IMPLEMENTASI ALGORITMA KRIPTOGRAFI CAST-128 TERHADAP TEKS

SKRIPSI

Oleh:
HILMAN FUADY
0510960031-96

PROGRAM STUDI ILMU KOMPUTER
JURUSAN MATEMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS BRAWIJAYA
MALANG
2012


UNIVERSITAS BRAWIJAYA

IMPLEMENTASI ALGORITMA KRIPTOGRAFI CAST-128 TERHADAP TEKS

Skripsi

Sebagai salah satu syarat untuk memperoleh gelar
Sarjana dalam bidang Ilmu Komputer

Oleh:
HILMAN FUADY
0510960031-96

PROGRAM STUDI ILMU KOMPUTER
JURUSAN MATEMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS BRAWIJAYA
MALANG

2012


UNIVERSITAS BRAWIJAYA

LEMBAR PENGESAHAN

IMPLEMENTASI ALGORITMA KRIPTOGRAFI CAST-128 TERHADAP TEKS

Oleh:
HILMAN FUADY
0510960031-96

Setelah dipertahankan di depan Majelis Penguji
pada tanggal 13 Agustus 2012
dan dinyatakan memenuhi syarat untuk memperoleh gelar
Sarjana dalam bidang ilmu komputer

Pembimbing I	Pembimbing II
Drs. Marji, M.T.	**Nurul Hidayat, S.Pd, M.Sc.**
NIP : 196708011992031001	**NIP : 196804302002121001**

Mengetahui,
Ketua jurusan matematika
Fakultas MIPA Universitas Brawijaya

Dr. Abdul Rouf Alghofari, M.Sc.
NIP : 196709071992031001


UNIVERSITAS BRAWIJAYA

LEMBAR PERNYATAAN

Saya yang bertandangan di bawah ini :

Nama	: Hilman Fuady
NIM	: 0510960031
Jurusan	: Matematika
Penulis Tugas Akhir Berjudul	: Implementasi Algoritma Kriptografi CAST-128 terhadap Teks

Dengan ini menyatakan bahwa :

1. Isi dai skripsi yang saya buat adalah benar-benar karya sendiri dan tidak menjiplak karya orang lain. Selain nama-nama yang termaktub di isi dan tertulis di daftar pustaka di skripsi ini.
2. Apabila di kemudian hari ternyata skripsi yang saya tulis terbukti hasil jiplakan, maka saya akan bersedia menanggung segala resiko yang akan saya terima.

   Demikian pernyataan ini dibuat dengan segala kesadaran

Malang, 13 Agustus 2012
Yang menyatakan,

Hilman Fuady
NIM : 0510960031

IMPLEMENTASI ALGORITMA KRIPTOGRAFI CAST-128 TERHADAP TEKS

ABSTRAK

Kriptografi adalah suatu ilmu untuk menyamarkan atau mengacak data dengan metode-metode tertentu, dalam kriptografi terdapat 2 proses yaitu enkripsi dan dekripsi. Enkripsi adalah suatu proses untuk menyamarkan atau mengacak suatu pesan yang masih asli (*plaintext*) menjadi pesan yang sudah teracak (*chipertext*). Salah satu algoritma yang terkenal adalah algoritma kriptografi *CAST-128* yang diciptakan oleh Carlisle Adams dan Stafford Adams pada tahun 1996. *CAST-128* memanfaatkan jaringan *feistel* 16 kali putaran, 64-bit blok teks-asli dibagi menjadi dua bagian yang sama yaitu bagian kiri dan bagian kanan dengan panjang yang sama dengan panjang 32bit, 8x32 entri *S-Box* dan masukan sebuah kunci dengan panjang sampai 128-bit. Terdapat tiga proses dalam *CAST-128* yaitu pembentukan *key* dimana nantinya akan dihasilkan 16 kunci masking dan 16 kunci rotasi yang baru kemudian proses enkripsi dan dekripsi yang memiliki jumlah putaran sebanyak 16. Operasi-operasi yang terdapat diantaranya yaitu XOR, penambahan dan subtitusi S-box.

Pada penelitian ini akan diimplementasikan perangkat lunak dengan menggunakan algoritma *CAST-128* untuk mengetahui kemampuan algoritma *CAST-128* mengenkripsi dan mendekripsi *file* teks. Selain itu juga akan dilakukan analisis terhadap waktu proses dan nilai *avalanche effect*. Nilai *avalanche effect* adalah suatu nilai untuk mengetahui ketahanan algoritma *CAST-128* terhadap serangan.

Dari analisis yang dilakukan di dapatkan waktu proses enkripsi untuk ukuran *file* 50KB rata-rata waktu enkripsi adalah sebesar 0,807s, untuk ukuran *file* 500KB rata-rata waktu enkripsi adalah sebesar 7,662s kemudian rata-rata waktu dekripsi untuk ukuran *file* 50KB adalah sebesar 0,794s, untuk ukuran *file* 500KB adalah sebesar 8,451s. Untuk nilai rata-rata *avalanche effect* yang ideal (45-60%) didapatkan pada pengujian terhadap perubahan *key* yaitu sebesar 50,12% pada perubahan *byte* dan 49,79% pada perubahan bit.

IMPLEMENTATION ALGORITHM CRYPTOGRAPHY CAST-128 TO TEXT

ABSTRACT

Cryptography is the practice and study of hiding data with certain methods. There are two methods for hiding the data, which are encryption and decryption. Encryption is the method of changing the data from plaintext to an unintelligible format (chipertext). CAST-128, which was invented by Carlisle Adams dan Stafford Adams at 1996, is one of the known algorithm for doing such purpose CAST-128 utilize 16 round Feistel network, 64-bit block of the original text is divided into two equal parts, namely the left and the right of equal length to the length 32bit, 8x32 S-Box entry and input key with a length up to 128 -bit. There are three processes in CAST-128, key schedulling which will be produced 16 rotation key and 16 masking key and then process the encryption and decryption with the round of 16. Operations that are among the XOR, addition and substitution of the S-box.

In this research, the algorithm was implemented on a software to know the CAST-128's ability to encrypt and decrypt text files. The results were also be measured based on the time taken to do the process and the value of avalanche effect. The value of avalanche effect is used to measure the defencive ability of CAST-128 to withstand attack.

Based on the anaysis, the time taken to encrypt 50kb files was in average 0,807s. For files with a size of 500kb, the average time was 7,662s. Meanwhile the decryption time taken for decrypting 50kb and 500 files were 0,794s and 8,451s respectively. The ideal avalanche effect's value (45-60%), was 50,12% for a change in the 1 byte key and 49,79% for 1 bit key.

KATA PENGANTAR

Dengan mengucapkan puji syukur kehadirat Tuhan Yang Maha Esa, atas segala rahmat dan karunia-Nya, penulis menyelesaikan skripsi dengan judul : ”IMPLEMENTASI ALGORITMA KRIPTOGRAFI CAST-128 TERHADAP TEKS”.

Penelitian ini ingin mengetahui kemampuan algoritma *CAST-128* dalam melakukan enkripsi dan dekripsi *file* teks dan juga untuk menguji waktu proses dan ketahanan algoritma tersebut.

Mulai perencanaan sampai penyelesaian skripsi ini, penulis telah banyak mendapat bantuan dari berbagai pihak, oleh karena itu dalam kesempatan ini penulis ingin mengucap banyak terimakasih kepada pihak-pihak sebagai berikut :

1. Drs. Marji, M.T, dan Nurul Hidayat S.Pd, M.Sc, selaku pembimbing skripsi yang telah sabar memberi bimbingan dan petunjuk, sehingga skripsi ini bisa terselesaikan.
2. Dr. Abdul Rouf Alghofari, M.Sc, selaku Ketua Jurusan Matematika Fakultas Matematika Dan Ilmu Pengetahuan Alam.
3. Drs. Marji, MT., selaku Ketua Program Studi Ilmu Komputer Jurusan Matematika.
4. Bondan Sapta Prakoso, ST, selaku pembimbing akademik yang telah sabar memberi bimbingan dan petunjuk selama masa studi.
5. Bapak ibu dosen Program Studi Ilmu Komputer Jurusan Matematika yang telah banyak memberikan ilmunya.
6. Para staf TU Jurusan Matematika yang telah banyak membantu segala macam urusan administrasi dan perlengkapan.
7. Dika, Rohmat, Adam, Werdha, Martheen, Nanang dan semua teman-teman di Prodi Ilmu Komputer yang telah memberikan banyak bantuan, masukan dan motivasi.
8. Humairah Fauziah yang terus menerus memberikan semangat untuk menyelesaikan skripsi ini.
9. Bapak dan Ibu orang tua yang telah memberikan dorongan dan doa restu, baik moral maupun material selama penulis menuntut ilmu.
10. Dan semua pihak yang telah membantu dalam pengerjaan skripsi ini yang tidak bisa penulis sebutkan satu persatu.

Semoga Tuhan Yang Maha Esa senantiasa memberikan Rahmat dan Karunia-Nya kepada semua pihak yang telah memberikan segala bantuan tersebut di atas. Skripsi ini tentu saja masih jauh dari sempurna, sehingga penulis dengan senang hati menerima kritik demi perbaikan. Kepada peneliti lain mungkin masih bisa mengembangkan hasil penelitian ini. Akhirnya semoga skripsi ini ada manfaatnya.

Malang, 30 Juli 2012

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR GRAFIK

DAFTAR SOURCECODE

BAB I
PENDAHULUAN

1.1 Latar Belakang

Informasi menentukan hampir setiap elemen dari kehidupan manusia. Informasi sangat penting artinya bagi kehudupan karena tanpa informasi maka hampir semuanya tidak dapat dilakukan dengan baik.

Pada zaman teknologi informasi, suatu pesan atau informasi merupakan suatu aset yang sangat berharga dan harus dilindungi. Kemajuan teknologi komputer membantu semua aspek kehidupan manusia. Dari hal yang sederhana sampai yang sangat rumit sekalipun bisa dikerjakan komputer. Contoh dari kemajuan teknologi komputer yang paling nyata yang dapat digunakan oleh semua orang adalah kecepatan dalam menyampaikan pesan dari tempat yang jauh. E-mail (*electronic mail*) merupakan fasilitas yang ditawarkan oleh kemajuan teknologi. Rupa pesan semakin bermacam-macam, seperti teks, gambar, suara, video, ataupun tabel. Pesan pun bisa dikirim menggunakan jasa pos, kurir, jaringan elektronik (internet), dan bisa disimpan di dalam buku, *hard-disc*, SC, DVD, kaset, dan lain-lain.(Ariyus, 2008).

Dengan adanya kemajuan dalam teknologi informasi, komunikasi dan komputer maka kemudian timbul masalah baru, yaitu masalah keamanan. Masalah keamanan merupakan salah satu aspek terpenting dari sebuah sistem informasi, seperti kejahatan komputer yang mencakup pencurian, penipuan, pemerasan, kompetisi, dan banyak lainnya. Jatuhnya informasi ke pihak lain, misalnya lawan bisnis, dapat menimbulkan kerugian bagi pemilik informasi.Oleh karena itu diperlukan suatu cara untuk menyamarkan suatu pesan yang biasa disebut kriptografi. Pesan adalah data atau informasi yang dapat dimengerti maknanya atau biasa disebut *plaintext.* Pesan dapat berupa data atau informasi yang dikirim (melalui kurir, saluran komunikasi data, dan lain-lain) atau yang disimpan di dalam media perekaman (kertas, *storage*, dan lain-lain). Agar pesan tidak dapat dimengerti maknanya oleh pihak lain, maka pesan disandikan ke bentuk lain. Bentuk pesan yang tersandi disebut *ciphertext*. *Ciphertext* harus dapat ditransformasi kembali menjadi *plaintext* (Munir, 2004).

Terdapat berbagai macam algoritma yang digunakan dalam menjaga keamanan informasi. Salah satu diantaranya adalah Algoritma CAST-128. Algoritma CAST-128 ini disebut sebagai salah satu algoritma kriptografi yang kuat terhadap berbagai macam kriptanalisis, termasuk *differential* dan *linear attack*. Oleh karena itu, tidak salah jika dikatakan bahwa CAST-128 dapat menjadi kandidat kuat untuk pemakaian enkripsi untuk keamanan komunitas internet. (Gunawan. 2006).

Algoritma ini diciptakan pada tahun 1996 oleh Carlisle Adams dan Stafford Tavares dari kanada. CAST-128 termasuk kelas algoritma enkripsi yang merupakan jaringan Feistel. Secara umum algoritma ini mirip dengan algoritma Data Encryption Standart (DES) (Ariyus, 2008). Algoritma ini memanfaatkan jaringan *feistel* 16 kali putaran, 64-bit blok teks-asli dibagi menjadi dua bagian yang sama yaitu bagian kiri dan bagian kanan dengan panjang yang sama dengan panjang 32bit, 8x32 entri *S-Box* dan masukan sebuah kunci dengan panjang sampai 128-bit(Ariyus, 2008).

Dalam penelitian ini akan dibahas dibahas waktu proses dan ketahanan algoritma *CAST-128*. Waktu proses dari algoritma *CAST-128* ini akan terdiri dari waktu proses pembangkitan kunci, waktu untuk enkripsi dan dekripsi *file*. Untuk ketahanan algoritma, metode yang sering dipakai adalah *avalanche effect*. Dimana suatu algoritma akan dikatakan baik dalam hal ketahanan terhadap serangan jika nilai *avalanche effect*-nya berkisar antara 45-60% (Rudianto, 2004). Dengan latar belakang tersebut maka dilakukan penelitian untuk skripsi ini dengan judul "**IMPLEMENTASI ALGORITMA KRIPTOGRAFI CAST-128 TERHADAP TEKS".**

1.2 Rumusan Masalah

Rumusan masalah yang akan dijadikan obyek penelitian yaitu :

1. Bagaimana implementasi algoritma *CAST-128* untuk melakukan enkripsi dan dekripsi *file* teks.
2. Berapa waktu proses algoritma.
3. Berapa nilai ketahanan algoritma dengan menggunakan metode *avalanche effect.*

1.3 Tujuan penelitian

1. Implementasi enkripsi dan dekripsi *file* teks menggunakan algoritma *CAST-128*.
2. Menghitung waktu proses algoritma *CAST-128*.
3. Menghitung nilai *avalanche effect* algoritma *CAST-128* untuk mengetahui ketahanan terhadap serangan.

1.4 Manfaat

Manfaat yang ingin dicapai dari penulisan skripsi ini adalah sistem yang memiliki ketahanan terhadap serangan kriptanalisis yang mampu membantu menyelesaikan masalah keamanan data informasi.

1.5 Batasan Masalah

Pada penelitian ini akan diberi batasan - batasan masalah sebagai berikut :

1. Perangkat lunak akan berupa aplikasi *web*.
2. Uji coba waktu proses enkripsi dan dekripsi akan dilakukan sebanyak lima kali dengan *file* teks **.txt*.
3. Data yang digunakan dalam uji coba waktu proses adalah *file* teks dengan ukuran beragam mulai dari 50KB hingga 500KB.
4. Data yang digunakan dalam uji *avalanche effect* adalah *file* teks dan *file* hasil enkripsi dengan ukuran beragam mulai dari 8*bytes* sampai 40*bytes*.
5. Jenis uji coba *avalanche effect* terdiri dari 3 macam yaitu : uji coba pada *plaintext,* pada *chipertext* dan pada *Key*.
6. Uji coba *avalanche effect* dilakukan pada skala bit dan *byte*. Dengan jumlah bit atau *byte* sebanyak satu dan dua.
7. Posisi perubahan pada saat uji coba *avalanche effect* dibagi menjadi 3 bagian yaitu : posisi bit/*byte* awal, tengah dan akhir.

1.6 Sistematika Penulisan

Buku tugas akhir ini terdiri dari beberapa bab, yang dijelaskan sebagai berikut:

1. **BAB I. PENDAHULUAN**
   Bab ini berisi latar belakang masalah, tujuan dan manfaat pembuatan tugas akhir, permasalahan, batasan masalah, dan sistematika penyusunan tugas akhir.
2. **BAB II. TINJAUAN PUSTAKA**
   Bab ini membahas beberapa teori penunjang yang berhubungan dengan pokok pembahasan dan mendasari pembuatan tugas akhir ini.
3. **BAB III. METODOLOGI DAN PERANCANGAN SISTEM**
   Bab ini membahas desain dari sistem yang akan dibuat meliputi : arsitektur, proses dan antarmuka perangkat lunak.
4. **BAB IV. IMPLEMENTASI DAN PEMBAHASAN**
   Bab ini membahas implementasi dari desain sistem disertai dengan potongan *source code* yang penting dalam aplikasi dan membahas uji coba dari aplikasi yang dibuat dengan melihat *output* yang dihasilkan oleh aplikasi, dan evaluasi untuk mengetahui kemampuan aplikasi.
5. **BAB V. PENUTUP**
   Bab ini berisi kesimpulan dari hasil uji coba yang dilakukan serta saran untuk pengembangan aplikasi selanjutnya.

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:

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

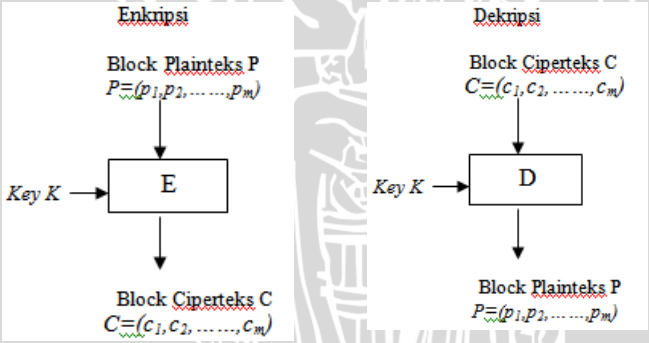
Untuk dekripsinya seperti pada persamaan 2.10:

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

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.

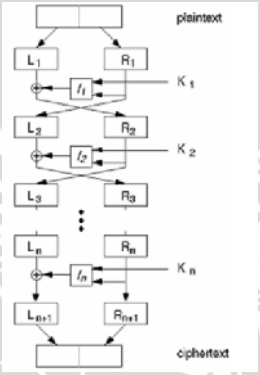
2.5.4 Penjadwalan Kunci

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:

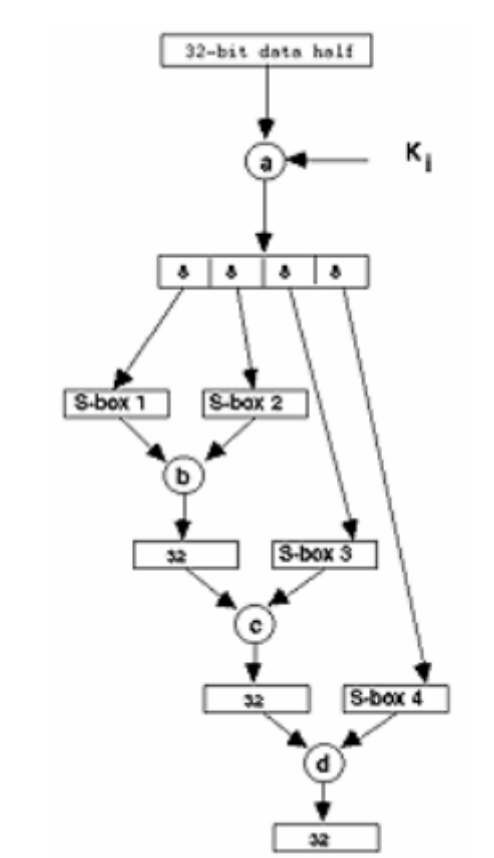


Gambar 2.2 Putaran *Feistel* n putaran.

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}.

