedua (putaran 1) adalah B_2, output putaran i adalah B_{i+1} dan seterusnya, sampai output putaran terakhir (dimana transformasi linier digantikan dengan operasi *key*-mixing tambahan) adalah B_{32}. Final permutation FP dioperasikan terhadap B_{32} untuk mendapatkan *Cipher*teks C.

Setiap fungsi putaran R_i (i Є {0,...,31}) menggunakan hanya satu S-box. Contoh, R_0 menggunakan S_0, dan 32 replikanya digunakan secara paralel. Replika pertama dari S_0 mengambil bit 0, 1, 2, dan 3 dari B_0 XOR K_0 sebagai inputnya dan mengembalikan empat bit pertama dari *intermediate vector* sebagai output. Replika berikutnya dari S_0 menerima input bit ke 4-7 dari B_0 XOR K_0 dan mengembalikan empat bit berikutnya dari intermedeate *vector*, dan seterusnya. Selanjutnya, *intermediate vector* ditransformasikan menggunakan transformasi linier menghasilkan B_1. Sama seperti sebelumnya, R_1 menggunakan 32 replika dari S_1 secara paralel pada output dari B_1 XOR K_1 dan

mentransformasikan outputnya menggunakan transformasi linier menghasilkan B_2.

Pada putaran terakhir R_{31}, S_{31} dioperasikan dengan output dari B_{31} XOR K_{31}, dan tidak dilakukan transformasi linier melainkan outputnya di-XOR-kan dengan K_{32}. Hasilnya (B_{32}) lalu dipermutasikan dengan FP menghasilkan *Cipher*teks.

Dalam 32 putaran tersebut digunakan 32 S-box yang berbeda yang masing-masing memetakan 4 bit input ke 4 bit output. Tiap S-box hanya digunakan dalam satu putaran, dimana dalam putaran tersebut, S-box digunakan 32 kali secara paralel. Tiga puluh dua S-box tersebut dipilih dari 32 garis terpisah pada delapan S-box DES. S_0 (digunakan pada putaran 0) adalah baris pertama dari S1 DES, S_1 (digunakan pada putaran 1) adalah baris keduak dari S1 DES, S_4 (digunakan pada putaran 4) adalah baris pertama dari S2 DES, dan seterusnya.

Sama seperti DES, initial permutation adalah inverse dari final permutation. *Cipher* serpent dapat dideskripsikan secara formal dengan persamaan sebagai berikut :

$$B_0 = IP(P)$$
$$B_{i+1} = R_i(B_i)$$
$$C = IP^{-1}(B_r)$$

Dimana

$$R_i(X) = L(S_i(X \text{ XOR } K_i)) \quad i = 0,...,r-2$$
$$R_i(X) = S_i(X \text{ XOR } K_i) \text{ XOR } K_r \quad i = r-1$$

Dimana S_i adalah aplikasi dari 32 S-box secara paralel, dan L adalah transformasi linier.

Meskipun tiap putaran nampak lebih lemah daripada putaran pada DES, namun kombinasi putarannya menutupi kelemahan tersebut. Kecepatan tiap putaran yang lebih baik dan penambahan jumlah putaran membuat *Cipher* ini hampir secepat DES dan jauh lebih aman.

Proses dekripsi berbeda dengan proses enkripsi. Pada proses dekripsi, harus digunakan inverse dari S-box, demikian pula inverse transformasi linier, dan urutan kunci internal yang dibalik.

2.3. Implementasi yang Efisien

Untuk mengimplementasikan rancangan *Cipher* diatas, digunakan mode *bitslice*. Ide dasarnya adalah cukup digunakan *Processor* 1 bit untuk mengimplementasikan untuk mengimplementasikan algoritma seperti DES dengan menggunakan instruksi lojik untuk mengemulasi tiap-tiap gerbang logika, jadi dapat digunakan *Processor* 32 bit untuk mengkomputasikan 32 blok DES yang berbeda secara paralel.

Langkah ini jauh lebih efisien daripada implementasi konvensional dimana sebuah *Processor* 32 bit kebanyakan idle ketika komputer beroperasi pada mode 6 bit, 4 bit, atau bahkan bit tunggal.

Cipher serpent dirancang sehingga semua operasi dapat dieksekusi menggunakan 32 proses paralel selama enkripsi dan dekripsi satu blok. Bahkan implementasi algoritma menggunakan mode *bitslice* lebih sederhana daripada menggunakan mode konvensional. Tidak diperlukan intial dan final permutation karena kedua permutasi yang dideskripsikan pada implementasi standard hanyalah konversi data ke representasi *bitslice*. Kita akan melihat representasi yang ekuivalen untuk implementasi *bitslice*.