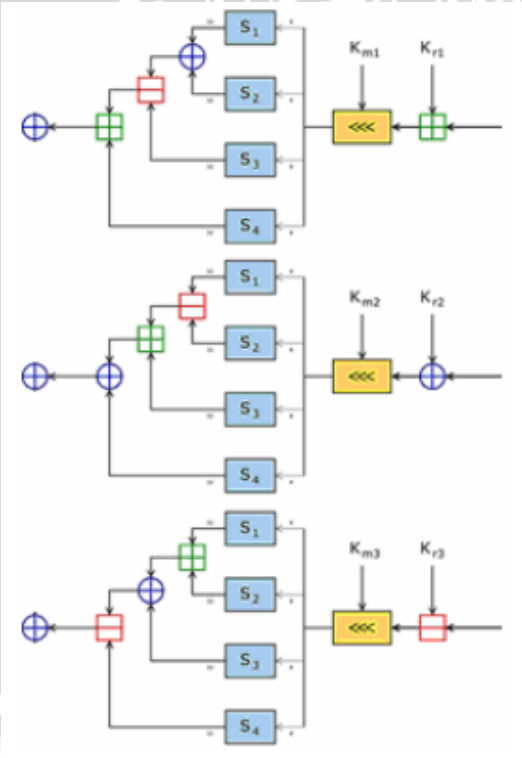
2. Tipe 2.
   - Operasi a adalah XOR dan penggeseran bit ke kiri (*circular left-shift operation*). Bit-bit masukan akan di-XOR-kan dengan Kmi (sesuai putaran ke-i) dan akan digeser ke kiri sebanyak Kri (sesuai putaran ke-i pula).
   - Operasi b adalah pengurangan bit modulo 2^{32}.
   - Operasi c adalah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

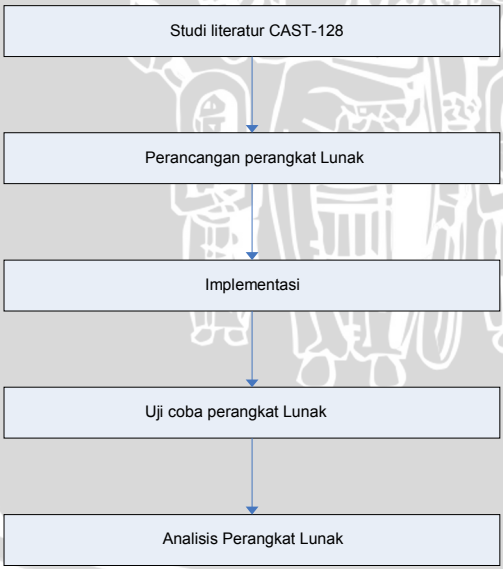

UNIVERSITAS BRAWIJAYA

BAB III
METODOLOGI DAN PERANCANGAN

Pada bab metode dan perancangan ini akan dibahas mengenai metode yang digunakan dalam pembuatan dan analisis perangkat lunak enkripsi dan dekripsi dengan algoritma *CAST-128* antara lain:

1. Melakukan studi literatur terhadap algoritma *CAST-128*, dengan memahami secara lebih dalam konsep enkripsi dan dekripsinya.
2. Melakukan perancangan perangkat lunak, meliputi proses input, proses penghitungan *subkey*, proses enkripsi, proses dekripsi dan perancangan *interface*.
3. Implementasi hasil perancangan dengan bahasa pemrograman PHP.
4. Melakukan uji coba enkripsi dan dekripsi *file* dengan perangkat lunak yang telah dibuat.
5. Melakukan beberapa uji coba yang berkaitan dengan waktu proses dan *avalanche effect.*

Langkah-langkah tersebut dapat dilihat seperti pada gambar 3.1.



Gambar 3.1 Diagram alir pembuatan perangkat lunak

3.1 Analisis Perangkat Lunak

Pada subbab analisis ini akan dibahas dekripsi dan batasan untuk perangkat lunak.

3.1.1 Deskripsi Perangkat Lunak

Perangkat lunak enkripsi dan dekripsi *file* teks ini bermanfaat untuk *user* yang ingin mengenkripsi *file* teks yang dimilikinya agar tidak diketahui oleh orang lain. Sistem ini akan menerima *input* berupa *key* dan *file* teks. Kemudian *file* teks tersebut akan dienkripsi menggunakan *key* yang telah diinputkan *user*. Dan hanya bisa didekripsi dengan *key* yang sama.

Algoritma *CAST-128* merupakan algoritma yang beroperasi dalam *digit* biner. Untuk itu setiap *file* teks yang akan dienkripsi maupun didekripsi harus diubah dahulu ke dalam bentuk biner.

Secara umum, proses-proses yang akan dilakukan saat user menggunakan perangkat lunak adalah sebagai berikut :

1. *User* diminta untuk menginputkan *file* teks yang akan dienkripsi.
2. User diminta untuk memasukkan kata kunci (*key)*. Kemudian dilakukan penghitungan *subkey* agar *key* yang di*input*kan *user* tadi siap dipakai untuk enkripsi dan dekripsi seperti yang telah dibahas pada subbab 2.5.3 pada proses pembangkitan kunci internal.
3. Kemudian dari data biner yang didapat dari *file* input akan dilakukan enkripsi dengan menggunakan *subkey*. Proses enkripsi seperti yang telah dijelaskan pada subbab 2.5.4.
4. *Cipher file* bisa didekripsi kembali menggunakan *key* yang sama. Proses dekripsi seperti yang telah dibahas pada subbab 2.5.5. *Flowchart* untuk proses enkripsi dan dekripsi secara umum ini dapat dilihat pada gambar 3.2 dan 3.3.

3.1.2 Batasan Perangkat Lunak

1. *File* yang akan dienkripsi harus berformat *.txt*.
2. Kata kunci menggunakan karakter ASCII.

3.2 Perancangan Perangkat Lunak

Pada subbab ini akan dibahas perancangan berbagai macam perancangan dari perangkat lunak diantaranya adalah : proses *input*, proses penghitungan *subkey,* proses enkripsi dan dekripsi.

3.2.1 Perancangan Proses *Input* Perangkat Lunak

Inputan dari user terdiri dari dua bagian yaitu *inputan* saat melakukan enkripsi *file* dan inputan saat melakukan dekripsi *file*. Dimana inputan saat melakukan enkripsi *file* terdiri dari *input file* teks yang akan dienkripsi dan kunci untuk melakukan enkripsi. Sedangkan *inputan* saat melakukan dekripsi terdiri dari *input file cipher* yang akan di dekripsi dan kunci untuk melakukan dekripsi. *File* teks yang akan di*input*kan *user* berupa *file* yang berektensi *.txt* dan kunci yang digunakan berupa karakter-karakter yang ada pada *keybord* komputer sepanjang 5 sampai 16 karakter. Proses ini pertama kali diawali dengan pembukaan *file* dalam bentuk *binary*, kemudian di baca hingga akhir *file* dan terakhir *file* tersebut ditutup. *Flowchart* untuk proses *input* perangkat lunak ini dapat dilihat pada gambar 3.4.

3.2.2 Perancangan Proses Penghitungan *subkey*

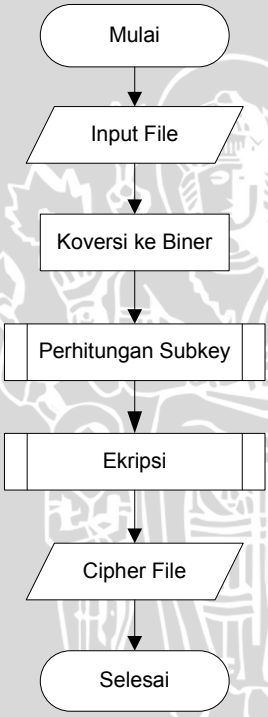
Dalam algortima *CAST-128*, kunci yang dimasukkan user tidak langsung digunakan dalam enkripsi dan dekripsi pesan. Akan tetapi diproses dahulu dengan *S-Box* yang sudah diinisialisasi sebelumnya.

Key yang bisa diterima oleh perangkat lunak ini sepanjang 40-128 bit atau 5-16 karakter ASCII. Apabila *key* melebihi 16 karakter maka karakter yang melebihi 16 itu dianggap tidak ada.

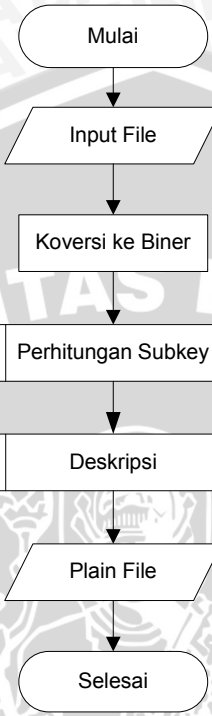
Proses penghitungan *subkey* pada perangkat lunak ini akan seperti berikut :

1. *Inputkey* yang berupa string akan di konversi ke dalam bentuk biner.
2. Perangkat lunak akan menginisialisasi *variabel array* yaitu *array* satu dimensi sebanyak 8 S-box (S1,S2,...,S8) yang sudah ditentukan.
3. *Key* akan dibagi ke dalam blok-blok berisi 8-bit atau 1 karakter dalam *array* satu dimensi X (x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, xA,xB, xC, xD, xE, xF).

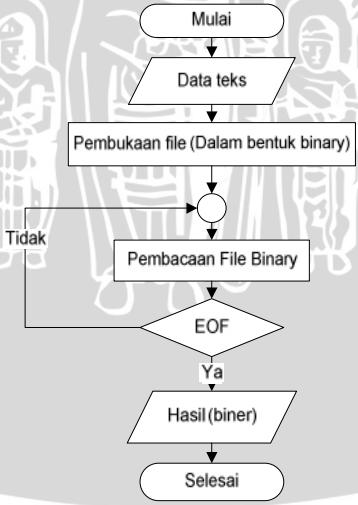
4. Perhitungan *Key Masking* dan *Key* Rotasi menggunakan Algoritma pada sub-bab 2.5.3
5. Pembentukan variabel x0x1x2x3 dengan cara menggabungkan x0 *shift left* 24, x1 *shift left* 16, x2 *shift left* 8, x3. Dengan cara yang sama bentuk x4x5x6x7, x8x9xAxB dan xCxDxExF.
6. Pembentukan variabel z0, z1, z2, z3 dengan z0z1z2z3 *shift right* 24 + binary 11111111, z0z1z2z3 *shift left* 16 + binary 11111111, z0z1z2z3 *shift left* 8 + binary 11111111, z0z1z2z3 + binary 11111111. Dengan cara yang sama bentuk z4, z5, z6, z7, z8, z9, zA, zB,zC, zD, zE dan zF.



Gambar 3.2 *Flowchart* proses enkripsi dalam perangkat lunak



Gambar 3.3 *Flowchart* proses dekripsi dalam perangkat lunak



Gambar 3.4 *Flowchart* proses *inputfile* pada perangkat lunak

3.2.3 Perancangan Proses Enkripsi

Proses enkripsi pada perangkat lunak akan akan seperti berikut:

1. *Plainfile* di-*input*-kan, apabila panjang *plainfile* bukan merupakan kelipatan 8 karakter maka dilakukan penambahan karakter *null* di kanan sampai panjang *plainfile* merupakan kelipatan 8 karakter.
2. *Input plainfile* yang berupa *string* akan di konversi ke dalam bentuk biner.
3. Enkripsi akan dilakukan pada blok yang berisi 64-bit atau 8 karakter.
4. Perangkat lunak akan membagi blok tersebut menjadi dua bagian sama besar yaitu bagian kiri (L) sebesar 32-bit dan bagian kanan (R) sebesar 32-bit. Kemudian L dan R tersebut akan dienkripsi dengan algoritma *CAST-128*. Langkah ke lima sampai delapan akan memperlihatkan proses enkripsi L dan R dengan tiga jenis fungsi *CAST*.
5. Dilakukan 16 iterasi apabila panjang *key* lebih dari 10 karakter atau 80-*byte* dan 12 iterasi apabila panjang *key* kurang atau sama dengan 10 karakter atau 80-*byte*. Dalam setiap iterasi hanya menerima 32-bit masukan. Pada iterasi ganjil menerima masukan bagian kanan (R) dan hasil dari fungsi *CAST* akan di XOR kan dengan bagian kiri (L). Dan sebaliknya untuk iterasi genap. *Flowchart* proses enkripsi perblok dapat dilihat pada gambar 3.7.
6. Penggunaan kuci *masking* dan kunci rotasi berutan sesuai urutan iterasi.
7. Didalam iterasi tersebut ada 3 jenis fungsi *CAST*, iterasi ke-1, 4, 7, 10, 13 dan 16 menggunakan fungsi tipe 1, iterasi ke-2, 5, 8, 11 dan 14 menggunakan fungsi tipe 2, iterasi ke-3, 6, 9 12 dan 15 menggunakan fungsi tipe 3. Secara garis besar fungsi CAST dibagi menjadi 4 operasi. Operasi a akan memproses 32bit *half* data dan dibagi menjadi 4 bagian. Masing-masing 8-bit dijadikan masukan untuk *S-box* 1, *S-box* 2, *S-box* 3, dan *S-box* 4. Operasi b menggabung *S-box* 1 dan *S-box* 2. Operasi c menggabung hasil operasi b dengan *S-box* 3. Dan operasi d menggabung hasil operasi c dengan *S-box* 4. *Flowchart* fungsi F ini dapat dilihat pada gambar 3.8.
8. Fungsi *CAST* tipe 1.

a. Operasi a menambahkan kunci *masking* dengan bit-bit masukan dan menggeser ke kiri sebanya kuci rotasi.
b. Operasi b adalah XOR.
c. Operasi c adalah pengurangan.
d. Operasi d adalah penjumlahan.

9. Fungsi *CAST* tipe 2.
   a. Operasi a meng XOR kan kunci *masking* dengan bit-bit masukan dan menggeser ke kiri sebanya kuci rotasi.
   b. Operasi b adalah pengurangan.
   c. Operasi c adalah penjumlahan.
   d. Operasi d adalah XOR.
10. Fungsi *CAST* tipe 3.
   a. Operasi a mengurangkan kunci *masking* dengan bit-bit masukan dan menggeser ke kiri sebanya kuci rotasi.
   b. Operasi b adalah penjumlahan.
   c. Operasi c adalah XOR.
   d. Operasi d adalah pengurangan.
11. L dan R digabungkan kembali.
12. Konversi hasil penggabungan L dan R ke dalam string sehingga menghasilkan *cipherfile*. *Flowchart* untuk proses enkripsi ini dapat dilihat pada gambar 3.6.

3.2.4 Perancangan Proses Dekripsi

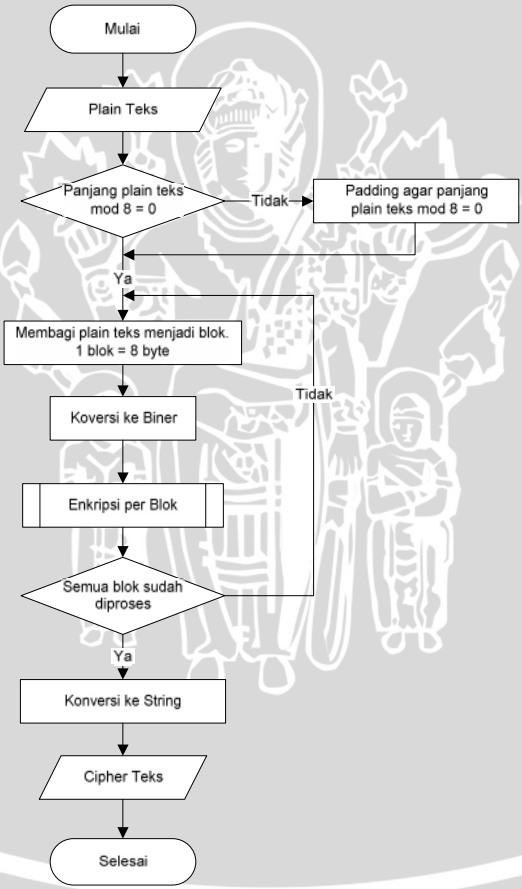
Proses dekripsi pada perangkat lunak akan akan seperti berikut:

1. *Cipherfile* di-*input*-kan, apabila panjang *cipherfile* bukan merupakan kelipatan 8 karakter maka dilakukan penambahan karakter *null* di kanan sampai panjang *cipherfile* merupakan kelipatan 8 karakter.
2. *Input cipherfile* yang berupa string akan di konversi ke dalam bentuk biner.
3. Dekripsi akan dilakukan pada blok yang berisi 64-bit atau 8 karakter.
4. Perangkat lunak akan membagi blok tersebut menjadi dua bagian sama besar yaitu bagian kiri (L) sebesar 32-bit dan bagian kanan (R) sebesar 32-bit. Kemudian L dan R tersebut

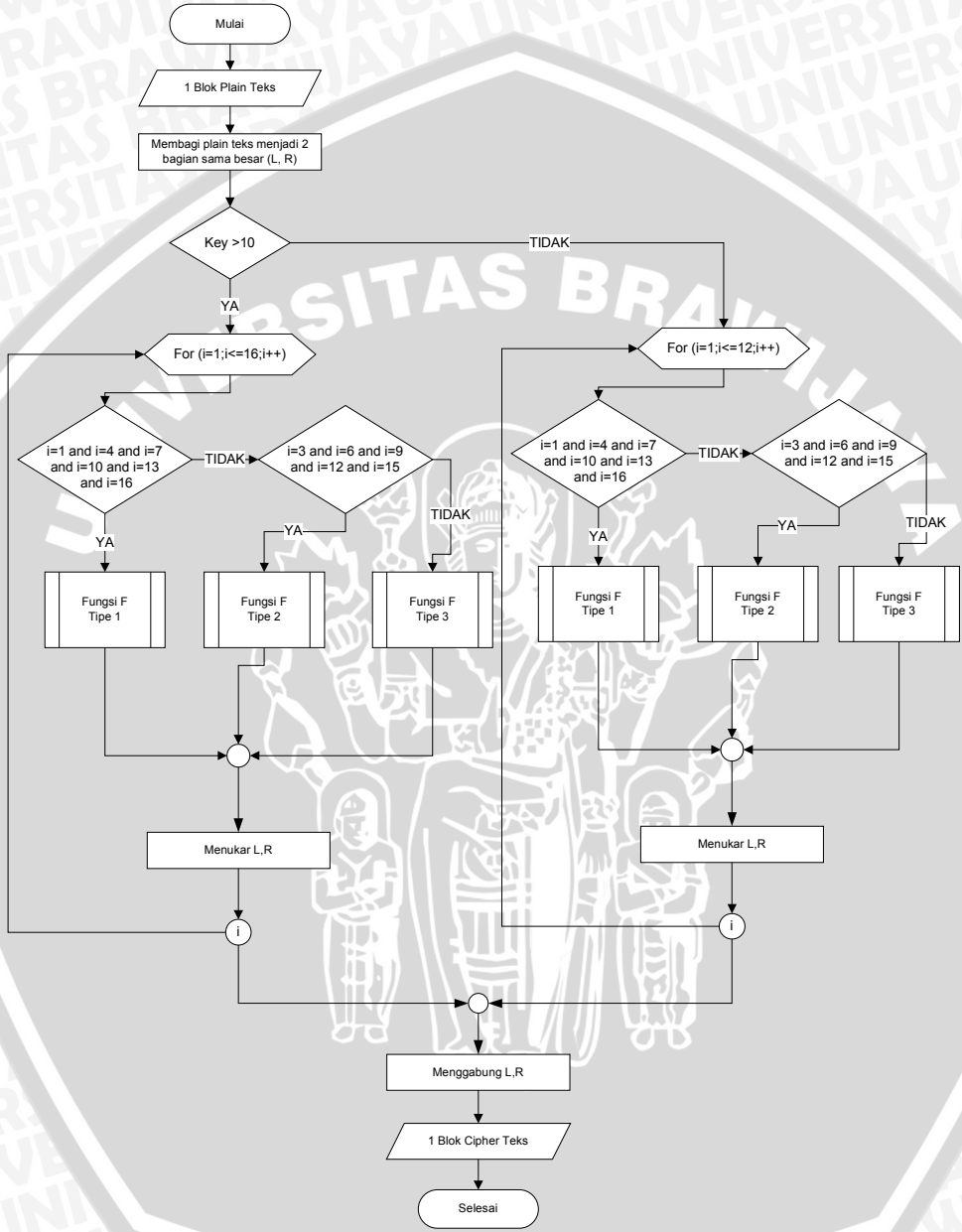
akan dienkripsi dengan algoritma *CAST-128*. Langkah ke lima sampai delapan akan memperlihatkan proses enkripsi XL dan XR dengan tiga jenis fungsi *CAST*.