Cipher terdiri atas 32 putaran. Plainteks menjadi data *intermediate* pertama $B_0 = P$, sesudahnya 32 putaran dijalankan, dimana tiap putaran $i \in \{0,...,31\}$ terdiri atas tiga operasi :

a) *Key* Mixing : Pada tiap putaran, kunci internal K_i dengan panjang 128 bit di-XOR-kan dengan *intermediate* data B_i.
b) S-Box : 128 bit kombinasi input dan kunci dibagi menjadi empat *word* yang masing-masing berukuran 32 bit. S-box yang diimplementasikan sebagai

urutan operasi lojik dioperasikan terhadap empat buah *word* diatas, dan menghasilkan empat *word* output. CPU mengeksekusi 32 replika S-box secara simultan menghasilkan $S_i(B_i$ XOR $K_i)$.

c) Transformasi Linier : 32 bit pada tiap-tiap *word* output dioperasikan sebagai berikut :

$X_0,X_1,X_2,X_3 = S_i(B_i \text{ XOR } K_i)$

$X_0 = X_0 <<< 13$

$X_2 = X_2 <<< 3$

$X_1 = X_1 \text{ XOR } X_0 \text{ XOR } X_2$

$X_3 = X_3 \text{ XOR } X_2 \text{ XOR } (X_0 << 3)$

$X_1 = X_1 <<< 1$

$X_3 = X_3 <<< 7$

$X_0 = X_0 \text{ XOR } X_1 \text{ XOR } X_3$

$X_2 = X_2 \text{ XOR } X_3 \text{ XOR } (X_1 << 7)$

$X_0 = X_0 <<< 5$

$X_2 = X_2 <<< 22$

$B_{i+1}= X_0,X_1,X_2,X_3$

Dimana <<< berarti rotasi, dan << melambangkan pergeseran. Pada putaran terakhir, tansformasi linier ini digantikan dengan *key*-mixing tambahan yaitu $B_r = S_{r-1}(B_{r-1} \text{ XOR } K_{r-1}) \text{ XOR } K_r$.

Alasan pertama memilih transformasi linier adalah untuk memaksimalkan efek *avalanche*. S-box DES mempunyai properti bahwa sebuah perubahan bit input akan mengakibatkan dua bit output berubah. *Difference* set dari {0, 1, 3, 5, 7, 13, 22} modulo 32 mempunyai hanya satu anggota yang sama, sehingga perubahan satu bit input akan mengakibatkan banyak perubahan bit output setelah dua atau lebih putaran. Efeknya adalah bahwa tiap bit plainteks, dan tiap bit kunci internal mempengaruhi semua bit data setelah tiga putaran. Alasan kedua adalah lebih sederhana dan dapat digunakan pada prosesor yang menggunakan sistem pipeline.

2.4. *Key* Schedule

Sebagaimana deskripsi pada *Cipher*, kita dapat mendeskripsikan *key* schedule dalam mode standard atau mode *bitslice*. *Cipher* serpent memerlukan 132 buah *word* yang masing-masing berukuran 32 bit. Pertama, kita ekspansi kunci masukan user sepanjang 256 bit menjadi 33 buah sub kunci (K_0, ..., K_{32}) dengan panjang masing-masing 128 bit. Kita tulis K sebagai delapan *word* (w_{-8}, ... , w_{-1}) yang masing-masing berukuran 32 bit lalu ekspansi menjadi *intermediate key* (disebut juga pre*key*) $w_0,...,w_{131}$ dengan cara rekursif sebagai berikut :

$w_i = (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \Phi \oplus i) <<< 11$

dimana Φ adalah bagian fraksional dari golden ratio $(\sqrt{5}+1)/2$ atau 0x9e3779b9 dalam heksadesimal. Polinomial $x^8+x^7+x^5+x^3+1$ adalah primitif dimana bersama dengan penjumlaha index putaran dipilih untuk memastikan distribusi bit kunci merata pada setiap putaran, dan untuk menghilangkan kunci lemah dan keterhubungan antar kunci.

Round key sekarang dihitung dari pre*key* menggunakan S-box pada mode *bitslice*. Input dan output S-box diambil pada jarak 33 *word*s, untuk meminimalkan potensi serangan dengan teknik diferensial pada beberapa putaran akhir. Kita menggunakan S-box untuk mentransformasi pre*key* w_i menjadi *word* k_i yang merupakan *round key* dengan membagi vektor pre*key* menjadi empat bagian dan mentransformasi *word* ke-i pada tiap-tiap bagian menggunakan $S_{(r+3-i) \bmod r}$. Dapat dilihat pada contoh untuk r = 32 sebagai berikut :

$\{k_0, k_{33}, k_{66}, k_{99}\} = S_3(w_0, w_{33}, w_{66}, w_{99})$

$\{k_1, k_{34}, k_{67}, k_{100}\} = S_3(w_1, w_{34}, w_{67}, w_{100})$

...

$\{k_{31}, k_{64}, k_{97}, k_{130}\} = S_3(w_{31}, w_{64}, w_{97}, w_{130})$

$\{k_{32}, k_{65}, k_{98}, k_{131}\} = S_3(w_{32}, w_{65}, w_{98}, w_{131})$

Lalu kita lakukan penomoran ulang terhadap 32 bit nilai k_j sebagai 128 bit kunci internal K_i (untuk i Є {0,...,r}) sebagai berikut :

$K_i = \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}$

Kemudian operasikan IP ke *round key* untuk menempatkan bit kunci ke dalam kolom yang benar.

2.5. Keamanan

2.5.1. Serangan Brute Force

Dengan ukuran blok 128 bit, serangan brute force akan memerlkan 2^{128} kombinasi plainteks yang berbeda agar seorang penyerang dapat mengenkripsi dan mendekripsi pesan dengan kunci yang tidak diketahui. Serangan ini dapat terjadi pada setiap *Block Cipher* 128 bit tanpa peduli rancangan cihernya.

2.5.2. Mode Operasi

Setelah mengenkripsi sekitar 2^{64} blok plainteks dalam mode CBC atau CFB, seseorang dapat menemukan dua blok *Cipher*teks yang sama. Hal ini memungkinkan penyerang untuk melakukan XOR atas dua blok plainteks yang berkorespondensi dengan blok *Cipher*teks tadi. Dengan lebih banyak plainteks yang demikian, hubungan antara plainteks dengan *Cipher*teks akan diketahui dengan kemungkinan yang lebih besar. Serangan ini dapat terjadi pada setiap *Block Cipher* 128 bit tanpa peduli rancangan cihernya.

2.5.3. Serangan *Key-Collision*

Untuk ukuran kunci k, serangan *key*-collision dapat digunakan untuk mengungkap pesan dengan kompleksitas $2^{k/2}$. Maka dengan panjang kunci 128 bit, kompleksitas untuk mengungkap *Cipher*teks hanya 2^{64}, dan dengan panjang kunci 256 bit kompleksitasnya 2^{128}. Serangan ini dapat dilakukan terhadap semua *Block Cipher*, dan tingkat kesulitannya bergantung pada panjang kuncinya, bukan pada rancangan algoritmanya.

2.5.4. Kriptanalisis Diferensial

Fakta penting tentang Serpent adalah bahwa karakteristik apapun harus mimimal memiliki satu S-box yang aktif pada tiap putaran. Rata-rata minimal dua S-box yang aktif yang diperlukan. Perbedaan hanya satu bit input menyebabkan perbedaan pada minimal dua bit output untuk tiap S-box. Sehingga apabila hanya sati bit yang berbeda pada input suatu putaran, maka minimal dua bit berbeda pada output, dan dua bit ini akan mempengaruhi dua S-box pada putaran berikutnya, dimana perbedaan output minimal empat bit pada putaran berikutnya.

Kita mencari kaakteristik terbaik pada *Cipher* ini. Untuk itu, kita mengambil asumsi kasus terburuk dimana semua masukan mempunyai probabilitas ½, kecuali beberapa masukan yang hanya memiliki kemungkinan satu bit input masukan dan satu bit keluaran, yang diasumsikan tidak mungkin (probabilitas nol). Batasan ini dipenuhi oleh semua S-box kecuali satu entri pada S_{30} dimana nilai maksimalnya 10/16. Tabel 1 menunjukkan probabilitas diferensial untuk tiap putaran dari 1-7.

Putaran	Probabilitas
1	2^{-1}
2	2^{-3}
3	2^{-7}
4	2^{-13}
5	2^{-21}
6	2^{-29}
7	$< 2^{-35}$

Tabel 1

Dapat kita lihat bahwa probabilitas untuk 6 putaran adalah 2^{-29}, maka kemungkinan untuk 24 putaran adalah $2^{-4.29} = 2^{-116}$. Pada prakteknya, probabilitas untuk 24 putaran lebih rendah dari batas ini. Apabila penyerang dapat mengimplamantasikan serangan 8R, penyerang membutuhkan 2^{117} plainteks. Apabila penyerang hanya dapat melakukan serangan 4R dengan karakteristik 28 putaran, peluangnya $2^{-4.35} = 2^{-140}$, dan

penyerang memerlukan lebih banyak plainteks dari yang ada.

Perhatikan bahwa apabila transformasi linier hanya menggunakan rotasi saja, maka tiap karakteistik dapat mempunyai 32 varian rotasi dengan probabilitas yang sama. Inilah alasannya kenapa serpent juga menggunakan pergeseran bit untuk menghindari karakteristik rotasi tersebut.

2.5.5. Kriptanalisis Linier

Dalam kriptanalisis linier, mungkin ditemukan hubungan antar bit pada S-box. Peluang keterhubungan ini adalah ½ ± 2/16. Maka, sebuah karakteristik linier 28 putaran yang hanya memiliki satu S-box aktif pada tiap putarannya memiliki peluang $½ \pm 2^{27}(2/16)^{28} = ½ \pm 2^{-57}$, dan serangan berdasarkan keterhubungan tersebut memerlukan 2^{114} plainteks yang diketahui.

Serangan yang lebih umum dapat menggunakan karakteristik linier dengan lebih dari satu S-box pada beberapa putaran. Untuk kasus ini, probabilitasnya ½ ± 6/16. Sebagaimana kriptanalisis diferensial, kita dapat menentukan batasan probabilitas untuk tiap karakteristik. Probabilitasnya dapat dilihat pada tabel 2.