5. Dilakukan 16 iterasi apabila panjang *key* lebih dari 10 karakter atau 80-*byte* dan 12 iterasi apabila panjang *key* kurang atau sama dengan 10 karakter atau 80-*byte*. Dalam setiap iterasi hanya menerima 32-bit masukan. Pada iterasi ganjil menerima masukan bagian kiri (L) dan hasil dari fungsi *CAST* akan di XOR kan dengan bagian kanan (R). Dan sebaliknya untuk iterasi genap. *Flowchart* proses dekripsi perblok dapat dilihat pada gambar 3.12.
6. Penggunaan kuci masking dan kunci rotasi dimulai dari kunci paling akhir beurut mundur.
7. Di dalam iterasi tersebut ada 3 jenis fungsi *CAST*, iterasi ke-1, 4, 7, 10, 13 dan 16 menggunakan fungsi tipe 1, iterasi ke-2, 5, 8, 11 dan 14 menggunakan fungsi tipe 2, iterasi ke-3, 6, 9 12 dan 15 menggunakan fungsi tipe 3. Secara garis besar fungsi *CAST* dibagi menjadi 4 operasi. Operasi a akan memproses 32bit *half* data dan dibagi menjadi 4 bagian. Masing-masing 8-bit dijadikan masukan untuk *S-box* 1, *S-box* 2, *S-box* 3, dan *S-box* 4. Operasi b menggabung *S-box* 1 dan *S-box* 2. Operasi c menggabung hasil operasi b dengan *S-box* 3. Dan operasi d menggabung hasil operasi c *dengan S-box* 4. *Flowchart* fungsi ini dapat dilihat pada gambar 3.8, gambar 3.9 dan gambar 3.10.
8. Fungsi *CAST* tipe 1.
    a. Operasi a menambahkan kunci *masking* dengan bit-bit masukan dan menggeser ke kiri sebanya kuci rotasi.
    b. Operasi b adalah XOR.
    c. Operasi c adalah pengurangan.
    d. Operasi d adalah penjumlahan.
9. Fungsi *CAST* tipe 2.
    a. Operasi a meng XOR kan kunci *masking* dengan bit-bit masukan dan menggeser ke kiri sebanya kuci rotasi.
    b. Operasi b adalah pengurangan.
    c. Operasi c adalah penjumlahan.
    d. Operasi d adalah XOR.

10. Fungsi *CAST* tipe 3.
    a. Operasi a mengurangkan kunci masking dengan bit-bit masukan dan menggeser ke kiri sebanya kuci rotasi.
    b. Operasi b adalah penjumlahan.
    c. Operasi c adalah XOR.
    d. Operasi d adalah pengurangan.
11. L dan R digabungkan kembali.
12. Konversi hasil penggabungan L dan R ke dalam string sehingga menghasilkan *plainfile. Flowchart* untuk proses dekripsi ini dapat dilihat pada gambar 3.11.

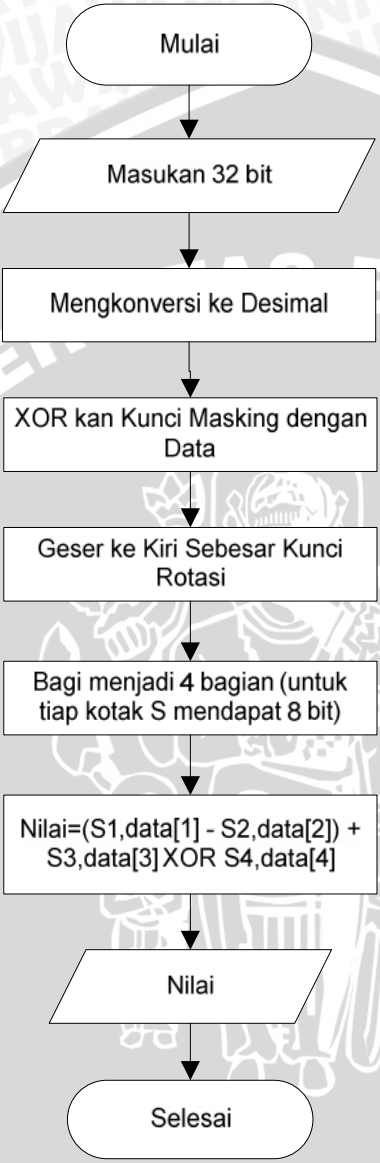


Gambar 3.5 *Flowchart* proses enkripsi

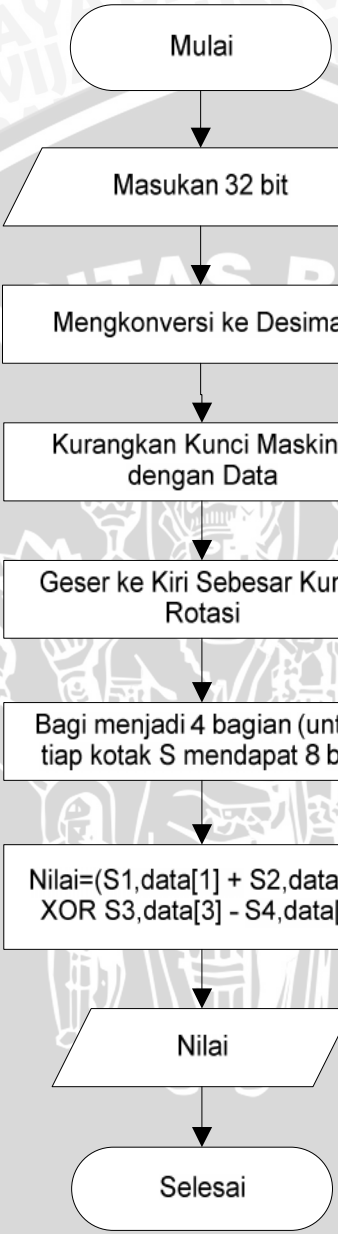


Gambar 3.6 *Flowchart* Enkripsi PerBlok

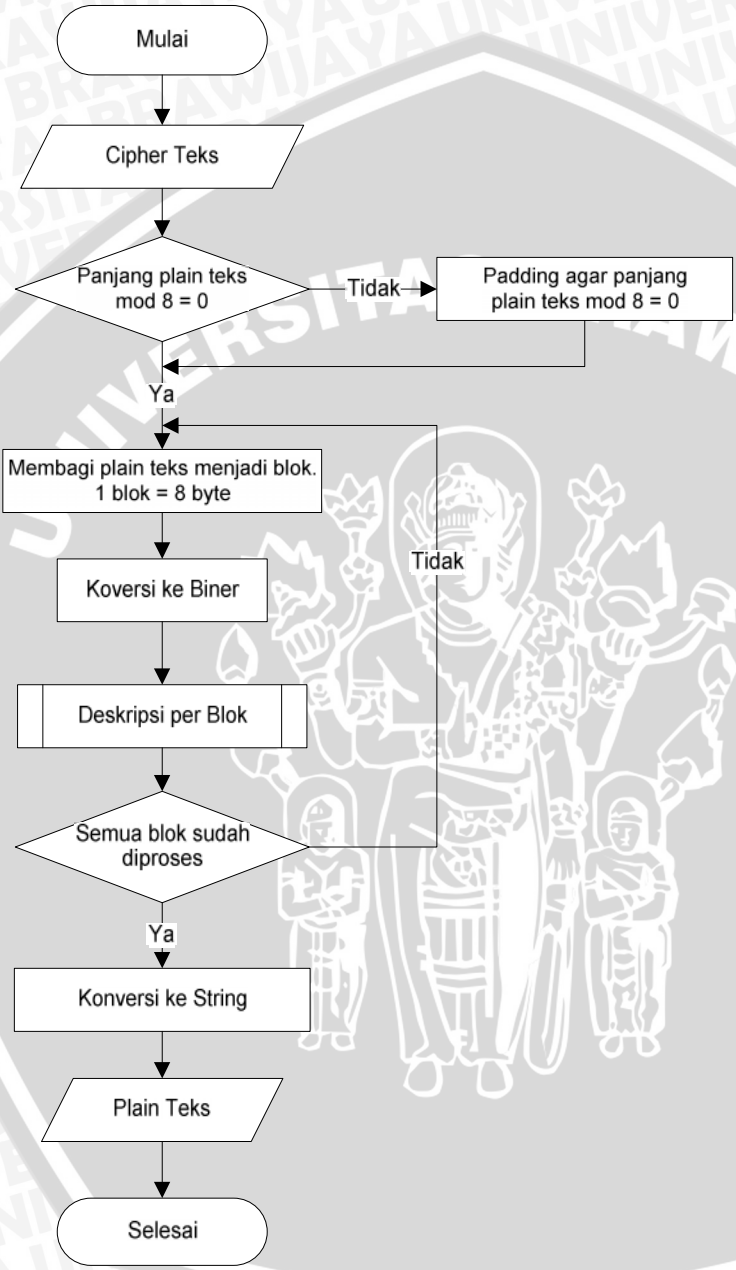
Gambar 3.7 *Flowchart* Fungsi F Tipe 1



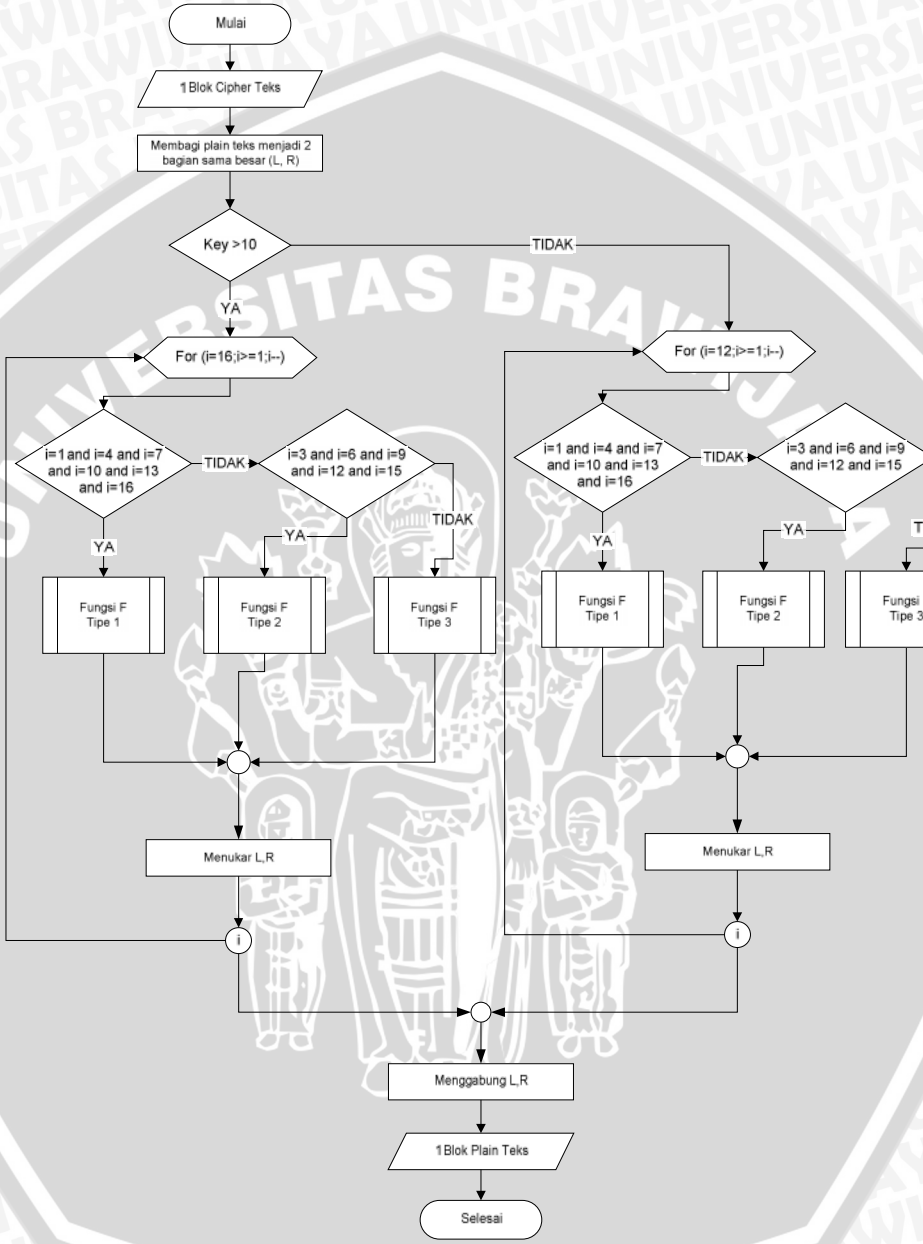
Gambar 3.8 *Flowchart* Fungsi F Tipe 2



Gambar 3.9 *Flowchart* Fungsi F Tipe 3



Gambar 3.10 *Flowchart* Proses dekripsi



Gambar 3.11 *Flowchart* Dekripsi PerBlok

3.3 Perhitungan Matematis

3.3.1 Perhitungan *Subkey*

Dalam perhitungan *subkey* pada algoritma *CAST-128* ini akan diperlukan Kunci Eksternal sepanjang 128 bit. Kemudian dibagi menjadi 16 *bytes* subkunci, yaitu: x0x1x2x3x4x5x6x7x8x9xAxBxC xDxExF, Dimana x0 menunjukkan *most significant byte*(MSB) dan xF menunjukkan *least significant byte*(LSB). Kemudian perhitungannya menggunakan algortima seperti pada subbab 2.5.3.

Kemudian sebagai contoh penghitungan *subkey* akan digunakan *key* "CAST-128". Langkah-langkah penghitungan untuk mendapatkan *subkey* adalah sebagai berikut :

1. *Key* "CAST-128" terdiri dari 8 karakter atau 64-bit. Karena kurang dari 16 karakter atau 128-bit maka perlu dilakukan *padding* menjadi "CAST-12800000000".
2. Ubah *Key* ke dalam bentuk biner. Dapat dilihat pada tabel 3.1. hasil konversi *key* ke bentuk biner adalah sebagai berikut :
   01000011010000010101001101010100001011010011000100110010001111000000000000000000000000000000000000000000000000000000000000000000000000

Tabel 3.1 Konversi *key* ke biner

string	heksadesimal	biner
C	43	01000011
A	41	01000001
S	53	01010011
T	54	01010100
-	2d	00101101
1	31	00110001
2	32	00110010
8	38	00111000
0	0	00000000
0	0	00000000
0	0	00000000
0	0	00000000
0	0	00000000
0	0	00000000
0	0	00000000
0	0	00000000

3. Perhitungan Keym(kunci *masking*) dan Keyr(Kunci Rotasi) :
   x0x1x2x3 = 01000011010000010101001101010100
   x4x5x6x7 = 00101101001100010011001000111000
   x8x9xAxB = 00000000000000000000000000000000
   xCxDxExF = 00000000000000000000000000000000

   z0z1z2z3=x0x1x2x3 XOR s5[xD] XOR s6[xF] XOR s7[xC] XOR s8[xE] XOR s7[x8]
   x0x1x2x3 = 01000011010000010101001101010100
   s5[xD] = 01111110110010010000110000000100
   s6[xF] = 11110110111110101000111110011101
   s7[xC] = 10000101111000000100000000011001
   s8[xE] = 11100010000101100011000000001101
   s7[x8] = 10000101111000000100000000011001
   z0z1z2z3 = 00101001011001001110000011000000

   z4z5z6z7 = x8x9xAxB XOR s5[z0] XOR s6[z2] XOR s7[z1] XOR s8[z3] XOR s8[xA]
   x8x9xAxB = 00000000000000000000000000000000
   s5[z0] = 01101000101011111000000001000000
   s6[z2] = 01100101001111010111111001101010
   s7[z1] = 11110111110111101011101110000101
   s8[z3] = 01110011000011101101111010111100
   s8[xA] = 11100010000101100011000000001101
   z4z5z6z7 = 01101011010101001010101100011110

   z8z9zAzB = xCxDxExF XOR s5[z7] XOR s6[z6] XOR s7[z5] XOR s8[z4] XOR s5[x9]
   xCxDxExF = 00000000000000000000000000000000
   s5[z7] = 10001101101110100001110011111110
   s6[z6] = 01100000011000101110001110010111
   s7[z5] = 10000010000111011011101010011111
   s8[z4] = 00000110011101101010001110101011
   s5[x9] = 01111110110010010000110000000100
   z8z9zAzB = 00010111011110101110101001011001

   zCzDzEzF = x4x5x6x7 XOR s5[zA] XOR s6[z9] XOR s7[zB] XOR s8[z8] XOR s6[xB]
   x4x5x6x7 = 00101101001100010011001000111000
   s5[zA] = 01101011101011000011000001111111

s6[z9] = 00110011001100111011000010010100
s7[zB] = 00000000100110110100111111000011
s8[z8] = 00000100100111101110110111111101
s6[xB] = 11110110111110101000111110011101 □
zCzDzEzF = 10000111010100011001111101110000

keym[1] = s5[z8] XOR s6[z9] XOR s7[z7] XOR s8[z6] XOR s5[z2]
s5[z8] = 01001100101010101101111011111111
s6[z9] = 00110011001100111011000010010100
s7[z7] = 00010010100001101011111011001111
s8[z6] = 00100010001101100001001110111101
s5[z2] = 01101100111101101110010001111001 □
keym[1] = 00100011110111110010011101100000

keym[2] = s5[zA] XOR s6[zB] XOR s7[z5] XOR s8[z4] XOR s5[z6]
s5[zA] = 01101011101011000011000001111111
s6[zB] = 11100111011011111111101111100111
s7[z5] = 10000010000111011011101010011111
s8[z4] = 00000110011101101010001110101011
s6[z6] = 01100000011000101110001110010111 □
keym[2] = 01101000110010100011000100111011

keym[3] = s5[zC] XOR s6[zD] XOR s7[z3] XOR s8[z2] XOR s7[z9]
s5[zC] = 10110000110101110000111010111010
s6[zD] = 01010011110000001000010000111010
s7[z3] = 10011000100000111111111001100110
s8[z2] = 10101010000100101110010011110010
s7[z9] = 11110100001100001100100001111101 □
keym[3] = 00100101101101100101100001101001

keym[4] = s5[zE] XOR s6[zF] XOR s7[z1] XOR s8[z0] XOR s8[zC]
s5[zE] = 01001001100100011111100001000000
s6[zF] = 01001110110001110101101110010101
s7[z1] = 11110111110111101011101110000101
s8[z0] = 11111101101010100011001101011101
s8[zC] = 00010101000101101000001011101011 □

keym[4] = 00011000001101001010100111100110

x0x1x2x3 = z8z9zAzB XOR s5[z5] XOR s6[z7] XOR s7[z4] XOR s8[z6] XOR s7[z0]
z8z9zAzB = 00010111011110101110101001011001
s5[z5] = 00001101000000011110100110000000
s6[z7] = 10101000100010000110000101001010
s7[z4] = 01010001111100001100100000000010
s8[z6] = 00100010001101100001001110111101
s7[z0] = 10110011101100101110100111001110 □
x0x1x2x3 = 01110010100001110101000011100010

x4x5x6x7 = z0z1z2z3 XOR s5[x0] XOR s6[x2] XOR s7[x1] XOR s8[x3] XOR s8[z2]
z0z1z2z3 = 00101001011001001110000011000000
s5[x0] = 11110110010101001110111111000101
s6[x2] = 11001110101100100010100101101111
s7[x1] = 01011101110110100000000000110011
s8[x3] = 11100000111101101010001001111010
s8[z2] = 10101010000100101110010011110010 □
x4x5x6x7 = 00000110101111000110000011010001

x8x9xAxB = z4z5z6z7 XOR s5[x7] XOR s6[x6] XOR s7[x5] XOR s8[x4] XOR s5[z1]
z4z5z6z7 = 01101011010101001010101100011110
s5[x7] = 00111111010010000001110110000111
s6[x6] = 10001101111001001011111110011001
s7[x5] = 00010111110111001011000011110000
s8[x4] = 00001110001001000001011000000000
s5[z1] = 01100011011001110011011110110110 □
x8x9xAxB = 10100011011001111001100001000110