Putaran	Probabilitas
1	$½ \pm 6/16 = ½ \pm 2^{-1.4}$
2	$½ \pm (6/16)^3 = ½ \pm 2^{-2.2}$
3	$½ \pm (6/16)^8 = ½ \pm 2^{-4.3}$
4	$½ \pm (6/16)^{14} = ½ \pm 2^{-6.8}$
5	$½ \pm (6/16)^{20} = ½ \pm 2^{-9.3}$
6	$½ \pm (6/16)^{27} = ½ \pm 2^{-12.2}$
7	$½ \pm (6/16)^{33} = ½ \pm 2^{-13.8}$

Tabel 2

Dapat dilihat bahwa probabilitas untuk 6 putaran adalah $½ \pm 2^{-12.2}$, kita dapat menyimpulkan bahwa kemungkinan untuk 24 putaran adalah $½ \pm 2^{-45.8}$. Probabilitas untuk 28 putaran adalah $½ \pm 2^{-52.2}$, dan serangan yang berdasarkan kemungkinan tersebut memerlukan minimal 2^{104} plainteks yang diketahui. Pada prakteknya, kompleksitas serangan akan lebih tinggi dari batas bawah ini.

2.5.6. Keterhubungan antar kunci

Karena *key* schedule menggunakan rotasi dan S-box, maka tidak akan ditemui hubungan antar kunci. Lebih-lebih, tiap putaran yang berbeda menggunakan S-box yang berbeda, sehingga apbila keterhubungan kunci ditemukan, serangan berbasiskan keterhubungan kunci tidak dapat dijalankan. Serpent tidak memiliki kelemahan yang terjadi akibat proses *key* schedule karena tidak ada kunci lemah, kunci semi-lemah, dan kunci yang sama.

2.6. Performansi

Pada percobaan menggunakan *Processor* Pentium 133MHz/MMX, implementasi 32 putaran *bitslice* yang belum dioptimasi menghasilkan kecepatan yang sedikit lebih lambat daripada DES. Kecepatannya 8.976.157 bit/sec, sedangkan implementasi DES yang telah dioptimasi berjalan pada kecepatan 9.824.864 bit/sec pada mesin yang sama.[3]

2.7. Varian Lain

Sebagaimana disebutkan sebelumnya, ada dua cara untuk menggandakan ukuran blok :

a) Meningkatkan panjang tiap *word* (dalam implementasi *bitslice*) dari 32 ke 64 bit (atau lebih).
b) Menggunakan fungsi putaran sebagaimana fungsi F pada jaringan Feistel.

Jika dua langkah ini dijalankan, ukuran blok akan berlipat empat.

Varian ini memerlukan modifikasi *Cipher*, seperti modifikasi konstanta putaran.

2.8. Kesimpulan

2.8.1. *Cipher* Serpent memenuhi standard AES.
2.8.2. *Cipher* Serpent mempunyai kecepatan secepat DES, dan lebih aman dibanding triple DES tiga kunci.
2.8.3. Keamanannya berdasarkan pada penggunaan kembali komponen-komponen DES dengan penyempurnaan.
2.8.4. Serpent bisa diimplementasikan dalam mode *bitslice* sehingga lebih efektif.

3. *Block Cipher* Rijndael

3.1. Pendahuluan

NIST (National Institute of Standards and Technology) mengadakan kompetisi untuk mendapatkan sebuah standard baru dalam penyandian data yang akan diberi nama AES (Advance Encryption Standard) yang akan menggantikan DES (Data Encryption Standard) yang dianggap sudah tidak aman lagi. *Block Cipher* Rijndael adalah salah satu dari 15 proposal yang diterima. Dan akhirnya *Block Cipher* ini memenangkan kompetisi dan dipakai sebagai AES.

NIST memilih algoritma ini berdasarkan beberapa pertimbangan. Ketika dibandingkan, kombinasi keamanan, kinerja, efisiensi, dan kemudahan implementasi Rijndael terbukti lebih unggul.

Secara spesifik, Rijndael memiliki kekonsistenan dalam performa hardware maupun software secara cukup luas, di berbagai lingkungan komputasi, tanpa terpengaruh dengan mode feedback atau non-feedback. Waktu yang diperlukan untuk *key* setup sangat singkat. Kebutuhan memori algoritma ini cukup rendah sehingga dapat dengan mudah diimplementasian di lingkngan kmputasi yang sangat sederhana (seperti pada smartcard). Sebagai tambahan, algoritma ini dapat diperkuat tanpa mengurangi performanya. Rijndael dirancang dengan beberapa fleksibilitas pada ukuran blok dan kunci, dan algoritma ini mengakomdasi perubahan pada jumlah *round*. Yang terakhir, *round* internal yang dimiliki Rijndael memiliki potensi dalam paralelisme instruction level.