xCxDxExF = zCzDzEzF XOR s5[xA] XOR s6[x9] XOR s7[xB] XOR s8[x8] XOR s6[z3]
zCzDzEzF = 10000111010100011001111101110000
s5[xA] = 01100110101101001111000010100011
s6[x9] = 10111000011110000011010010111111
s7[xB] = 11001111000110011101111101011000
s8[x8] = 11100101100000001011001111100110
s6[z3] = 00001000101010010011000011110110 □

xCxDxExF = 01111011101011010000011100100100

keym[5] = s5[x3] XOR s6[x2] XOR s7[xC] XOR s8[xD] XOR s5[x8]
s5[x3] = 11010000110011101111101001100101
s6[x2] = 11001110101100100010100101101111
s7[xC] = 11001000101001110001001100000010
s8[xD] = 00110111110111111001001100101011
s5[x8] = 01000100010010001001010000000110 □
keym[5] = 10100101010011001100011100100101

keym[6] = s5[x1] XOR s6[x0] XOR s7[xE] XOR s8[xF] XOR s6[xD]
s5[x1] = 10110000110101110000111010111010
s6[x0] = 01000010110100010101110110011001
s7[xE] = 00100000001010001101101000011111
s8[xF] = 10110011000000011101010000001010
s6[xD] = 10110110110010000101001010000011 □
keym[6] = 11010111111001110000111110110101

keym[7] = s5[x7] XOR s6[x6] XOR s7[x8] XOR s8[x9] XOR s7[x3]
s5[x7] = 00111111010010000001110110000111
s6[x6] = 10001101111001001011111110011001
s7[x8] = 00010101111000001101011111111001
s8[x9] = 10000101100111000001010110100101
s7[x3] = 00111000101000001100000001111101 □
keym[7] = 00011010011100001010000000111111

keym[8] = s5[x5] XOR s6[x4] XOR s7[xA] XOR s8[xB] XOR s8[x7]
s5[x5] = 10110011110011011100111101110010
s6[x4] = 11101100111011010101110010111100
s7[xA] = 00101000111001110100111001000001
s8[xB] = 01011000110010110111111000000111
s8[x7] = 00000000110110100110110101110111 □
keym[8] = 00101111110101101100111011111111

z0z1z2z3=x0x1x2x3 XOR s5[xD] XOR s6[xF] XOR s7[xC] XOR s8[xE] XOR s7[x8]

x0x1x2x3 = 01110010100001110101000011100010
s5[xD] = 01111101000101100001101110111010
s6[xF] = 11010000110101010001100100110010
s7[xC] = 11001000101001110001001100000010
s8[xE] = 00000101001011001110100010110101
s7[x8] = 00010101111000001101011111111001 □
z0z1z2z3 = 00000111001011110111111000100100

z4z5z6z7 = x8x9xAxB XOR s5[z0] XOR s6[z2] XOR s7[z1] XOR s8[z3] XOR s8[xA]
x8x9xAxB = 10100011011001111001100001000110
s5[z0] = 00010111001100010001011001111111
s6[z2] = 11101001101010011101100001001000
s7[z1] = 11010011010011010111010100010110
s8[z3] = 10110011000000011101010000001010
s8[xA] = 11001101011111101101011100000101 0 □
z4z5z6z7 = 11110000110011100101100101100111

z8z9zAzB = xCxDxExF XOR s5[z7] XOR s6[z6] XOR s7[z5] XOR s8[z4] XOR s5[x9]
xCxDxExF = 01111011101011010000011100100100
s5[z7] = 10001110110010100011011011000001
s6[z6] = 11100111011011111111011111100111
s7[z5] = 11000011111101011110000110010100
s8[z4] = 11101001011101100010010110100101
s5[x9] = 10001110110010100011011011000001 □
z8z9zAzB = 10110110010000010011100011110010

zCzDzEzF = x4x5x6x7 XOR s5[zA] XOR s6[z9] XOR s7[zB] XOR s8[z8] XOR s6[xB]
x4x5x6x7 = 00000110101111000110000011010001
s5[zA] = 11011111000100111010001010000000
s6[z9] = 10101010100100101000001000100011
s7[zB] = 11010001100110001000111100110101
s8[z8] = 10110100100010100010010001100101
s6[xB] = 11101111111010001100001101101110 □
zCzDzEzF = 11111001110001110010100001001100

keym[9] = s5[z3] XOR s6[z2] XOR s7[zC] XOR s8[zD] XOR s5[z9]

s5[z3] = 11000001000001101110110011010111
s6[z2] = 11101001101010011101100001001000
s7[zC] = 10000100101100011101001101110000
s8[zD] = 11111001110000010111101010001111
s5[z9] = 11111110001110001101001110011001 □
keym[9] = 10101011111001110100111011111001

keym[10] = s5[z1] XOR s6[z0] XOR s7[zE] XOR s8[zF] XOR s6[zC]
s5[z1] = 01001110001010011111111010100111
s6[z0] = 00110010010101010101001110101100
s7[zE] = 00010000011101111000100110111110
s8[zF] = 00111100111100011101001011100010
s6[zC] = 01001001110010010010111101010100 □
keym[10] = 00011001001100111101100100000011

keym[11] = s5[z7] XOR s6[z6] XOR s7[z8] XOR s8[z9] XOR s7[z2]
s5[z7] = 10001110110010100011011011000001
s6[z6] = 11100111011011111111101111100111
s7[z8] = 00000101001111100101011000011010
s8[z9] = 00101001100110001101111100000100
s7[z2] = 10111110100010111001110100101101 □
keym[11] = 11111011100010001101100100010101

keym[12] = s5[z5] XOR s6[z4] XOR s7[zA] XOR s8[zB] XOR s8[z6]
s5[z5] = 01111000010011010110101100010111
s6[z4] = 00111011010011001011111110011111
s7[zA] = 01000111101111000010100000101001
s8[zB] = 00001110001001010010010001001011
s8[z6] = 00101101010111100011001101010100 □
keym[12] = 00100111110001101110101110111110

x0x1x2x3 = z8z9zAzB XOR s5[z5] XOR s6[z7] XOR s7[z4] XOR s8[z6] XOR s7[z0]
z8z9zAzB = 10110110010000010011100011110010
s5[z5] = 01111000010011010110101100010111
s6[z7] = 10111000011110000011010010111111
s7[z4] = 10010001110110100101010111110100

s8[z6] = 00101101010111100011001101010100
s7[z0] = 00100000001010001101101000011111 □
x0x1x2x3 = 11101010110110001101101111100101

x4x5x6x7 = z0z1z2z3 XOR s5[x0] XOR s6[x2] XOR s7[x1] XOR s8[x3] XOR s8[z2]
z0z1z2z3 = 00000111001011110111111000100100
s5[x0] = 01101011101011000011000001111111
s6[x2] = 01111111100101111100010110101011
s7[x1] = 11010011101101011010101100110100
s8[x3] = 00001101011101110001110000101011
s8[z2] = 01111100110100010110111011111100 □
x4x5x6x7 = 10110001000001110101001000010011

x8x9xAxB = z4z5z6z7 XOR s5[x7] XOR s6[x6] XOR s7[x5] XOR s8[x4] XOR s5[z1]
z4z5z6z7 = 11110000110011100101100101100111
s5[x7] = 11001100111101011100000110000000
s6[x6] = 11111110100010010011011001010101
s7[x5] = 00100000001010001101101000011111
s8[x4] = 11101111001101000111100011011101
s5[z1] = 01001110001010011111111010100111 □
x8x9xAxB = 01000011100001111111001011010111

xCxDxExF = zCzDzEzF XOR s5[xA] XOR s6[x9] XOR s7[xB] XOR s8[x8] XOR s6[z3]
zCzDzEzF = 11111001110001110010100001001100
s5[xA] = 01110101010101010100010000101100
s6[x9] = 01011001001010111111001010100000
s7[xB] = 11000110000111100100010110111110
s8[x8] = 10011011011011011111010010010001
s6[z3] = 11010000110101010001100100110010 □
xCxDxExF = 01011000000111100011110100101101

keym[13] = s5[x8] XOR s6[x9] XOR s7[x7] XOR s8[x6] XOR s5[x2]
s5[x8] = 00000110001001000000011111101010
s6[x9] = 01011001001010111111001010100000
s7[x7] = 11100010010011101111000110111101
s8[x6] = 11001001101011111111011001111011

s5[x3] = 00001111111101101111100011110011 □
keym[13] = 01111011000110010000000110001111

keym[14] = s5[xA] XOR s6[xB] XOR s7[x5] XOR s8[x4] XOR s5[x6]
s5[xA] = 01110101010101010100010000101100
s6[xB] = 01010100111101001010000010000100
s7[x5] = 00100000001010001101101000011111
s8[x4] = 11101111001101000111100011011101
s6[x7] = 11100100011000100101111001111110 □
keym[14] = 00001010110111110001100000010100

keym[15] = s5[xC] XOR s6[xD] XOR s7[x3] XOR s8[x2] XOR s7[x8]
s5[xC] = 10111100111100111111000010101010
s6[xD] = 10101000100010000110000101001010
s7[x3] = 01011000100111011101001110010000
s8[x2] = 11000111111010111000111100110111
s7[x8] = 01110010111111000000100000011010 □
keym[15] = 11111001111100011100010101011101

keym[16] = s5[xE] XOR s6[xF] XOR s7[x1] XOR s8[x0] XOR s8[xD]
s5[xE] = 11001000101011011110110110110011
s6[xF] = 01001100011111110100010001001000
s7[x1] = 11010011101101011010101100110100
s8[x0] = 00110110100110010111101100000111
s8[xD] = 01110010110111110001100100011011 □
keym[16] = 00010011001000010110000011010011

z0z1z2z3=x0x1x2x3 XOR s5[xD] XOR s6[xF] XOR s7[xC] XOR s8[xE] XOR s7[x8]
x0x1x2x3 = 11101010110110001101101111100101
s5[xD] = 10001101101110100001110011111110
s6[xF] = 01001100011111110100010001001000
s7[xC] = 10010010010101000100101010001011
s8[xE] = 10101110011000111010111111110010
s7[x8] = 01110010111111000000100000011010 □
z0z1z2z3 = 01100101110101100110111000110000

z4z5z6z7 = x8x9xAxB XOR s5[z0] XOR s6[z2] XOR s7[z1] XOR s8[z3] XOR s8[xA]
x8x9xAxB = 01000011100001111111001011010111
s5[z0] = 01010000111101011011011000010110
s6[z2] = 11100010001000100000101010111110
s7[z1] = 00111101000111111100111001101111
s8[z3] = 10000010111100111101000001010101
s8[xA] = 00001110001001010010010001001011 □
z4z5z6z7 = 01000000100110010111010000001110

z8z9zAzB = xCxDxExF XOR s5[z7] XOR s6[z6] XOR s7[z5] XOR s8[z4] XOR s5[x9]
xCxDxExF = 01011000000111100011110100101101
s5[z7] = 10101100111101100010010000111010
s6[z6] = 01111011011011100010011111111111
s7[z5] = 11000010011000010000101011001010
s8[z4] = 10111011110100110101000001001001
s5[x9] = 10110000110101110000111010111010 □
z8z9zAzB = 01000110111000110110101011010001

zCzDzEzF = x4x5x6x7 XOR s5[zA] XOR s6[z9] XOR s7[zB] XOR s8[z8] XOR s6[xB]
x4x5x6x7 = 10110001000001110101001000010011
s5[zA] = 11111011100010000111101000110111
s6[z9] = 01010000000101111101010101011011
s7[zB] = 00111100100110010111111001111110
s8[z8] = 01011000110010110111111000000111
s6[xB] = 01010100111101001010000010000100 □
zCzDzEzF = 00101010001111100101110110000010

keyr[1] = 0x1F && (s5[z8] XOR s6[z9] XOR s7[z7] XOR s8[z6] XOR s5[z2])
s5[z8] = 10010000110001111001010100000101
s6[z9] = 01010000000101111101010101011011
s7[z7] = 01001011001101101001010111110010
s8[z6] = 10110000111111100001001101001100
s5[z2] = 10110110111101011000100111011110 □
0x1F = 00000000000000000000000000011111&&
keyr[1] = 00000000000000000000000000011110

keyr[2] = s5[zA] XOR s6[zB] XOR s7[z5] XOR s8[z4] XOR s5[z6]
s5[zA] = 11111011100010000111101000110111
s6[zB] = 00111000000101001111001000000000
s7[z5] = 11000010011000010000101011001010
s8[z4] = 10111011110100110101000001001001
s6[z6] = 01111011011011100010011111111111 □
0x1F = 00000000000000000000000000011111&&
keyr[2] = 00000000000000000000000000001011

keyr[3] = s5[zC] XOR s6[zD] XOR s7[z3] XOR s8[z2] XOR s7[z9]
s5[zC] = 11101101000011001001111001010110
s6[zD] = 00001000101000011001100001100110
s7[z3] = 01001110011110110011101011111111
s8[z2] = 00100100001001011001111111010111
s7[z9] = 11111101000101100000011011110010 □
0x1F = 00000000000000000000000000011111&&
keyr[3] = 00000000000000000000000000001010

keyr[4] = s5[zE] XOR s6[zF] XOR s7[z1] XOR s8[z0] XOR s8[zC]
s5[zE] = 01100100111011100010110101111110
s6[zF] = 11110011101001011111011001110110
s7[z1] = 00111101000111111100111001101111
s8[z0] = 00111110001101111000000101100000
s8[zC] = 00010111011011110100001111101000 □
0x1F = 00000000000000000000000000011111&&
keyr[4] = 00000000000000000000000000001111

x0x1x2x3 = z8z9zAzB XOR s5[z5] XOR s6[z7] XOR s7[z4] XOR s8[z6] XOR s7[z0]
z8z9zAzB = 01000110111000110110101011010001
s5[z5] = 11000000111100010110010010001010
s6[z7] = 11110011100011111111011100110010
s7[z4] = 00001010100101100001001010001000
s8[z6] = 10110000111111100001001101001100
s7[z0] = 01100001111111100000001100111100 □
x0x1x2x3 = 10101110000010111111101110010001

x4x5x6x7 = z0z1z2z3 XOR s5[x0] XOR s6[x2] XOR s7[x1] XOR s8[x3] XOR s8[z2]
z0z1z2z3 = 01100101110101100110111000110000
s5[x0] = 10011100101011011001000000010000
s6[x2] = 01110001001010001010010001010100
s7[x1] = 01010000111100011000101110000010
s8[x3] = 10011111001100100110010001000010
s8[z2] = 00100100001001011001111111010111 □
x4x5x6x7 = 01100011101101010010101001100011

x8x9xAxB = z4z5z6z7 XOR s5[x7] XOR s6[x6] XOR s7[x5] XOR s8[x4] XOR s5[z1]
z4z5z6z7 = 01000000100110010111010000001110
s5[x7] = 00000100101001011100001010000100
s6[x6] = 11101000011010111110001111011010
s7[x5] = 00011001110111100111111010101110
s8[x4] = 01000110101001010010010101100100
s5[z1] = 01011111010001101011000000101010 □
x8x9xAxB = 10101100011010101011111010110000

xCxDxExF = zCzDzEzF XOR s5[xA] XOR s6[x9] XOR s7[xB] XOR s8[x8] XOR s6[z3]
zCzDzEzF = 00101010001111100101110110000010
s5[xA] = 00100000100100110110000001111001
s6[x9] = 11001001110001001100100000111011
s7[xB] = 11010110100110010010100101101110
s8[x8] = 11011101000001101100101010100010
s6[z3] = 00001000001110010001100110100111 □
xCxDxExF = 11000000110011110000111110101011

keyr[5] = s5[x3] XOR s6[x2] XOR s7[xC] XOR s8[xD] XOR s5[x8]
s5[x3] = 10010100111101110100101111000000
s6[x2] = 01110001001010001010010001010100
s7[xC] = 10011000100000111111111001100110
s8[xD] = 00011010000000000111001001101110
s5[x8] = 00111100101001011101011100010111 □
0x1F = 00000000000000000000000000011111&&
keyr[5] = 00000000000000000000000000001011

keyr[6] = s5[x1] XOR s6[x0] XOR s7[xE] XOR s8[xF] XOR s6[xD]
s5[x1] = 01110011010110101011101000000000
s6[x0] = 00111100110000101010110011111011
s7[xE] = 10110010100001110000011111011110
s8[xF] = 00100010001101100001001110111101
s6[xD] = 00110110000101000000000010111100 □
0x1F = 00000000000000000000000000011111&&
keyr[6] = 00000000000000000000000000000100

keyr[7] = s5[x7] XOR s6[x6] XOR s7[x8] XOR s8[x9] XOR s7[x3]
s5[x7] = 00000100101001011100001010000100
s6[x6] = 11101000011010111110001111011010
s7[x8] = 10010011110100101001101000100010
s8[x9] = 11011011000001111011101000001100
s7[x3] = 01000011100000000101000011100011 □
0x1F = 00000000000000000000000000011111&&
keyr[7] = 00000000000000000000000000010011

keyr[8] = s5[x5] XOR s6[x4] XOR s7[xA] XOR s8[xB] XOR s8[x7]
s5[x5] = 00010111011011010100100001101111
s6[x4] = 11011010010110100010011011000000
s7[xA] = 10011110101000101001010011111011
s8[xB] = 01010111000101011111011010110111
s8[x7] = 01000110101001010010010101100100 □
0x1F = 00000000000000000000000000011111&&
keyr[8] = 00000000000000000000000000000111

z0z1z2z3=x0x1x2x3 XOR s5[xD] XOR s6[xF] XOR s7[xC] XOR s8[xE] XOR s7[x8]
x0x1x2x3 = 10101110000010111111101110010001
s5[xD] = 01011000111010111011000101101110
s6[xF] = 01100000011000101110001110010111
s7[xC] = 10011000100000111111111001100110
s8[xE] = 00110111110111011101110111111100
s7[x8] = 10010011110100101001101000100010 □
z0z1z2z3 = 10101010000011100001000011010000

z4z5z6z7 = x8x9xAxB XOR s5[z0] XOR s6[z2] XOR s7[z1] XOR s8[z3] XOR s8[xA]
x8x9xAxB = 10101100011010101011111010110000
s5[z0] = 10110011000110110010101111100001
s6[z2] = 00110011111100010100100101100001
s7[z1] = 01001011001101101001010111110010
s8[z3] = 00010001010000000011000010010010
s8[xA] = 00111000110101111110010110110010□
z4z5z6z7 = 01001110001000011001110011100010

z8z9zAzB = xCxDxExF XOR s5[z7] XOR s6[z6] XOR s7[z5] XOR s8[z4] XOR s5[x9]
xCxDxExF = 11000000110011110000111110101011
s5[z7] = 11010000110011101111101001100101
s6[z6] = 00001110111100111100100010100110
s7[z5] = 01110101011001011011110111100100
s8[z4] = 01000010010011110111011000011000
s5[x9] = 11111011100010000111101000110111□
z8z9zAzB = 11010010010100001000110010100011

zCzDzEzF = x4x5x6x7 XOR s5[zA] XOR s6[z9] XOR s7[zB] XOR s8[z8] XOR s6[xB]
x4x5x6x7 = 01100011101101010010101001100011
s5[zA] = 10010100110010100000101101010110
s6[z9] = 11001110101100100010100101101111
s7[zB] = 00010101111000001101011111111001
s8[z8] = 01001010000011001101110101100001
s6[xB] = 01001110100011110000001001010010□
zCzDzEzF = 00101000101011100000000010010000

keyr[9] = s5[z3] XOR s6[z2] XOR s7[zC] XOR s8[zD] XOR s5[z9]
s5[z3] = 01000100000010010100111110000101
s6[z2] = 00110011111100010100100101100001
s7[zC] = 00010000011101111000100110111110
s8[zD] = 11000100001001001000001010001001
s5[z9] = 10010001000111100111001110011010□
0x1F = 00000000000000000000000000011111&&
keyr[9] = 00000000000000000000000000001001

keyr[10] = s5[z1] XOR s6[z0] XOR s7[zE] XOR s8[zF] XOR s6[zC]
s5[z1] = 10101100111101100010010000111010
s6[z0] = 10001000110010011000111110001000
s7[zE] = 10000101111000000100000000011001
s8[zF] = 10110110111100101100111100111011
s6[zC] = 11111101010000010001100101111110 □
0x1F = 00000000000000000000000000011111&&
keyr[10] = 00000000000000000000000000001110

keyr[11] = s5[z7] XOR s6[z6] XOR s7[z8] XOR s8[z9] XOR s7[z2]
s5[z7] = 11010000110011101111101001100101
s6[z6] = 00001110111100111100100010100110
s7[z8] = 01011110010011111001010100000100
s8[z9] = 10011101000101111101111011100111
s7[z2] = 10100000101111110111100111101110 □
0x1F = 00000000000000000000000000011111&&
keyr[11] = 00000000000000000000000000010110

keyr[12] = s5[z5] XOR s6[z4] XOR s7[zA] XOR s8[zB] XOR s8[z6]
s5[z5] = 10111010100011110110010111001011
s6[z4] = 10011010011010011010000000101111
s7[zA] = 10100011110110011101001010110000
s8[zB] = 11100101100000001011001111100110
s8[z6] = 11001110101001001101010000101000 □
0x1F = 00000000000000000000000000011111&&
keyr[12] = 00000000000000000000000000011010

x0x1x2x3 = z8z9zAzB XOR s5[z5] XOR s6[z7] XOR s7[z4] XOR s8[z6] XOR s7[z0]
z8z9zAzB = 11010010010100001000110010100011
s5[z5] = 10111010100011110110010111001011
s6[z7] = 01010001101001000111011111101010
s7[z4] = 11000110111001101111101000010100
s8[z6] = 11001110101001001101010000101000
s7[z0] = 11010111000101101110011101000000 □
x0x1x2x3 = 11100110001011110101011111111110

x4x5x6x7 = z0z1z2z3 XOR s5[x0] XOR s6[x2] XOR s7[x1] XOR s8[x3] XOR s8[z2]
z0z1z2z3 = 10101010000011100001000011010000
s5[x0] = 10100000100111000111111101110000
s6[x2] = 10101001101010011001001110000111
s7[x1] = 11010011010011010111010100010110
s8[x3] = 10001101101100101010001010000011
s8[z2] = 11011110100110101101111010110001 □
x4x5x6x7 = 00100011010111101111010100000011

x8x9xAxB = z4z5z6z7 XOR s5[x7] XOR s6[x6] XOR s7[x5] XOR s8[x4] XOR s5[z1]
z4z5z6z7 = 01001110001000011001110011100010
s5[x7] = 10100110001100110111100100010001
s6[x6] = 11110101010001001110110111101011
s7[x5] = 00011010111011000011110010101001
s8[x4] = 10101010010000000010000101100100
s5[z1] = 10101100111101100010010000111010 □
x8x9xAxB = 00000001000011000011000111101111

xCxDxExF = zCzDzEzF XOR s5[xA] XOR s6[x9] XOR s7[xB] XOR s8[x8] XOR s6[z3]
zCzDzEzF = 00101000101011100000000010010000
s5[xA] = 00000010110100011100000000000000
s6[x9] = 00011010101101101010011010111000
s7[xB] = 11110011100000010111100100010100
s8[x8] = 10111011110111011111111111111100
s6[z3] = 11101000100000010110111101001010 □
xCxDxExF = 10010000000101001000111110001010

keyr[13] = s5[x8] XOR s6[x9] XOR s7[x7] XOR s8[x6] XOR s5[x2]
s5[x8] = 00101100011011100111010010111001
s6[x9] = 00011010101101101010011010111000
s7[x7] = 11001111110001100101011010010011
s8[x6] = 00001101001000000101100111010001
s5[x3] = 01101101010001111101111000001000 □
0x1F = 00000000000000000000000000011111&&
keyr[13] = 00000000000000000000000000001011

keyr[14] = s5[xA] XOR s6[xB] XOR s7[x5] XOR s8[x4] XOR s5[x6]
s5[xA] = 00000010110100011100000000000000
s6[xB] = 10011010101101101111011011110101
s7[x5] = 00011010111011000011110010101001
s8[x4] = 10101010010000000010000101100100
s6[x7] = 11100010001100110111111101111100 □
0x1F = 00000000000000000000000000011111&&
keyr[14] = 00000000000000000000000000000100

keyr[15] = s5[xC] XOR s6[xD] XOR s7[x3] XOR s8[x2] XOR s7[x8]
s5[xC] = 01011000000010100010010010011111
s6[xD] = 10100011000010001110101010011001
s7[x3] = 10010000011100010110111101001011
s8[x2] = 00000110110011010001101011111000
s7[x8] = 00110011001010111111010101100111 □
0x1F = 00000000000000000000000000011111&&
keyr[15] = 00000000000000000000000000010010

keyr[16] = s5[xE] XOR s6[xF] XOR s7[x1] XOR s8[x0] XOR s8[xD]
s5[xE] = 01100001100001001011010110111110
s6[xF] = 01111101101001001100111011000000
s7[x1] = 11010011010011010111010100010110
s8[x0] = 01100111110011011011000101010110
s8[xD] = 10111011001001000011101000001111 □
0x1F = 00000000000000000000000000011111&&
keyr[16] = 00000000000000000000000000010001

3.3.2 Perhitungan Enkripsi

Setelah melakukan perhitungan *subkey* langkah selanjutnya adalah melakukan enkripsi pada sebuah *string*. Contoh *string* yang akan dienkripsi adalah ”KOMPUTER”. Karena *string* ini terdiri dari 64-bit maka tidak dilakukan *padding*.

Langkah-langkah perhitungan matematisnya adalah sebagai berikut :

1. Konversi *string* diatas kedalam bentuk biner. Hasil konversi biner *string* ”KOMPUTER” yaitu :

01001011010011111010011010101000001010101010100010
0010101010010.

Proses konversi dimulai dari konversi string ke heksadesimal kemudian heksadesimal ke biner. Proses tersebut bisa dilihat pada tabel 3.2.

Tabel 3.2 konversi plaintext kedalam biner

STRING	HEKSADESIMAL	BINER
K	4b	01001011
O	4f	01001111
M	4d	01001101
P	50	01010000
U	55	01010101
T	54	01010100
E	45	01000101
R	52	01010010

2. Bagi kedalam 2 blok masing-masing 32-bit.

L=01001011010011111010011010101000000
R=01010101010101000100010101010010

3. Lakukan enkripsi *CAST-128*.

- Iterasike-1 menggunakan Fungsi *CAST* tipe 1

Tambahkan R dengan Kunci *Masking* ke-1
I = keym[1] + R
keym[1]=00100011110111110010011101100000
R=01010101010101000100010101010010 +
I=01111001001100110110110010110010

I=I <<< keyr[1]
I=10011110010011001101101100101100

F=((s1[Ia] XOR s2[Ib])- s3[Ic])- s3[Id]
s1[Ia]=10101010010100011010011110011011
s2[Ib]=01010111010100111000101011010101 □

F=00100000000101001001111100010111
s3[Ic]=00001100111011010110001111010000 -
F=00111010010010110001011110110000
s4[Id]=10000010011110110110100011010000 +
F=10111001000111111001101110110000

L= L XOR F
L=01001011010011110100110101010000
F=10111001000111111001101110110000□
L=11110010010100001101011011100000

- Iterasi ke-2 Menggunakan Fungsi *CAST* tipe 2

XOR kan L dengan Kunci *Masking* ke-2
I = keym[2] XOR L
keym[2]=01101000110010100011000100111011
L=11110010010100001101011011100000□
I=10011010100110101110011111011011

I=I <<<keyr[2]
I=11010111001111101101110011010100

F=((s1[Ia] - s2[Ib])+ s3[Ic]) XOR s3[Id]
s1[Ia]=10111100001100000110111011011001
s2[Ib]=10100010110101001011011101101101 -
F=00011011010011001000111101100000
s3[Ic]=01111100011000111011001011001111+
F=11101010011011000111101100110010
s4[Id]=00110101101110100011111001001010□
F=11001011100111000010011110001100

R= R XOR F
R=01010101010101000100010101010010
F=11001011100111000010011110001100□
R=10011110110010000110001011011110

- Iterasi ke-3 Menggunakan Fungsi *CAST* tipe 3

Kurangkankan Kunci *Masking* ke-3 dengan R

I = keym[3] - R
keym[3]=00100101101101100101100001101001
R=10011110110010000110001011011110 -
I=10000110111011011111010110001011
I=I <<< keyr[3]
I=10110111110101100010111000011011

((s1[Ia] + s2[Ib])XOR s3[Ic])- s3[Id]
s1[Ia]=10101100001110010101011100001010
s2[Ib]=11000011011110110100110100001001 +
F=10011011101000010111111111000110
s3[Ic]=00100010101100001100000001010100 □
F=00010101110010101010110100000111
s4[Id]=00100010010101000000111100101111 -
F=11101000101100010100110100001001
L= L XOR F
L=11110010010100001101011011100000
F=11101000101100010100110100001001 □
L=00011010111000011001101111101001

.....
.....
.....
.....

Hasil iterasi ke-12
L = 11010110000110101111101010011001
R = 11111111111100101111010001101100

Tukar L dan R
L= 11111111111100101111010001101100
R= 11010110000110101111101010011001

Gabungkan L dan R
11111111111100101111010001101100110101100001101011111
101010011001
Konversi ke *heksadesimal* dan *string*
Heksadesimal = fff2f46cd61afa99
String = ÿòôlÖú™

3.3.3 Perhitungan Dekripsi

Selanjutnya akan dilakukan dekripsi dari *stringciphertext* di atas dengan *key* yang sama saat enkripsi yaitu ”CAST-128”. Langkah-langkah dekripsi akan seperti berikut :

1. Konversi *ciphertext* kedalam bentuk biner. Hasil konversi *chipertext* ke biner adalah sebagai berikut :
   11010110000110101111101010011001111111111111110010111
   1010001101100

2. Bagi ke dalam 2 blok masing-masing 32-bit
   L=11111111111100101111010001101100
   R=11010110000110101111101010011001

   Tukar L dan R
   L= 11010110000110101111101010011001
   R= 11111111111100101111010001101100

3. Lakukan enkripsi algoritma *CAST-128* dengan menggunakan urutan iterasi dan Kunci yang terbalik dari proses enkripsi. Iterasi dimulai dari 12 dan berakhir pada iterasi ke-1 kemudian di gabung dengan L dan R.

   - Iterasi ke-12 Menggunakan Fungsi *CAST* tipe 3

     Kurangkankan Kunci *Masking* ke-12 dengan L
     I = keym[12] - L
     keym[12]=00100111110001101110101110111110
     L=11010110000110101111101010011001-
     I=01010001101010111111000100100101

     I=I geser keyr[12]
     I=10010101010001101010111111000100

     F=((s1[Ia] + s2[Ib])XOR s3[Ic])- s3[Id]
     s1[Ia]=11010101101111011001111010011000
     s2[Ib]=10111001010010011110001101010100+
     F=10101111100111100100000010111111
     s3[Ic]=10110011010001111100110010010110 □
     F=10000011101000011100110001111010
     s4[Id]=01000111010001001110101011010100 -

F=01011010000100010100100001100101

R= R XOR F
R=11111111111100101111010001101100
F=01011010000100010100100001100101 □
R=10100101111000111011110000001001

- Iterasi ke-11 Menggunakan Fungsi *CAST* tipe 2

XOR kan R dengan Kunci *Masking* ke-11
I = keym[11] XOR R
keym[11]=11111011100010001101100100010101
R=10100101111000111011110000001001 □
I=01011110011010110110010100011100

I=I geser keyr[11]
I=01000111000101111001101011011001

F=((s1[Ia] - s2[Ib])+ s3[Ic]) XOR s3[Id]
s1[Ia]=10110000010001100110100111111110
s2[Ib]=10100010110100101011110100101101 -
F=11001111100000101001110000000011
s3[Ic]=11011000011100010000111101101001 +
F=10011110001011111010000011110111
s4[Id]=01010110010010001111011100100101 □
F=01111011000011000001111010000000

L= L XOR F
L=11010110000110101111101010011001
F=01111011000011000001111010000000 □
L=10101101000101101110010000011001

- Iterasi ke-10 Menggunakan Fungsi *CAST* tipe 1

Tambahkan L dengan Kunci *Masking* ke-10
I = keym[10] + L
keym[10]=00011001001100111101100100000011
L=10101101000101101110010000011001+
I=11000110010010101011110100011100

I=I geser keyr[10]
I=10101111010001110011000110010010
F=((s1[Ia] XOR s2[Ib])- s3[Ic])- s3[Id]
s1[Ia]=10110011010001111100110010010110
s2[Ib]=10110000010001100110100111111110
F=11111101100000001111011010100010
s3[Ic]=01010000101110110110010010100010 -
F=01000111101001010011000001010111
s4[Id]=10011110001000000100110100000010 +
F=00110111011010000111110101110111

R= R XOR F
R=10100101111000111011110000001001
F=00110111011010000111110101110111
R=10010010100010111100000101111110
.....
.....
.....
.....

Hasil Keluaran iterasi ke-1
L=01001011010011110100110101010000
R=01010101010101000100010101010010

Gabungkan L dan R
01001011010011110100110101010000010101010101010001000010101010010

Konversi ke *heksadesimal* dan *string*
Heksadesimal = 4b4f4d5055544552
String = KOMPUTER

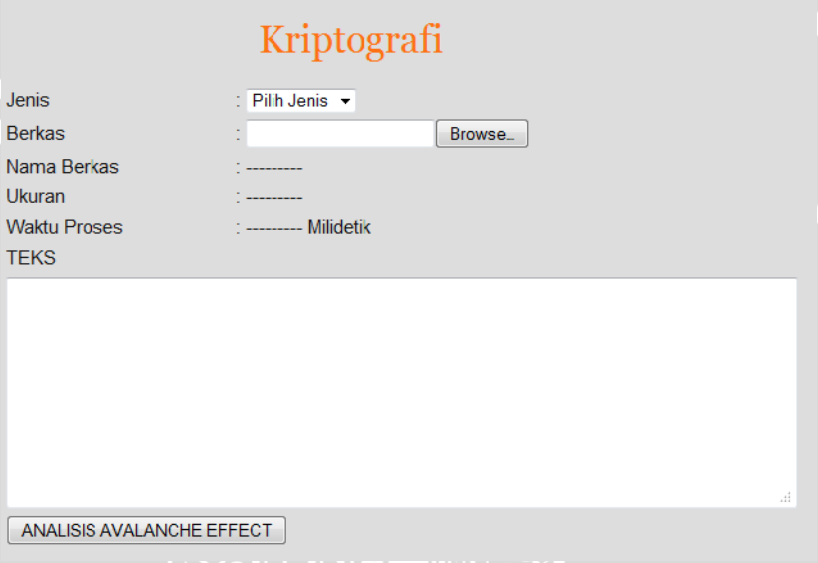
3.4 Perancangan *Interface*

Rancangan *interface* untuk aplikasi ini akan memiliki beberapa menu yaitu :

1. Jenis : Untuk memilih jenis kriptografi. Enkripsi atau dekripsi.
2. Berkas : Untuk upload file **.txt*.

3. Nama Berkas : Menampilkan nama berkas yang diupload.
4. Ukuran : Menampilkan ukuran berkas yang diupload.
5. Waktu Proses : Menampilkan lama proses enkripsi atau dekripsi
6. Teks : Berisi teks hasil enkripsi atau dekripsi
7. Analisis Avalanche Effect : Link menuju halaman analisi *Avalanche Effect*.

Rancangan *interface* untuk aplikasi enkripsi dan dekripsi *CAST-128* dapat dilihat pada gambar 3.11.



Gambar 3.12 Rancangan interface

3.5 Perancangan Analisis Waktu proses dan *Avalanche Effect*

Pada aplikasi enkripsi dan dekripsi dengan algoritma *CAST-128* ini dapat menerima masukan berupa *file* teks. Parameter yang digunakan untuk melakukan uji coba pada *file* teks tersebut diantaranya adalah waktu proses dan *avalanche effect.*

Pada analisis waktu proses akan dihitung waktu proses enkripsi dan dekripsi pada beberapa *file* dengan ukuran berbeda. *File-file* yang akan diuji diantaranya berukuran 50Kb, 100Kb, 150Kb 200Kb, 250Kb, 300Kb, 350Kb, 400Kb, 450Kb dan 500Kb. Tujuan dari uji coba waktu proses ini adalah untuk mengetahui apakah nanti

waktu proses akan berbanding lurus dengan ukuran *file* atau tidak. Bentuk tabel dari hasil uji waktu proses ini dapat dilihat pada tabel 3.3 dan tabel 3.4.

Pengujian selanjutnya adalah pengujian *avalanche effect*. Pada pengujian ini akan digunakan lima *file* dengan ukuran yang berbeda mulai ukuran 8 *bytes* hingga 40 *bytes*. Kemudian beberapa perubahan yang dilakukan dengan uji coba dengan parameter *avalance effect* adalah :

1. Perubahan bit/*byte* serta posisi pada *plaintext* terhadap *chipertext* dengan *key* enkripsi yang sama.
2. Perubahan bit/*byte* serta posisi pada *key* enkripsi terhadap *chipertext* dengan *plaintext* yang sama.
3. Perubahan bit/*byte* serta posisi pada *chipertext* terhadap *plaintext* dengan *key* enkripsi yang sama.

Tujuan dari uji coba menggunakan *avalanche effect* ini adalah untuk mengetahui ketahanan algoritma kriptografi *CAST-128*. Bentuk tabel hasil analisis *avalanche effect* ini dapat dilihat pada tabel 3.5.

Tabel 3.3 Rancangan tabel hasil uji untuk parameter waktu proses enkripsi.

Nama *File*	Ukuran *File*	Waktu Proses Enkripsi

Keterangan tabel 3.3 :

- Kolom Nama *File* berisikan nama *file* yang akan diuji
- Kolom Ukuran *File* berisikan ukuran *file* yang akan di uji (dalam *byte*)
- Waktu Proses Enkripsi berisikan waktu yang digunakan dalam mengenkripsi sebuah *file* (dalam milidetik)

Tabel 3.4 Rancangan tabel hasil uji untuk parameter waktu proses dekripsi.

Nama *File*	Ukuran *File*	Waktu Proses Dekripsi

Keterangan tabel 3.4 :
- Kolom Nama *File* berisikan nama *file* yang akan diuji
- Kolom Ukuran *File* berisikan ukuran *file* yang akan di uji (dalam *byte*)
- Waktu Proses Dekripsi berisikan waktu yang digunakan dalam mendekripsi sebuah *file* (dalam milidetik)

Tabel 3.5 Rancangan tabel uji untuk parameter *avalanche effect*

Nama *File*	Ukuran *File*	Jumlah Perubahan bit/byte	Posisi Perubahan	*Avalanche Effect* (%)

Keterangan tabel 3.3 :
- Kolom Nama *File* berisikan nama *file* yang akan diuji
- Kolom Ukuran *File* berisikan ukuran *file* yang akan di uji (dalam byte).
- Kolom Jumlah Perubahan bit/*byte* berisikan jumlah bit atau *byte* yang diubah.
- Kolom Posisi Perubahan berisikan posisi bit/*byte* yang dirubah misal di awal, tengah atau akhir.
- Kolom *Avalanche Effect* berisikan nilai dari *avalanche effectfile* yang diuji.

BAB IV
IMPLEMENTASI DAN PEMBAHASAN

Pada bab ini akan dilakukan implementasi algoritma *CAST-128* dan analisis pada perangkat lunak tersebut.

4.1 Lingkungan Implementasi

Pada lingkungan implementasi ada dua faktor yang mempengaruhi yaitu lingkungan perangkat keras dan lingkungan perangkat lunak.