3.2. Deskripsi *Block Cipher* Rijndael

Rijndael adalah *Block Cipher* yang bersifat iteratif. Artinya algoritma ini menggunakan *Cipher* berulang dalam beberapa putaran. Rijndael beroperasi pada unit dasar berupa byte (8 bit) yang diperlakukan sebagai single entity. Algoritma ini juga mengakomodasi penggunaan panjang blok dan panjang kunci yang bervariasi. Ini sesuai dengan persyaratan kompetisi AES yang mengharuskan *Cipher* yang berkompetisi mampu menangani panjang blok dan panjang kunci antara 128 sampai 256 bit. Dalam implementasinya, terdapat dua jenis *Cipher* Rijndael, yaitu *Cipher* dengan panjang blok 128 bit, dan *Cipher* dengan panjang blok 256 bit. Rijndael didesain sedemikian rupa sehingga cukup kuat terhadap kriptanalisis linier dan diferensial. Di dalam algoritma ini terdapat beberapa transformasi yang dibagi dalam beberapa komponen, masing-masing dengan fungsinya masing-masing. Pada bagian ini akan dijelaskan mengenai struktur *Cipher* dan transformasi komponen.

3.2.1. State, Kunci, dan Jumlah Putaran

State didefinisikan sebagai hasil sementara dari proses enkripsi. State diinisialisasi dengan plainteks. Struktur data state berbentuk matriks of byte. Matriks ini terdiri atas empat baris dan N_b buah kolom. N_b bernilai panjang blok dibagi 32.

Kunci juga didefinisikan sebagai matriks of byte dengan jumlah baris 4 dan jumlah kolom N_k. N_k bernilai panjang kunci dibagi 32. Kadang-kadang kinci juga digambarkan sebagai array linier yang komponennya adalah *word* 4 byte. *Word* terdiri atas empat byte pada kolom yang bersesuaian. Ilustrasi state dan kunci dapat dilihat pada Gambar 1.

Panjang Blok dan panjang kunci menentukan banyaknya putaran yang dilakukan pada proses enkripsi. Tabel banyaknya putaran berdasarkan panjang blok dan panjang kunci dapat dilihat pada Tabel 3.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Gambar 1. State dengan N_b=6 dan *Cipher key* dengan N_k=4

		N_b		
		4	6	8
N_k	4	10	12	14
	6	12	12	14
	8	14	14	14

Tabel 3

3.2.2. Transformasi

Pada setiap putaran dilakukan empat jenis transformasi. Transformasi-tansformasi tersebut adalah SubByte, ShiftRow, MixColumn, dan Add*RoundKey*. Pada putaran terakhir, terdapat sedikit perbedaan dalam susunan transformasinya. Transformasi pada putaran terakhir adalah SubByte, ShiftRow, dan Add*RoundKey*. Pada putaran terakhir transformasi MixColumn ditiadakan. Transformasi-transformasi tersebut dikenakan pada masing-masing blok yang ada.

Transformasi SubByte

Transformasi SubByte adalah substitusi non-linier yang beroperasi pada masing-masing byte pada state secara independen. Transformasi ini menggunakan S-box sebagai matriks substitusi. Transformasi SubByte dinotasikan sebagai SubByte(State).

S-box yang sifatnya invertible dikonstruksi dengsn menggabungkan dua transformasi :

a) Lakukan inverse perkalian (multiplicative inverse) pada finite field $GF(2^8)$, elemen {00} dipetakan ke dirinya sendiri.

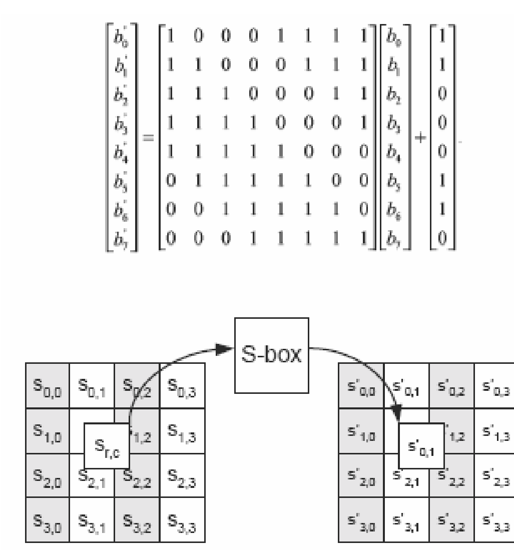
b) Lakukan transformasi affine berikut (pada GF(2)):

$$b_i' = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$$

Untuk $0 \leq i < 8$, dimana b_i adalah bit ke-i dari byte, dan c, adalah bit ke-i dari byte c dengan nilai {63} atau {01100011}. Dalam bentuk matriks digambarkan sebagai berikut :

Gambar 2. S-box

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

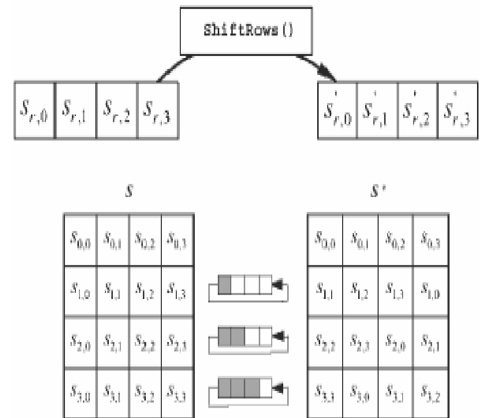


Gambar 3. SubByte() menerapkan S-box pada tiap byte di state

S-box yang digunakan dalam transformasi SubByte() direpresentasikan alam bentuk heksadesimal. Sebagai contoh jukan $S_{1,1}$ = {53}, maka substitusi nilai akan ditentukan dengan irisan dari baris '5' dan kolom '3'.

Transformasi ShiftRow

Pada ShiftRow, tiga baris terakhir dari state digeser ke kiri secara cyclic. Banyaknya pergeseran ditentukan oleh indeks baris. Baris pertama digeser ke kiri sebanyak C_1 byte, baris kedua sebanyak C_2 byte, dan baris ketiga sebanyak C_3 byte. C_1, C_2, dan C_3 beergantung pada panjang blok N_b, yang besarnya dapat dilihat pada Tabel 4. Transformasi ShiftRow dinotasikan sebagai ShiftRow(State).



Gambar 4. Shift Row

N_b	C_1	C_2	C_3
4	1	2	3
6	1	2	3
8	1	3	4

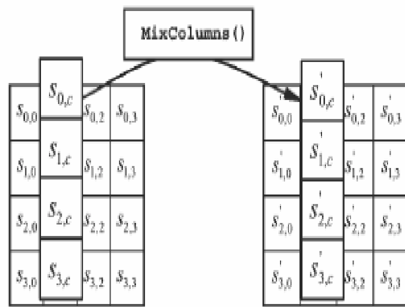
Tabel 4

Transformasi MixColumn

Dalam MixColumn, kolom dari matriks state dianggap sebagai polinomial GF(2^8), dan perkalian modulo $x^4 + 1$ dengan polinomial konstan $c(x) = 3x^3 + x^2 + x + 2$. Ini dapat juga ditulis sebagai perkalian matriks b(x) = x(x) x a(x) =

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \text{ x } \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

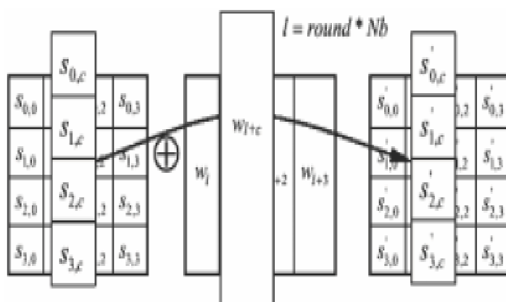
Operasi perkalian matriks diatas yang dilakukan terhadap semua kolom pada state dinotasikan sebagai MixColumn(State).



Gambar 5. MixColumn()

Add*Round Key*

Pada operasi ini, sebuah *round key* (kunci pada satu putaran) dioperasikan pada state dengan operasi bitwise XOR. *Round key* diturunkan dari *Cipher key* melalui proses *key* schedulling. Panjang *round key* sama dengan panjang blok N_b. Transformasi yang mengoperasikan *round key* dengan state dengan operasi XOR dinotasikan dengan Add*RoundKey*(State, *RoundKey*).



Gambar 6. Add*RoundKey*()

3.2.3. Penjadwalan Kunci (*Key* Schedule)

Round Key yang digunakan pada operasi Add*RoundKey*(State, *RoundKey*) diturunkan dari *Cipher key* dengan proses *key* schedule. Proses ini terdiri atas dua komponen yaitu *key* expansion dan *round key* selection.

Prinsip dasarnya adalah sebagai berikut :

a) Banyaknya *round key* sama dengan panjang kunci dikali dengan jumlah putaran ditambah 1. Misal untuk panjang blok 128 bit dan 10 kali putaran, diperlukan 128 x (10+1) = 1408 *round key*.
b) *Cipher key* diekspansi menjadi expanded *key*.
c) *Round key* diambil dari expanded *key* tersebut dengan cara sebagai berikut. *Round key* pertama mengandung N_b *word* pertama, *round key* kedua mengandung N_b *word* berikutnya, dan seterusnya.

Ekspansi Kunci (*Key* Expansion)

Proses ini mengubah *Cipher key* menjadi extended *key*. Expanded *key* adalah array linier yang terdiri atas empat byte dan dinotasikan sebagai $W\left[N_b(N_r+1)\right]$. N_k byte pertama terdiri atas *Cipher key*. Byte-byte lainnya didefinisikan secara rekursif oleh byte-byte lainnya dengan indeks lebih kecil. *Key* schedule tergantung pada nilai N_k. Terdapat dua versi yaitu versi $N_k \leq 6$, versi $N_k > 6$. Versi $N_k \leq 6$ sebagai berikut :

```
KeyExpansion(CipherKey, W) {
For (i = 0; i < Nk;i++) W[i] = CipherKey[i];
For (j = 0; i < Nb(Nr+1);j+=Nk) {
  W[j] = W[j - Nk] XOR SubByte(Rotl(W[j-1])) XOR Rcon[j/Nk];
  For(i=1;i<Nk && i+j < Nb(Nr+1);i++)
        W[i+j] = W[i+j-Nk] XOR W[i+j-1];
  }
}
```

Ketika semua byte *Cipher key* telah digunakan, setiap byte W[i] sama dengan XOR byte sebelumnya W[i-1] dengan N_k posisi sebelumnya W[i-N_k]. Untuk byte di posisi yang

merupakan kelipatan N_k, sebuah transformasi dilakukan terhadap W[i-1] sebelum XOR, lalu di-XOR-kan lagi dengan sebuah konstanta putaran. Transformasi ini meliputi pergeseran byte secara cyclic yang dinotasikan dengan Rotl, diikuti dengan transformasi SubByte. Hal ini dilakukan pada tiap 4 *word*.