4.1.2 Lingkungan Perangkat Keras

Perangkat keras yang digunakan untuk implementasi Algoritma *CAST-128* memiliki spesifikasi sebagai berikut :

1. *Processor* intel *Core i3* @2,27GHz.
2. *Memory*3 GB.
3. *Harddisk* 320 GB.

4.1.3 Lingkungan Perangkat Lunak

Perangkat lunak yang digunakan untuk implementasi Algoritma *CAST-128* adalah sebagai berikut :

1. Sistem Operasi Microsoft Windows 7 Ultimate.
2. *Tools programing* : PHP.

4.2 Implementasi Program

Berdasarkan perancangan pada subbab 3.2 maka pada bab ini akan dibahas mengenai implementasi dari perancangan tersebut.

4.2.1 Proses *Input File*

Pada perangkat lunak ini sebuah *file* **.txt* sebagai *inputan* dari *user* akan dibaca perangkat lunak per*byte* dan disimpan ke variabel *source* dan kemudian *byte-byte* tersebut akan dikonversi kembali ke *string* untuk selanjutnya dilakukan proses enkripsi.

Untuk melakukan *openfile* ini digunakan `move_uploaded_file` dan menggunakan fungsi `file_get_contents()`. Dimana pada proses open ini akan didapatkan beberapa atribut berkas yang diunggah seperti tipe, ukuran, nama dan lokasi asal berkas. Fungsi

`move_uploaded_file` untuk memindahakan atau mengunggah berkas pada server. Dan fungsi `file_get_contents` untuk mendapatkan atau membaca isi file yang diunggah. Ketiga fungsi tersebut merupakan fungsi yag sudah tersedia pada PHP. Untuk lebih jelasnya proses *input file* dapat dilihat pada *sourcecode* 4.1.

```
$tipe_file   = $_FILES['berkas']['type'];
$lokasi_file = $_FILES['berkas']['tmp_name'];
$nama_file   = $_FILES['berkas']['name'];
$ukuran_file = $_FILES['berkas']['size'];

if ($tipe_file == "text/txt")
{
$direktori ="berkas/$nama_file";
move_uploaded_file($lokasi_file,"$direktori");
$asli=file_get_contents($direktori);
$panjang=strlen($asli);
$hitung=0;
$aa=0;
$bb=80;
while ($aa<=$panjang)
{
$kata[$hitung] = substr($asli,$aa,$bb);
$aa=$aa+$bb;
$hitung++;
}
if($panjang%$bb!=0)
{
$kata[$hitung] = substr($asli,$aa);
$hitung++;
}
```

Sourcecode 4.1 Proses `Open`

4.2.2 Proses Penghitungan *Subkey*

Pada proses penghitungan *subkey* proses pertama yang dilakukan adalah mengambil *stringkey* masukan dari *user* yang kemudian di*padding* dan dikonversi ke dalam *byte* dan disimpan dalam variabel `kunci_masking` dan `kunci_rotasi` yang bertipe *byte array*. Hal tersebut terdapat pada *Sourcecode 4.2*.

```
$this->r = strlen($kunci) < 11? 12 : 16;
    $kunci = $this->formatkunci($kunci);
    $this->penjadwalan_kunci($kunci);
```

Sourcecode 4.2Proses Perhitungan Subkey

Pada baris pertama *Sourcecode 4.2* dihitung panjang karakter kunci yang di*input*kan untuk mengetahui panjang rotasi yang akan digunakan dalam proses enkripsi/deskipsi. Dan fungsi `formatkunci`

untuk memecah kunci ke dalam array dengan bentuk nilai ASCII dari masing-masing karakter dalam kunci. Untuk lebih jelasnya fungsi `formatkunci` dapat dilihat pada *Sourcecode 4.3*.

```
function formatkunci($data)
	{
		$return = array_values(unpack("C*",$data));
		for($i = count($return); $i < 16; $i++)
			$return[$i] = 0;
		return $return;
	}
```

Sourcecode 4.3 Fungsi `formatkunci`.

Fungsi `penjadwalan_kunci` berfungsi untuk menghitung ke-16 kunci masking dan ke-16 fungsi rotasi. Untuk lebih jelasnya fungsi `penjadwalan_kunci` dapat dilihat pada *Sourcecode 4.4*.

```
function penjadwalan_kunci($kunci)
{
$x0x1x2x3 = $this->buat32($kunci,0x0);
$x4x5x6x7 = $this->buat32($kunci,0x4);
$x8x9xAxB = $this->buat32($kunci,0x8);
$xCxDxExF = $this->buat32($kunci,0xC);

$b = $this->pecah32($x0x1x2x3); $x0 = $b[0]; $x1 = $b[1]; $x2 =
  $b[2]; $x3 = $b[3];
$b = $this->pecah32($x4x5x6x7); $x4 = $b[0]; $x5 = $b[1]; $x6 =
  $b[2]; $x7 = $b[3];
$b = $this->pecah32($x8x9xAxB); $x8 = $b[0]; $x9 = $b[1]; $xA =
  $b[2]; $xB = $b[3];
$b = $this->pecah32($xCxDxExF); $xC = $b[0]; $xD = $b[1]; $xE =
  $b[2]; $xF = $b[3];

$z0z1z2z3  =  $x0x1x2x3  ^  $this->s5[$xD]  ^  $this->s6[$xF]  ^
  $this->s7[$xC] ^ $this->s8[$xE] ^ $this->s7[$x8];
$b = $this->pecah32($z0z1z2z3); $z0 = $b[0]; $z1 = $b[1]; $z2 =
  $b[2]; $z3 = $b[3];
$z4z5z6z7  =  $x8x9xAxB  ^  $this->s5[$z0]  ^  $this->s6[$z2]  ^
  $this->s7[$z1] ^ $this->s8[$z3] ^ $this->s8[$xA];
$b = $this->pecah32($z4z5z6z7); $z4 = $b[0]; $z5 = $b[1]; $z6 =
  $b[2]; $z7 = $b[3];
$z8z9zAzB  =  $xCxDxExF  ^  $this->s5[$z7]  ^  $this->s6[$z6]  ^
  $this->s7[$z5] ^ $this->s8[$z4] ^ $this->s5[$x9];
$b = $this->pecah32($z8z9zAzB); $z8 = $b[0]; $z9 = $b[1]; $zA =
  $b[2]; $zB = $b[3];
$zCzDzEzF  =  $x4x5x6x7  ^  $this->s5[$zA]  ^  $this->s6[$z9]  ^
  $this->s7[$zB] ^ $this->s8[$z8] ^ $this->s6[$xB];
$b = $this->pecah32($zCzDzEzF); $zC = $b[0]; $zD = $b[1]; $zE =
  $b[2]; $zF = $b[3];

$this->kunci_masking[1]  =  $this->s5[$z8]  ^  $this->s6[$z9]  ^
  $this->s7[$z7] ^ $this->s8[$z6] ^ $this->s5[$z2];
```

```
$this->kunci_masking[2]  =  $this->s5[$zA]  ^  $this->s6[$zB]  ^
  $this->s7[$z5] ^ $this->s8[$z4] ^ $this->s6[$z6];
$this->kunci_masking[3]  =  $this->s5[$zC]  ^  $this->s6[$zD]  ^
  $this->s7[$z3] ^ $this->s8[$z2] ^ $this->s7[$z9];
$this->kunci_masking[4]  =  $this->s5[$zE]  ^  $this->s6[$zF]  ^
  $this->s7[$z1] ^ $this->s8[$z0] ^ $this->s8[$zC];

$x0x1x2x3  =  $z8z9zAzB  ^  $this->s5[$z5]  ^  $this->s6[$z7]  ^
  $this->s7[$z4] ^ $this->s8[$z6] ^ $this->s7[$z0];
$b = $this->pecah32($x0x1x2x3); $x0 = $b[0]; $x1 = $b[1]; $x2 =
  $b[2]; $x3 = $b[3];
$x4x5x6x7  =  $z0z1z2z3  ^  $this->s5[$x0]  ^  $this->s6[$x2]  ^
  $this->s7[$x1] ^ $this->s8[$x3] ^ $this->s8[$z2];
$b = $this->pecah32($x4x5x6x7); $x4 = $b[0]; $x5 = $b[1]; $x6 =
  $b[2]; $x7 = $b[3];
$x8x9xAxB  =  $z4z5z6z7  ^  $this->s5[$x7]  ^  $this->s6[$x6]  ^
  $this->s7[$x5] ^ $this->s8[$x4] ^ $this->s5[$z1];
$b = $this->pecah32($x8x9xAxB); $x8 = $b[0]; $x9 = $b[1]; $xA =
  $b[2]; $xB = $b[3];
$xCxDxExF  =  $zCzDzEzF  ^  $this->s5[$xA]  ^  $this->s6[$x9]  ^
  $this->s7[$xB] ^ $this->s8[$x8] ^ $this->s6[$z3];
$b = $this->pecah32($xCxDxExF); $xC = $b[0]; $xD = $b[1]; $xE =
  $b[2]; $xF = $b[3];

$this->kunci_masking[5]   =  $this->s5[$x3]  ^  $this->s6[$x2]  ^
  $this->s7[$xC] ^ $this->s8[$xD] ^ $this->s5[$x8];
$this->kunci_masking[6]   =  $this->s5[$x1]  ^  $this->s6[$x0]  ^
  $this->s7[$xE] ^ $this->s8[$xF] ^ $this->s6[$xD];
$this->kunci_masking[7]   =  $this->s5[$x7]  ^  $this->s6[$x6]  ^
  $this->s7[$x8] ^ $this->s8[$x9] ^ $this->s7[$x3];
$this->kunci_masking[8]   =  $this->s5[$x5]  ^  $this->s6[$x4]  ^
  $this->s7[$xA] ^ $this->s8[$xB] ^ $this->s8[$x7];

$z0z1z2z3  =  $x0x1x2x3  ^  $this->s5[$xD]  ^  $this->s6[$xF]  ^
  $this->s7[$xC] ^ $this->s8[$xE] ^ $this->s7[$x8];
$b = $this->pecah32($z0z1z2z3); $z0 = $b[0]; $z1 = $b[1]; $z2 =
  $b[2]; $z3 = $b[3];
$z4z5z6z7  =  $x8x9xAxB  ^  $this->s5[$z0]  ^  $this->s6[$z2]  ^
  $this->s7[$z1] ^ $this->s8[$z3] ^ $this->s8[$xA];
      $b  =  $this->pecah32($z4z5z6z7);  $z4  =  $b[0];  $z5  =
  $b[1]; $z6 = $b[2]; $z7 = $b[3];
$z8z9zAzB  =  $xCxDxExF  ^  $this->s5[$z7]  ^  $this->s6[$z6]  ^
  $this->s7[$z5] ^ $this->s8[$z4] ^ $this->s5[$x9];
$b = $this->pecah32($z8z9zAzB); $z8 = $b[0]; $z9 = $b[1]; $zA =
  $b[2]; $zB = $b[3];
$zCzDzEzF  =  $x4x5x6x7  ^  $this->s5[$zA]  ^  $this->s6[$z9]  ^
  $this->s7[$zB] ^ $this->s8[$z8] ^ $this->s6[$xB];
$b = $this->pecah32($zCzDzEzF); $zC = $b[0]; $zD = $b[1]; $zE =
  $b[2]; $zF = $b[3];

$this->kunci_masking[9]   =  $this->s5[$z3]  ^  $this->s6[$z2]  ^
  $this->s7[$zC] ^ $this->s8[$zD] ^ $this->s5[$z9];
$this->kunci_masking[10]  =  $this->s5[$z1]  ^  $this->s6[$z0]  ^
  $this->s7[$zE] ^ $this->s8[$zF] ^ $this->s6[$zC];
$this->kunci_masking[11]  =  $this->s5[$z7]  ^  $this->s6[$z6]  ^
  $this->s7[$z8] ^ $this->s8[$z9] ^ $this->s7[$z2];
```

```
$this->kunci_masking[12] = $this->s5[$z5] ^ $this->s6[$z4] ^
 $this->s7[$zA] ^ $this->s8[$zB] ^ $this->s8[$z6];

$x0x1x2x3 = $z8z9zAzB ^ $this->s5[$z5] ^ $this->s6[$z7] ^
 $this->s7[$z4] ^ $this->s8[$z6] ^ $this->s7[$z0];
$b = $this->pecah32($x0x1x2x3); $x0 = $b[0]; $x1 = $b[1]; $x2 =
 $b[2]; $x3 = $b[3];
$x4x5x6x7 = $z0z1z2z3 ^ $this->s5[$x0] ^ $this->s6[$x2] ^
 $this->s7[$x1] ^ $this->s8[$x3] ^ $this->s8[$z2];
$b = $this->pecah32($x4x5x6x7); $x4 = $b[0]; $x5 = $b[1]; $x6 =
 $b[2]; $x7 = $b[3];
$x8x9xAxB = $z4z5z6z7 ^ $this->s5[$x7] ^ $this->s6[$x6] ^
 $this->s7[$x5] ^ $this->s8[$x4] ^ $this->s5[$z1];
$b = $this->pecah32($x8x9xAxB); $x8 = $b[0]; $x9 = $b[1]; $xA =
 $b[2]; $xB = $b[3];
$xCxDxExF = $zCzDzEzF ^ $this->s5[$xA] ^ $this->s6[$x9] ^
 $this->s7[$xB] ^ $this->s8[$x8] ^ $this->s6[$z3];
$b = $this->pecah32($xCxDxExF); $xC = $b[0]; $xD = $b[1]; $xE =
 $b[2]; $xF = $b[3];

$this->kunci_masking[13] = $this->s5[$x8] ^ $this->s6[$x9] ^
 $this->s7[$x7] ^ $this->s8[$x6] ^ $this->s5[$x3];
$this->kunci_masking[14] = $this->s5[$xA] ^ $this->s6[$xB] ^
 $this->s7[$x5] ^ $this->s8[$x4] ^ $this->s6[$x7];
$this->kunci_masking[15] = $this->s5[$xC] ^ $this->s6[$xD] ^
 $this->s7[$x3] ^ $this->s8[$x2] ^ $this->s7[$x8];
$this->kunci_masking[16] = $this->s5[$xE] ^ $this->s6[$xF] ^
 $this->s7[$x1] ^ $this->s8[$x0] ^ $this->s8[$xD];

$z0z1z2z3 = $x0x1x2x3 ^ $this->s5[$xD] ^ $this->s6[$xF] ^
 $this->s7[$xC] ^ $this->s8[$xE] ^ $this->s7[$x8];
$b = $this->pecah32($z0z1z2z3); $z0 = $b[0]; $z1 = $b[1]; $z2 =
 $b[2]; $z3 = $b[3];
$z4z5z6z7 = $x8x9xAxB ^ $this->s5[$z0] ^ $this->s6[$z2] ^
 $this->s7[$z1] ^ $this->s8[$z3] ^ $this->s8[$xA];
$b = $this->pecah32($z4z5z6z7); $z4 = $b[0]; $z5 = $b[1]; $z6 =
 $b[2]; $z7 = $b[3];
$z8z9zAzB = $xCxDxExF ^ $this->s5[$z7] ^ $this->s6[$z6] ^
 $this->s7[$z5] ^ $this->s8[$z4] ^ $this->s5[$x9];
$b = $this->pecah32($z8z9zAzB); $z8 = $b[0]; $z9 = $b[1]; $zA =
 $b[2]; $zB = $b[3];
$zCzDzEzF = $x4x5x6x7 ^ $this->s5[$zA] ^ $this->s6[$z9] ^
 $this->s7[$zB] ^ $this->s8[$z8] ^ $this->s6[$xB];
$b = $this->pecah32($zCzDzEzF); $zC = $b[0]; $zD = $b[1]; $zE =
 $b[2]; $zF = $b[3];

$this->kunci_rotasi[1] = 0x1F & ($this->s5[$z8] ^ $this-
 >s6[$z9] ^ $this->s7[$z7] ^ $this->s8[$z6] ^ $this->s5[$z2]);
$this->kunci_rotasi[2] = 0x1F & ($this->s5[$zA] ^ $this-
 >s6[$zB] ^ $this->s7[$z5] ^ $this->s8[$z4] ^ $this->s6[$z6]);
$this->kunci_rotasi[3] = 0x1F & ($this->s5[$zC] ^ $this-
 >s6[$zD] ^ $this->s7[$z3] ^ $this->s8[$z2] ^ $this->s7[$z9]);
$this->kunci_rotasi[4] = 0x1F & ($this->s5[$zE] ^ $this-
 >s6[$zF] ^ $this->s7[$z1] ^ $this->s8[$z0] ^ $this->s8[$zC]);

$x0x1x2x3 = $z8z9zAzB ^ $this->s5[$z5] ^ $this->s6[$z7] ^
```

```
  $this->s7[$z4] ^ $this->s8[$z6] ^ $this->s7[$z0];
$b = $this->pecah32($x0x1x2x3); $x0 = $b[0]; $x1 = $b[1]; $x2 =
  $b[2]; $x3 = $b[3];
$x4x5x6x7  =  $z0z1z2z3  ^  $this->s5[$x0]  ^  $this->s6[$x2]  ^
  $this->s7[$x1] ^ $this->s8[$x3] ^ $this->s8[$z2];
$b = $this->pecah32($x4x5x6x7); $x4 = $b[0]; $x5 = $b[1]; $x6 =
  $b[2]; $x7 = $b[3];
$x8x9xAxB  =  $z4z5z6z7  ^  $this->s5[$x7]  ^  $this->s6[$x6]  ^
  $this->s7[$x5] ^ $this->s8[$x4] ^ $this->s5[$z1];
$b = $this->pecah32($x8x9xAxB); $x8 = $b[0]; $x9 = $b[1]; $xA =
  $b[2]; $xB = $b[3];
$xCxDxExF  =  $zCzDzEzF  ^  $this->s5[$xA]  ^  $this->s6[$x9]  ^
  $this->s7[$xB] ^ $this->s8[$x8] ^ $this->s6[$z3];
$b = $this->pecah32($xCxDxExF); $xC = $b[0]; $xD = $b[1]; $xE =
  $b[2]; $xF = $b[3];

$this->kunci_rotasi[5]    =  0x1F  &  ($this->s5[$x3]  ^  $this-
  >s6[$x2] ^ $this->s7[$xC] ^ $this->s8[$xD] ^ $this->s5[$x8]);
$this->kunci_rotasi[6]    =  0x1F  &  ($this->s5[$x1]  ^  $this-
  >s6[$x0] ^ $this->s7[$xE] ^ $this->s8[$xF] ^ $this->s6[$xD]);
$this->kunci_rotasi[7]    =  0x1F  &  ($this->s5[$x7]  ^  $this-
  >s6[$x6] ^ $this->s7[$x8] ^ $this->s8[$x9] ^ $this->s7[$x3]);
$this->kunci_rotasi[8]    =  0x1F  &  ($this->s5[$x5]  ^  $this-
  >s6[$x4] ^ $this->s7[$xA] ^ $this->s8[$xB] ^ $this->s8[$x7]);

$z0z1z2z3  =  $x0x1x2x3  ^  $this->s5[$xD]  ^  $this->s6[$xF]  ^
  $this->s7[$xC] ^ $this->s8[$xE] ^ $this->s7[$x8];
$b = $this->pecah32($z0z1z2z3); $z0 = $b[0]; $z1 = $b[1]; $z2 =
  $b[2]; $z3 = $b[3];
$z4z5z6z7  =  $x8x9xAxB  ^  $this->s5[$z0]  ^  $this->s6[$z2]  ^
  $this->s7[$z1] ^ $this->s8[$z3] ^ $this->s8[$xA];
$b = $this->pecah32($z4z5z6z7); $z4 = $b[0]; $z5 = $b[1]; $z6 =
  $b[2]; $z7 = $b[3];
$z8z9zAzB  =  $xCxDxExF  ^  $this->s5[$z7]  ^  $this->s6[$z6]  ^
  $this->s7[$z5] ^ $this->s8[$z4] ^ $this->s5[$x9];
$b = $this->pecah32($z8z9zAzB); $z8 = $b[0]; $z9 = $b[1]; $zA =
  $b[2]; $zB = $b[3];
$zCzDzEzF  =  $x4x5x6x7  ^  $this->s5[$zA]  ^  $this->s6[$z9]  ^
  $this->s7[$zB] ^ $this->s8[$z8] ^ $this->s6[$xB];
$b = $this->pecah32($zCzDzEzF); $zC = $b[0]; $zD = $b[1]; $zE =
  $b[2]; $zF = $b[3];

$this->kunci_rotasi[9]    =  0x1F  &  ($this->s5[$z3]  ^  $this-
  >s6[$z2] ^ $this->s7[$zC] ^ $this->s8[$zD] ^ $this->s5[$z9]);
$this->kunci_rotasi[10]   =  0x1F  &  ($this->s5[$z1]  ^  $this-
  >s6[$z0] ^ $this->s7[$zE] ^ $this->s8[$zF] ^ $this->s6[$zC]);
        $this->kunci_rotasi[11]   =  0x1F  &  ($this->s5[$z7]  ^
  $this->s6[$z6]  ^  $this->s7[$z8]  ^  $this->s8[$z9]  ^  $this-
  >s7[$z2]);
        $this->kunci_rotasi[12]   =  0x1F  &  ($this->s5[$z5]  ^
  $this->s6[$z4]  ^  $this->s7[$zA]  ^  $this->s8[$zB]  ^  $this-
  >s8[$z6]);

$x0x1x2x3  =  $z8z9zAzB  ^  $this->s5[$z5]  ^  $this->s6[$z7]  ^
  $this->s7[$z4] ^ $this->s8[$z6] ^ $this->s7[$z0];
$b = $this->pecah32($x0x1x2x3); $x0 = $b[0]; $x1 = $b[1]; $x2 =
```

```
  $b[2]; $x3 = $b[3];
$x4x5x6x7  =  $z0z1z2z3  ^  $this->s5[$x0]  ^  $this->s6[$x2]  ^
  $this->s7[$x1] ^ $this->s8[$x3] ^ $this->s8[$z2];
$b = $this->pecah32($x4x5x6x7); $x4 = $b[0]; $x5 = $b[1]; $x6 =
  $b[2]; $x7 = $b[3];
$x8x9xAxB  =  $z4z5z6z7  ^  $this->s5[$x7]  ^  $this->s6[$x6]  ^
  $this->s7[$x5] ^ $this->s8[$x4] ^ $this->s5[$z1];
$b = $this->pecah32($x8x9xAxB); $x8 = $b[0]; $x9 = $b[1]; $xA =
  $b[2]; $xB = $b[3];
$xCxDxExF  =  $zCzDzEzF  ^  $this->s5[$xA]  ^  $this->s6[$x9]  ^
  $this->s7[$xB] ^ $this->s8[$x8] ^ $this->s6[$z3];
$b = $this->pecah32($xCxDxExF); $xC = $b[0]; $xD = $b[1]; $xE =
  $b[2]; $xF = $b[3];

$this->kunci_rotasi[13]  =  0x1F  &  ($this->s5[$x8]  ^  $this-
  >s6[$x9] ^ $this->s7[$x7] ^ $this->s8[$x6] ^ $this->s5[$x3]);
$this->kunci_rotasi[14]  =  0x1F  &  ($this->s5[$xA]  ^  $this-
  >s6[$xB] ^ $this->s7[$x5] ^ $this->s8[$x4] ^ $this->s6[$x7]);
$this->kunci_rotasi[15]  =  0x1F  &  ($this->s5[$xC]  ^  $this-
  >s6[$xD] ^ $this->s7[$x3] ^ $this->s8[$x2] ^ $this->s7[$x8]);
$this->kunci_rotasi[16]  =  0x1F  &  ($this->s5[$xE]  ^  $this-
  >s6[$xF] ^ $this->s7[$x1] ^ $this->s8[$x0] ^ $this->s8[$xD]);
}
```

Sourcecode 4.4 fungsi penjadwalan_kunci

Fungsi `penjadwalan_kunci` pada *Sourcecode 4.4* sesuai dengan algroritma pembangkitan kunci pada subbab 2.5.3.

4.2.3 Proses Enkripsi

Pada saat *file* dibuka isi dari variabel `source` yang berupa *byte array* dikonversi ke dalam *string* dan disimpan dalam variabel `$asli` agar bisa dituliskan ke `textarea`. Kemudian saat akan melakukan enkripsi ini isi dari variabel `$asli` akan dikonversi kembali ke dalam bentuk *byte array* untuk dienkripsi. Hal tersebut tedapat pada Pada *Sourcecode 4.5*.

```
$a=0;
while($a<$hitung)
{
$hasil[$a]=$coba->enkripsi($kata[$a], $_POST[kunci]);
$a++;
}
```

Sourcecode 4.5 Proses Enkripsi

Fungsi `enkripsi pada` *Sourcecode 4.5* memanggil fungsi `enkripsi` pada *sourcecode 4.6*. Dimana pada fungsi ini data yang akan dienkripsi akan dipecah pecah kedalam blok-blok dengan ukuran 8 *byte*. Dan akan di*padding* jika panjang kurang dari 8 *byte*. Kemudian dilakukan enkripsi perblok.

```
function enkripsi($data,$kunci=null)
{
$return = '';
$this->r = strlen($kunci) < 11? 12 : 16;
$kunci = $this->formatkunci($kunci);
$this->penjadwalan_kunci($kunci);

$prevcipher = $this->iv;

for($i = 0; $i < strlen($data); $i += $this->ukuranblok)
{
$blok = substr($data,$i,$this->ukuranblok);

if(strlen($blok) < $this->ukuranblok)
       $blok = str_pad($blok,8,"\0");

$blok = $blok ^ $prevcipher;
$prevcipher = $this->enkripsi_blok($blok);
$return .= $prevcipher;
}

return $return;
}
```

Sourcecode 4.6 Fungsi *enkripsi*

Fungsi enkripsi_perblok pada *Sourcecode 4.9* dilakukan rotasi jaringan *feistel* sebanyak 12 atau 16. Sesuai dengan perhitungan jumlah rotasi pada *Sourcecode 4.2*.

```
function enkripsi_blok($data)
{
  $L = $this->formatdata(substr($data,0,4));
  $R = $this->formatdata(substr($data,4,4));

  $L ^= $this->f0($R,$this->kunci_masking[1],$this-
    >kunci_rotasi[1]);
  $R ^= $this->f1($L,$this->kunci_masking[2],$this-
    >kunci_rotasi[2]);
  $L ^= $this->f2($R,$this->kunci_masking[3],$this-
    >kunci_rotasi[3]);
  $R ^= $this->f0($L,$this->kunci_masking[4],$this-
    >kunci_rotasi[4]);
  $L ^= $this->f1($R,$this->kunci_masking[5],$this-
    >kunci_rotasi[5]);
  $R ^= $this->f2($L,$this->kunci_masking[6],$this-
    >kunci_rotasi[6]);
  $L ^= $this->f0($R,$this->kunci_masking[7],$this-
    >kunci_rotasi[7]);
  $R ^= $this->f1($L,$this->kunci_masking[8],$this-
    >kunci_rotasi[8]);
  $L ^= $this->f2($R,$this->kunci_masking[9],$this-
    >kunci_rotasi[9]);
  $R ^= $this->f0($L,$this->kunci_masking[10],$this-
    >kunci_rotasi[10]);
```

```
  $L ^= $this->f1($R,$this->kunci_masking[11],$this-
    >kunci_rotasi[11]);
  $R ^= $this->f2($L,$this->kunci_masking[12],$this-
    >kunci_rotasi[12]);
  if ($this->r > 12)
  {
    $L ^= $this->f0($R,$this->kunci_masking[13],$this-
      >kunci_rotasi[13]);
    $R ^= $this->f1($L,$this->kunci_masking[14],$this-
      >kunci_rotasi[14]);
    $L ^= $this->f2($R,$this->kunci_masking[15],$this-
      >kunci_rotasi[15]);
    $R ^= $this->f0($L,$this->kunci_masking[16],$this-
      >kunci_rotasi[16]);
  }

  $encrypted = pack("N",$R) . pack("N",$L);

  return $encrypted;
}
```

*Sourcecode 4.7*Fungsi *Enkripsi Perblok*

Fungsi `enkripsi_blok` ini terdapat tiga jenis fungsi *CAST* yang digunakan dalam proses enkripsi. Ketiga fungsi tersebut digunakan secara bergantian dengan fungsi f0 terlebih dahulu diikuti f1 dan f2 kemudian f0 kembali dan begitu seterusnya. Untuk lebih jelasnya tentang ketiga fugsi tersebut dapat dilihat pada *Sourcecode 4.8*.

```
function f0($data,$kunci_masking,$kunci_rotasi)
{
$I = $kunci_masking + $data;
$bin = str_pad(decbin($I),32,'0',STR_PAD_LEFT);
$I = bindec(substr($bin,$kunci_rotasi)
      .substr($bin,0,$kunci_rotasi));
$f = $this->s1[($I >> 24) & 0xFF] ^ $this->s2[($I >> 16) &
      0xFF];
$f = $f - $this->s3[($I >> 8) & 0xFF];
$f = $f + $this->s4[$I & 0xFF];
return $f;
}

function f1($data,$kunci_masking,$kunci_rotasi)
{
$I = $kunci_masking ^ $data;
$bin = str_pad(decbin($I),32,'0',STR_PAD_LEFT);
$I = bindec(substr($bin,$kunci_rotasi) .
      substr($bin,0,$kunci_rotasi));
$f = $this->s1[($I >> 24) & 0xFF] - $this->s2[($I >> 16) &
      0xFF];
$f = $f + $this->s3[($I >> 8) & 0xFF];
$f = $f ^ $this->s4[$I & 0xFF];
return $f;
}
```

```
function f2($data,$kunci_masking,$kunci_rotasi)
{
$I = 0xFFFFFFFF & ($kunci_masking - $data);
$bin = str_pad(decbin($I),32,'0',STR_PAD_LEFT);
$I = bindec(substr($bin,$kunci_rotasi) .
      substr($bin,0,$kunci_rotasi));
$f = ($this->s1[($I >> 24) & 0xFF] + $this->s2[($I >> 16) &
      0xFF]);
$f = $f ^ $this->s3[($I >> 8) & 0xFF];
$f = 0xFFFFFFFF & ($f - $this->s4[$I & 0xFF]);
return $f;
}
```

Sourcecode 4.8 Fungsi-fungsi *CAST*

4.2.4 Proses Dekripsi

Pada proses dekripsi sebenarnya tahap-tahapnya hampir sama dengan proses enkripsi perbedaannya hanya pada fungsi `dekripsi_blok`nya saja. Dimana putaran untuk *chipertextnya* merupakan pembalikan dari putaran fungsi `enkripsi_blok`. Fungsi `DekripsiPerBlok` dapat dilihat pada *Sourcecode 4.9.*

```
function dekripsi_blok($data)
{
$R = $this->formatdata(substr($data,0,4));
$L = $this->formatdata(substr($data,4,4));

if ($this->r > 12)
{
  $R ^= $this->f0($L,$this->kunci_masking[16],$this-
        >kunci_rotasi[16]);
  $L ^= $this->f2($R,$this->kunci_masking[15],$this-
        >kunci_rotasi[15]);
  $R ^= $this->f1($L,$this->kunci_masking[14],$this-
        >kunci_rotasi[14]);
  $L ^= $this->f0($R,$this->kunci_masking[13],$this-
        >kunci_rotasi[13]);
}
$R ^= $this->f2($L,$this->kunci_masking[12],$this-
      >kunci_rotasi[12]);
$L ^= $this->f1($R,$this->kunci_masking[11],$this-
      >kunci_rotasi[11]);
$R ^= $this->f0($L,$this->kunci_masking[10],$this-
      >kunci_rotasi[10]);
$L ^= $this->f2($R,$this->kunci_masking[9],$this-
      >kunci_rotasi[9]);
$R ^= $this->f1($L,$this->kunci_masking[8],$this-
      >kunci_rotasi[8]);
$L ^= $this->f0($R,$this->kunci_masking[7],$this-
      >kunci_rotasi[7]);
$R ^= $this->f2($L,$this->kunci_masking[6],$this-
      >kunci_rotasi[6]);
```

```
$L ^= $this->f1($R,$this->kunci_masking[5],$this-
      >kunci_rotasi[5]);
$R ^= $this->f0($L,$this->kunci_masking[4],$this-
      >kunci_rotasi[4]);
$L ^= $this->f2($R,$this->kunci_masking[3],$this-
      >kunci_rotasi[3]);
$R ^= $this->f1($L,$this->kunci_masking[2],$this-
      >kunci_rotasi[2]);
$L ^= $this->f0($R,$this->kunci_masking[1],$this-
      >kunci_rotasi[1]);

$decrypted = pack("N",$L) . pack("N",$R);

return $decrypted;
}
```

Sourcecode 4.9 Fungsi `DekripsiPerBlok`

4.2.5 Proses avalanche effect

Seperti yang sudah dijelaskan pada subbab 2.6 bahwa untuk mengetahui baik tidaknya sebuah algoritma kriptografi maka dilakukan analisis *avalanche effect*. Berikut adalah implementasi dari *avalanche effect*.

```
function av_ef($ubah, $asli)
{
        $i=0;
        $beda=0;
        while($i<strlen($asli))
        {
                if($ubah[$i]!=$asli[$i])
                {
                        $beda=$beda+1;
                }
                $i++;
        }
        $nilai=($beda/strlen($asli))*100;
        return $nilai;
}
```

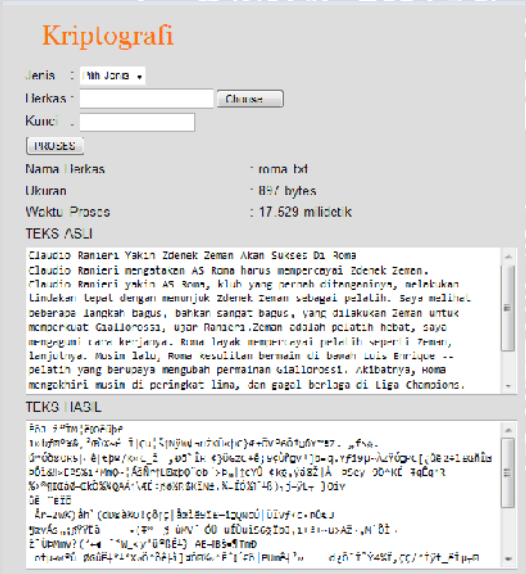
Sourcecode 4.10 Fungsi `av_ef`

Fungsi `av_ef` ini membutuhkan masukan berupa dua buah *string* yang akan dibandingkan, kemudian dibagi panjang salah satu *string* dan dikali seratus. Nilai balikan dari fungsi ini akan berupa nilai *avalanche effect*. Implementasi Fungsi `av_ef` ditunjukan pada *Sourcecode 4.10*.

4.3 Implementasi *Interface*

Berdasarkan rancangan Antarmuka pada subbab 3.4 maka dihasilkan 4 halaman , halaman pertama adalah kriptografi dimana fungsinya untuk melakukan enkripsi atau dekripsi terhadap *file *.txt* kemudian halaman kedua sampai keempat adalah uji1, uji2 dan uji 3 yang berfungsi melakukan analisis *avalanche effect* pada algoritma *CAST-128*.

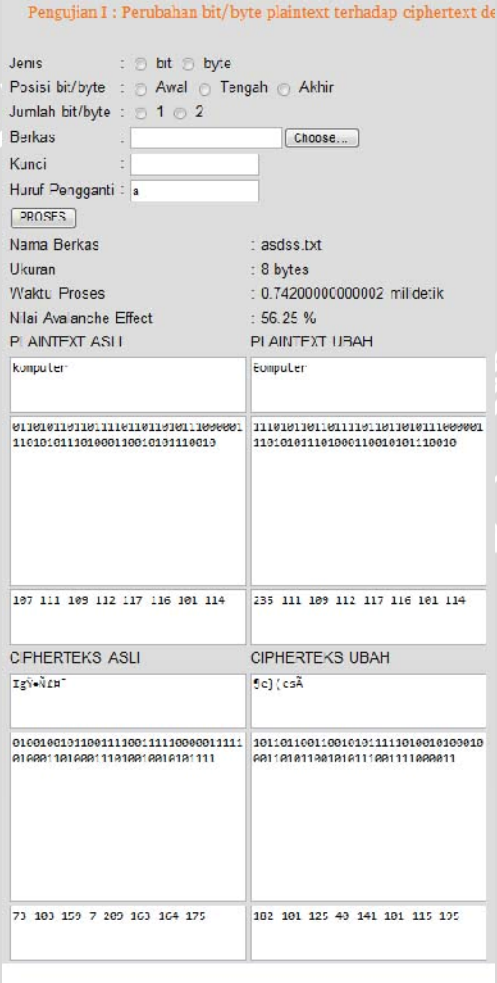
Pada halaman kriptografi ada 2 fitur utama yaitu enkripsi dan dekripsi. Proses berjalannya aplikasi adalah pertama *user* di minta untuk memilih jenis kriptografi, yaitu enkripsi dan dekripsi. Kemudian *user* diminta untuk memilih *file* yang akan diproses dan memasukkan *key*. Setelah klik tombol PROSES maka enkripsi atau dekripsi akan berlangsung dan akan ditampilkan nama berkas, ukuran berkas, waktu proses, teks asli dan teks hasil. Implementasi antarmuka halaman kriptografi dapat dilihat pada gambar 4.1.



Gambar 4.1 Halaman Kriptografi

Pada pengujian I yaitu perubahan bit/*byte plaintext* terhadap *ciphertext* dengan *key* enkripsi yang sama. Proses berjalannya *user* diminta memilih jenis perubahannya, posisi perubahan, jumlah

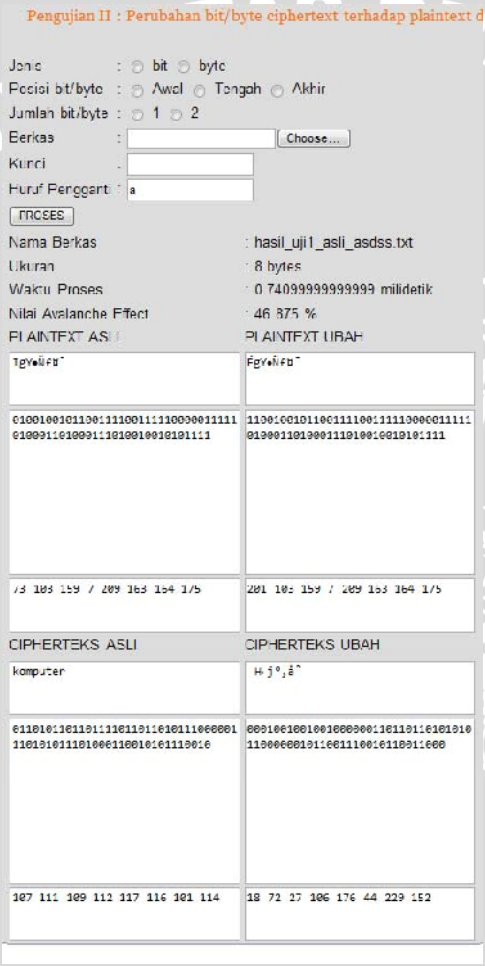
perubahan, memilih berkas, memasukkan kunci dan memasukkan huruf pengganti. Setelah klik tombol PROSES maka enkripsi akan berlangsung dan akan ditampilkan nama berkas, ukuran berkas, waktu proses, teks asli, teks hasil, nilai *avalanche effect*-nya, plaintext asli, plaintext ubah, *ciphertext* asli dan *ciphertext* ubah. Implementasi antarmuka halaman uji1 dapat dilihat pada gambar 4.2.