Key expansion untuk $N_k > 6$ sangat mirip, namun menggunakan tambahan aplikasi SubByte.

Konstanta putaran independen terhadap N-k dan didefinisikan dengan : Rcon[i] = (RC[i], 0, 0, 0). Dengan RC[0] = 1, RC[1] = 2. RC[i-1] adalah perkalian pada field GF(2^8).

Pemilihan *Round Key*

Round key i diambil dari W[N_bi] sampai W[N_b(i+1)]. *Key* schedule dapat diimplementasikan tanpa penggunaan array W secara eksplisit. Pada implementasi pada komputer dengan RAM sangat kecil, *round key* dapat dihitung pada saat running menggunakan buffer sebesar N_k byte.

3.2.4. *Cipher*

Cipher Rijndael terdiri atas :

a) Initial Add*RoundKey*
b) N_r – 1 putaran
c) Putaran terakhir

Dalam pseudocode dapat dituliskan sebagai berikut :

```
Rijndael(State,
CipherKey) {
KeyExpansion(CipherKe
y, ExpandedKey);
AddRoundKey(State,
ExpandedKey);
For(i=1;i<Nr;i++)
Round(State,
ExpandedKey+Nbi);
FinalRound(State,
ExpandedKey+NbNr);
}
```

Key Expansion bisa dilakukan sebelumnya, sehingga *Cipher* Rijndael dapat dituliskan sebagai berikut :

```
Rijndael(State,
CipherKey) {
AddRoundKey(State,
ExpandedKey);
For(i=1;i<Nr;i++)
Round(State,
ExpandedKey+Nbi);
FinalRound(State,
ExpandedKey+NbNr);
}
```

3.2.5. Inverse *Cipher*

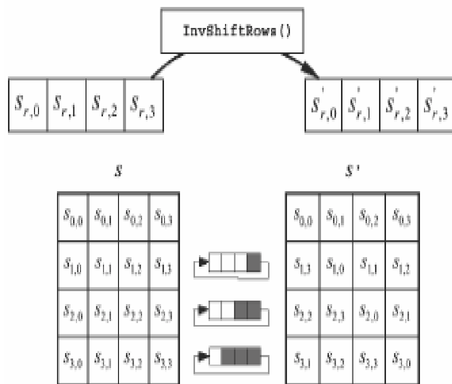
Putaran-putaran transformasi pada *Cipher* Rijndael bukanlah jaringan Feistel. Keuntungan dari *Cipher* yang menggunakan jaringan Feistel adalah bahwa inverse *Cipher*nya hampir sama dengan *Cipher*. Pada jaringan Feistel, hanya urutan kunci putarannya yang harus diinverse tanpa harus mengubah struktur jaringannya. Pada *Cipher* Rijndael tidak dapat dilakukan seperti itu. Pada prinsipnya, proses dekripsi dilakukan dengan mengoperasikan *Cipher*text dengan semua inverse dari komponen transformasi pada urutan yang berkebalikan.

Namun, transformasi pada tiap putaran dan struktur *Cipher*nya telah dirancang untuk mengatasi masalah tersebut secara parsial. Dengan menggunakan beberapa properti aljabar, kita dapat menurunkan representasi persamaan untuk inverse *Cipher*, yang mempunyai struktur yang sama dengan *Cipher*. Ini berarti bahwa putaran inverse *Cipher* ”nampak” sama dengan putaran *Cipher*, kecuali bahwa SubByte, MixColumn, dan ShiftRow telah digantikan dengan masing-masing inverse-nya. *Round key* pada representasi ini berbeda dengan *round key* yang digunakan pada saat enkripsi.

Elemen matriks yang berkorespondensi dengan operasi inverse pada MixColumn mempunyai nilai selain 1, 2, dan 3. Oleh karena it, perkalian matriks tidak dapat dilakukan seefisien pada saat enkripsi. Performance *Cipher* biasanya berkurang sekitar 50%. Turunnya performansi ini dapat diatasi dengan menggunakan tabel tambahan untuk menggantikan perkalian dengan operasi lain yang lebih sedikit memakan memori.

3.2.5.1. Inverse Shift Row

Inverse Shift Row merupakan kebalikan dari transformasi **ShiftRows().** Byte yang terdapat di tiga baris terakhir dari *state* digeser secara siklik (*cyclically shifted*) dengan besaran yang berbeda (*offsets*). Baris pertama, r = 0, tidak digeser. Ketiga baris berikutnya digeser secara siklik sebesar ***Nb*** – *shift*(r,*Nb*) byte, dimana nilai *shift*(r,*Nb*) ditentukan oleh nomor baris



Gambar 7. Inverse Shift Row

3.2.5.2. Inverse Sub Byte

Inverse sub byte adalah inverse dari transformasi substitusi byte, dimana inverse S- box diaplikasikan kepada setiap byte dalam matriks state. Hal ini didapat dengan mengaplikasikan invers dari transformasi affine (3.5.1) lalu dilanjutkan dengan mengambil inverse perkalian (*multiplicative inverse*) dalam GF(2^8).

3.2.5.3. Inverse Mix Column

Adalah invers dari transformasi **MixColumns()**. InvMixColumns() beroperasi dalam matriks *state* per kolom, memperlakukan tiap kolom sebagai polinom 4-*term* (*four-term polynomial*) sebagaimana dibahas dalam Bagian 3.4.3. Kolom-kolom dipandang sebagai polinom atas GF(2^8) dan mengalikan modulo x^4+1dengan polinom tetap $a^{-1}(x)$, yang diperoleh dari

$a^{-1}(x)$ = {0b}x^3 + {0d}x^2 + {09}x + {0e}

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Gambar 8.Inverse Mix Column

3.2.5.4. Inverse Add *Round Key*

Sangat sederhana yakni hanya oerasi XOR karena inverse operasi XOR adalah operasi XOR juga.

3.2.5.5. Inverse Cipher Ekuivalen

Dalam *inverse cipher* sederhana yang diperlihatkan *inverse cipher*, urutan tranformasi berbeda dengan *cipher*, sementara bentuk dari *key schedule* untuk enkripsi dan dekripsi teap sama. Namun demikian, beberapa keunikan dari algoritma AES membolehkan *inverse cipher* ekivalen yang memiliki urutan transformasi yang sama seperti *cipher* (dengan transformasi digantikan oleh inverse transformasinya). Hal ini dicapai

dengan melakukan perubahan pada *key schedule*. Dua keunikan yang memungkinkan *inverse cipher* ekivalen ini adalah:

1. Transformasi **SubBytes()** dan **ShiftRows()** dapat saling menggantikan (*commute*); yaitu, transfomasi SubBytes() yang segera diikuti transformasi ShiftRows() ekivalen dengan transformasi ShiftRows() yang segera diikuti dengan transformasi SubBytes(). Hal yang sama berlaku pula untuk inverse mereka, **InvSubBytes()** dan **InvShiftRows()**.
2. Operasi pengacakan kolom -- **MixColumns()** dan **InvMixColumns()** -- bersifat linear terhadap kolom input, yang artinya

InvMixColumns(state XOR Round Key) = **InvMixColumns(state) XOR InvMixColumns(Round Key)** Keunikan ini membolehkan urutan transformasi InvSubByte() dan InvShiftRows() untuk dibalik. Urutan transformasi AddRoundKey() dan InvMixColumns() dapat pula dipertukarkan, (

Inverse cipher yang ekivalen didefinisikan dengan membalik urutan transformasi InvSubBytes() dan InvShiftRows() yang diperlihatkan pada *pseudocode inverse cipher*, dan dengan membalik urutan dari transformasi AddRoundKey() dan InvMixColumns() yang digunakan dalan *round loop* setelah sebelumnya memodifikasi *key schedule* dekripsi untuk *round* = 1 sampai *Nr*-1 menggunakan transformasi InvMixColumns(). *Nb* word yang pertama dan terakhir dari *key schedule* dekripsi tidak perlu diubah dalam hal ini.

3.3. Kesimpulan

3.3.1. Rijndael adalah block cipher yang sangat cepat.

3.3.2. Dapat diimplementasikan pada Smart Card dengan sedikit kode dan sedikir memori.

3.3.3. Tahan terhadap berbagai timing attack.

3.3.4. Mampu menangani panjang blok yang berbeda.

3.3.5. Kekurangannya adalah bahwa inverse cipher berbeda dari cipher. Inverse cipher biasanya 1,5 atau 2 kali lebih lambah daripada cipher.

4. Kesimpulan

4.1. Kedua algoritma Serpent dan Rijndael adalah algoritma yang kuat terhadap berbagai serangan yang umum diketahui seperti serangan kriptanalisis diferensial dan linier.

4.2. Rijndael unggul dari segi kecepatan dibanding serpent, namun Serpent unggul dari segi keamanan diantaranya karena menggunakan lebih banyak putaran dibanding Rijndael. Hal ini menjadi *overhead* bagi Serpent dalam hal kecepatan.

4.3. Keduanya menggunakan implementasi modern yang dirancang agar dapat berjalan secara efisien pada processor modern, diantaranya dengan mode bitslice.

4.4. Keduanya memiliki kelemahan yaitu bahwa inverse cipher berbeda dengan cipher dan lebih lambat daripada cipher.

5. Referensi

[1] http://www.cl.cam.ac.uk/~rja14/serpent.html

[2]http://www.nist.gov/aes

[3]http://.../serpent.pdf

[4]http://.../rijndael.pdf

[5]Mulyawan, Satria Bakti. IMPLEMENTASI ALGORITMA ENKRIPSI RIJNDAEL (GLADMAN) DALAM OPENSSL-0.9.7.2002.