Gambar 4.2 Halaman Uji1

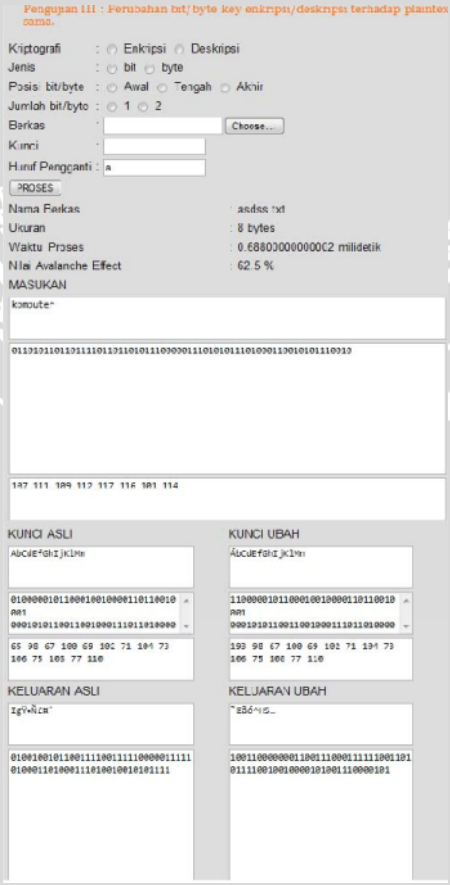
Pada pengujian II yaitu perubahan bit/*byte ciphertext* terhadap *plaintext* dengan *key* enkripsi yang sama. Proses berjalannya *user*

diminta memilih jenis perubahannya, posisi perubahan, jumlah perubahan, memilih berkas, memasukkan kunci dan memasukkan huruf pengganti. Setelah klik tombol PROSES maka dekripsi akan berlangsung dan akan ditampilkan nama berkas, ukuran berkas, waktu proses, teks asli, teks hasil, nilai *avalanche effect*-nya, *plaintext* asli, *plaintext* ubah, *ciphertext* asli dan *ciphertext* ubah. Implementasi antarmuka halaman uji2 dapat dilihat pada gambar 4.3.



Gambar 4.3 Halaman Uji2

Pada pengujian III yaitu perubahan bit/*byte key* enkripsi/dekripsi terhadap *plaintext* dan *ciphertext* yang sama. Proses berjalannya *user* diminta memilih kriptografi, jenis perubahannya, posisi perubahan, jumlah perubahan, memilih berkas, memasukkan kunci dan memasukkan huruf pengganti. Setelah klik tombol PROSES maka dekripsi akan berlangsung dan akan ditampilkan nama berkas, ukuran berkas, waktu proses, teks asli, teks hasil, nilai *avalanche effect*-nya, masukan, kunci asli, kunci ubah, *ciphertext* asli dan *ciphertext* ubah. Dan implementasi antarmuka halaman uji3 dapat dilihat pada gambar 4.4.



Gambar 4.4 Halaman Uji 3

4.4 Hasil Uji

Pada pengujian waktu proses enkripsi dan dekripsi dilakukan uji sebanyak lima kali dengan data yang sama. Pada bab ini hanya akan ditampilkan hasil rata-rata pengujian tersebut. Untuk melihat tabel pengujian secara lengkap dapat dilihat pada lampiran.

Pada tabel 4.1 dan 4.2 akan diperlihatkan hasil uji rata-rata waktu enkripsi dan dekripsi. Kemudian tabel 4.3 sampai 4.8 akan menunjukkan tabel hasil uji dengan parameter *avalanche effect*.

Dari tabel 4.1 dan 4.2 dapat dilihat waktu proses enkripsi dan dekripsi pada tiap *file* uji di atas pada ukuran sama adalah berbeda. Akan tetapi selisih nilainya relatif kecil. Seperti pada ukuran 50KB, rata-rata waktu proses untuk enkripsi sebesar 0,807detik sedangkan pada dekripsi adalah 0,794 detik. Selisihnya adalah 0,011 detik atau 11 milidetik. Detail hasil uji rata-rata waktu proses enkripsi dapat dilihat pada tabel 4.1.

Tabel 4.1 Hasil uji rata-rata waktu proses enkripsi.

Nama *File*	Ukuran *File*	Rata-rata Waktu Proses Enkripsi (s)
50KB.txt	50KB	0,807
100KB.txt	100KB	1,669
150KB.txt	150KB	2,430
200KB.txt	200KB	3,201
250KB.txt	250KB	4,273
300KB.txt	300KB	5,174
350KB.txt	350KB	5,898
400KB.txt	400KB	6,713
450KB.txt	450KB	7,662
500KB.txt	500KB	8,412

Sementara detail hasil uji rata-rata waktu proses dekripsi dapat dilihat pada tabel 4.2.

Tabel 4.2 Hasil uji rata-rata waktu proses dekripsi.

Nama *File*	Ukuran *File*	Rata-rata Waktu Proses Dekripsi (s)
50KB.txt	50KB	0,794
100KB.txt	100KB	1,637
150KB.txt	150KB	2,461
200KB.txt	200KB	3,309
250KB.txt	250KB	4,281
300KB.txt	300KB	5,036
350KB.txt	350KB	6,099
400KB.txt	400KB	6,866
450KB.txt	450KB	7,605
500KB.txt	500KB	8,451

Untuk pengujian terhadap *avalanche effect*. *File* yang akan digunakan adalah 5 buah *file* yang berukuran 8 *byte*, 16 *byte*, 24 *byte*, 32 *byte* dan 40 *byte*. Kemudian beberapa parameter yang diujikan yaitu :

1. Perubahan terhadap *plaintext*, *chipertext*, dan *key*.
2. Perubahan pada bit
3. Perubahan pada *byte*
4. Perubahan pada jumlah *byte*/bit
5. Perubahan pada posisi *byte*/bit

Berikut beberapa tabel hasil pengujian *avalanche effect* :

Yang pertama yaitu perubahan bit dan posisi pada *plaintext* terhadap *chipertext* dengan *key* enkripsi yang sama yaitu ramadhanku dapat dilihat pada tabel 4.3.

Tabel 4.3 Perubahan bit dan posisi pada *plaintext* terhadap *chipertext* dengan *key* enkripsi yang sama (*key*: ramadhanku)

Nama *file*	Ukuran *file*	Jumlah perubahan bit/*byte*	Posisi perubahan	*Avalanche effect*
8byte.txt	8 *byte*	1 bit	awal	57.81
16byte.txt	16*byte*	1 bit	awal	55.47
24byte.txt	24*byte*	1 bit	awal	54.69
32byte.txt	32*byte*	1 bit	awal	55.08
40byte.txt	40*byte*	1 bit	awal	54.38
rata-rata				55.49
8byte.txt	8 *byte*	1 bit	tengah	54.69
16byte.txt	16*byte*	1 bit	tengah	24.22
24byte.txt	24*byte*	1 bit	tengah	36.46
32byte.txt	32*byte*	1 bit	tengah	24.61
40byte.txt	40*byte*	1 bit	tengah	31.88
rata -rata				34.37
8byte.txt	8 *byte*	1 bit	akhir	40.63
16byte.txt	16*byte*	1 bit	akhir	29.69
24byte.txt	24*byte*	1 bit	akhir	16.15
32byte.txt	32*byte*	1 bit	akhir	10.55
40byte.txt	40*byte*	1 bit	akhir	9.69
rata-rata				21.34
8byte.txt	8 *byte*	2 bit	awal	54.69
16byte.txt	16*byte*	2 bit	awal	52.34
24byte.txt	24*byte*	2 bit	awal	51.56
32byte.txt	32*byte*	2 bit	awal	51.95
40byte.txt	40*byte*	2 bit	awal	53.13
rata-rata				52.57
8byte.txt	8 *byte*	2 bit	tengah	56.25
16byte.txt	16*byte*	2 bit	tengah	25
24byte.txt	24*byte*	2 bit	tengah	35.42
32byte.txt	32*byte*	2 bit	tengah	27.34
40byte.txt	40*byte*	2 bit	tengah	30

rata -rata				34,80
8byte.txt	8 *byte*	2 bit	akhir	40.63
16byte.txt	16*byte*	2 bit	akhir	19.53
24byte.txt	24*byte*	2 bit	akhir	18.75
32byte.txt	32*byte*	2 bit	akhir	11.72
40byte.txt	40*byte*	2 bit	akhir	10
rata-rata				20.13
rata-rata total				36.48

Dari perubahan 1 bit di awal dari *plaintext* terhadap *chipertext* dengan *key* yang sama diperoleh rata-rata nilai 55.49 kemudian untuk perubahan 1 bit di tengah rata-rata nilai *avalanche effect*nya 34.37, untuk perubahan 1 bit di akhir 21.34.

Kemudian untuk perubahan 2 bit di awal rata-rata nilainya adalah 52.57, di tengah adalah 34,80 dan di akhir adalah 20.13.

Nilai rata-rata total adalah sebesar 36.48, Nilai tersebut masih kurang dari nilai seharusnya yang diharapkan yaitu 45 persen yang merupakan nilai ideal untuk *avalanche effect*.

Kemudian yang kedua yaitu perubahan *byte* dan posisi pada *plaintext* terhadap *chipertext* dengan *key* enkripsi yang sama yaitu ramadhanku dengan karakter pengganti ‘a’ dapat dilihat pada tabel 4.4.

Tabel 4.4 Perubahan *byte* dan posisi pada *plaintext* terhadap *chipertext* dengan *key* enkripsi yang sama (*key*: ramadhanku, karakter pengganti 'a')

Nama *file*	Ukuran *file*	Jumlah perubahan bit/*byte*	Posisi perubahan	*Avalanche effect*
8byte.txt	8 *byte*	1 byte	awal	53.13
16byte.txt	16*byte*	1 byte	awal	51.56
24byte.txt	24*byte*	1 byte	awal	54.69
32byte.txt	32*byte*	1 byte	awal	54.3
40byte.txt	40*byte*	1 byte	awal	50.94

rata-rata				52.92
8byte.txt	8 *byte*	1 byte	tengah	54.69
16byte.txt	16*byte*	1 byte	tengah	28.13
24byte.txt	24*byte*	1 byte	tengah	33.33
32byte.txt	32*byte*	1 byte	tengah	26.95
40byte.txt	40*byte*	1 byte	tengah	31.88
rata -rata				35.00
8byte.txt	8 *byte*	1 byte	akhir	51.56
16byte.txt	16*byte*	1 byte	akhir	25.78
24byte.txt	24*byte*	1 byte	akhir	16.67
32byte.txt	32*byte*	1 byte	akhir	10.94
40byte.txt	40*byte*	1 byte	akhir	11.88
rata-rata				23.37
8byte.txt	8 *byte*	2 byte	awal	56.25
16byte.txt	16*byte*	2 byte	awal	50.78
24byte.txt	24*byte*	2 byte	awal	50
32byte.txt	32*byte*	2 byte	awal	48.05
40byte.txt	40*byte*	2 byte	awal	47.5
rata-rata				50.52
8byte.txt	8 *byte*	2 byte	tengah	50
16byte.txt	16*byte*	2 byte	tengah	25.78
24byte.txt	24*byte*	2 byte	tengah	34.38
32byte.txt	32*byte*	2 byte	tengah	22.27
40byte.txt	40*byte*	2 byte	tengah	31.56
rata -rata				32.80
8byte.txt	8 *byte*	2 byte	akhir	45.31
16byte.txt	16*byte*	2 byte	akhir	26.56
24byte.txt	24*byte*	2 byte	akhir	15.63
32byte.txt	32*byte*	2 byte	akhir	12.5
40byte.txt	40*byte*	2 byte	akhir	11.56
rata-rata				22.31
rata-rata total				36.15

Dari perubahan 1 *byte* di awal dari *plaintext* terhadap *chipertext* dengan *key* yang sama diperoleh rata-rata nilai 52.92 kemudian untuk perubahan 1 *byte* di tengah rata-rata nilai *avalanche effectnya* 35.00, untuk perubahan 1 bit di akhir 23.37.

Kemudian untuk perubahan 2 *byte* di awal rata-rata nilainya adalah 50.52, di tengah adalah 32.80 dan di akhir adalah 22.31.

Nilai rata-rata total adalah sebesar 36.15. Nilai tersebut masih kurang dari nilai seharusnya yang diharapkan yaitu 45 persen yang merupakan nilai ideal untuk *avalanche effect*.

Kemudian yang ketiga yaitu perubahan bit dan posisi pada *key* enkripsi terhadap *chipertext* dengan *plaintext* yang sama yaitu ramadhanku dapat dilihat pada tabel 4.5.

Tabel 4.5 Perubahan bit serta posisinya pada *key* enkripsi terhadap *chipertext* dengan *plaintext* yang sama. (*key*: ramadhanku)

Nama *file*	Ukuran *file*	Jumlah perubahan bit/*byte*	Posisi perubahan	*Avalanche effect*
8byte.txt	8 *byte*	1 bit	awal	46.88
16byte.txt	16*byte*	1 bit	awal	50.78
24byte.txt	24*byte*	1 bit	awal	46.88
32byte.txt	32*byte*	1 bit	awal	47.66
40byte.txt	40*byte*	1 bit	awal	46.56
rata-rata				47.75
8byte.txt	8 *byte*	1 bit	tengah	50
16byte.txt	16*byte*	1 bit	tengah	55.47
24byte.txt	24*byte*	1 bit	tengah	50.52
32byte.txt	32*byte*	1 bit	tengah	53.52
40byte.txt	40*byte*	1 bit	tengah	51.56
rata -rata				52.21
8byte.txt	8 *byte*	1 bit	akhir	56.25
16byte.txt	16*byte*	1 bit	akhir	51.56
24byte.txt	24*byte*	1 bit	akhir	52.08

32byte.txt	32*byte*	1 bit	akhir	50.78
40byte.txt	40*byte*	1 bit	akhir	51.25
rata-rata				52.38
8byte.txt	8 *byte*	2 bit	awal	42.19
16byte.txt	16*byte*	2 bit	awal	53.13
24byte.txt	24*byte*	2 bit	awal	52.08
32byte.txt	32*byte*	2 bit	awal	51.56
40byte.txt	40*byte*	2 bit	awal	50.63
rata-rata				49.92
8byte.txt	8 *byte*	2 bit	tengah	56.25
16byte.txt	16*byte*	2 bit	tengah	52.34
24byte.txt	24*byte*	2 bit	tengah	51.56
32byte.txt	32*byte*	2 bit	tengah	47.27
40byte.txt	40*byte*	2 bit	tengah	47.19
rata -rata				50.92
8byte.txt	8 *byte*	2 bit	akhir	51.56
16byte.txt	16*byte*	2 bit	akhir	46.09
24byte.txt	24*byte*	2 bit	akhir	47.4
32byte.txt	32*byte*	2 bit	akhir	45.7
40byte.txt	40*byte*	2 bit	akhir	46.88
rata-rata				47.53
rata-rata total				50.12

Dari perubahan 1 bit di awal dari *key* terhadap *chipertext* dengan *plaintext* yang sama diperoleh rata-rata nilai 47.75 kemudian untuk perubahan 1 bit di tengah rata-rata nilai *avalanche effectnya* 52.21, untuk perubahan 1 bit di akhir adalah 52.38.

Kemudian untuk perubahan 2 bit di awal rata-rata nilainya adalah 49.92, di tengah adalah 50.92 dan di akhir adalah 47.53.

Nilai rata-rata total adalah sebesar 50.12. Nilai tersebut merupakan nilai ideal untuk *avalanche effect* karena lebih dari 45%.

Kemudian yang keempat yaitu perubahan *byte* dan posisi pada *key* enkripsi terhadap *chipertext* dengan *plaintext* yang sama yaitu

ramadhanku dengan karakter pengganti ‘a’ dapat dilihat pada tabel 4.6.

Tabel 4.6 Perubahan *byte* serta posisinya pada *key* enkripsi terhadap *chipertext* dengan *plaintext* yang sama. (*key:* ramadhanku, karakter pengganti = 'a')

Nama *file*	Ukuran *file*	Jumlah perubahan bit/*byte*	Posisi perubahan	*Avalanche effect*
8byte.txt	8 *byte*	1 byte	awal	46.88
16byte.txt	16*byte*	1 byte	awal	51.56
24byte.txt	24*byte*	1 byte	awal	56.25
32byte.txt	32*byte*	1 byte	awal	55.08
40byte.txt	40*byte*	1 byte	awal	52.81
rata-rata				52.52
8byte.txt	8 *byte*	1 byte	tengah	50
16byte.txt	16*byte*	1 byte	tengah	53.91
24byte.txt	24*byte*	1 byte	tengah	52.6
32byte.txt	32*byte*	1 byte	tengah	51.95
40byte.txt	40*byte*	1 byte	tengah	51.88
rata -rata				52.07
8byte.txt	8 *byte*	1 byte	akhir	46.88
16byte.txt	16*byte*	1 byte	akhir	50.78
24byte.txt	24*byte*	1 byte	akhir	50
32byte.txt	32*byte*	1 byte	akhir	49.61
40byte.txt	40*byte*	1 byte	akhir	48.75
rata-rata				49.20
8byte.txt	8 *byte*	2 byte	awal	46.88
16byte.txt	16*byte*	2 byte	awal	51.56
24byte.txt	24*byte*	2 byte	awal	56.25
32byte.txt	32*byte*	2 byte	awal	55.08
40byte.txt	40*byte*	2 byte	awal	52.81
rata-rata				52.52

8byte.txt	8 *byte*	2 byte	tengah	50
16byte.txt	16*byte*	2 byte	tengah	53.91
24byte.txt	24*byte*	2 byte	tengah	52.6
32byte.txt	32*byte*	2 byte	tengah	51.95
40byte.txt	40*byte*	2 byte	tengah	51.88
rata -rata				52.07
8byte.txt	8 *byte*	2 byte	akhir	34.38
16byte.txt	16*byte*	2 byte	akhir	37.5
24byte.txt	24*byte*	2 byte	akhir	41.15
32byte.txt	32*byte*	2 byte	akhir	44.53
40byte.txt	40*byte*	2 byte	akhir	44.38
rata-rata				40.39
rata-rata total				49.79

Dari perubahan 1 *byte* di awal dari *key* terhadap *chipertext* dengan *plaintext* yang sama diperoleh rata-rata nilai 52.52, kemudian untuk perubahan 1 *byte* di tengah rata-rata nilai *avalanche effectnya* 52.07, untuk perubahan 1 *byte* di akhir adalah 49.20.

Kemudian untuk perubahan 2 *byte* di awal rata-rata nilainya adalah 52.52, di tengah adalah 52.07 dan di akhir adalah 40.39.

Nilai total adalah sebesar 49.79. Nilai tersebut merupakan nilai ideal untuk *avalanche effect* karena lebih dari 45.

Kemudian yang keempat yaitu perubahan bit dan posisi pada *chipertext* terhadap *plainteks* dengan *key* yang sama yaitu ramadhanku dapat dilihat pada tabel 4.7.

Tabel 4.7 Perubahan bit dan posisi pada *chipertext* terhadap *plaintext* dengan *key* dekripsi yang sama (*key*: ramadhanku)

Nama *file*	Ukuran *file*	Jumlah perubahan bit/*byte*	Posisi perubahan	*Avalanche effect*
8byte.txt	8 *byte*	1 bit	awal	54.69
16byte.txt	16*byte*	1 bit	awal	28.13

24byte.txt	24*byte*	1 bit	awal	18.75
32byte.txt	32*byte*	1 bit	awal	14.06
40byte.txt	40*byte*	1 bit	awal	11.25
rata-rata				25.38
8byte.txt	8 *byte*	1 bit	tengah	60.94
16byte.txt	16*byte*	1 bit	tengah	25
24byte.txt	24*byte*	1 bit	tengah	18.23
32byte.txt	32*byte*	1 bit	tengah	12.11
40byte.txt	40*byte*	1 bit	tengah	10
rata -rata				25.26
8byte.txt	8 *byte*	1 bit	akhir	51.56
16byte.txt	16*byte*	1 bit	akhir	21.09
24byte.txt	24*byte*	1 bit	akhir	17.19
32byte.txt	32*byte*	1 bit	akhir	12.89
40byte.txt	40*byte*	1 bit	akhir	10.94
rata-rata				22.73
8byte.txt	8 *byte*	2 bit	awal	42.19
16byte.txt	16*byte*	2 bit	awal	22.66
24byte.txt	24*byte*	2 bit	awal	15.1
32byte.txt	32*byte*	2 bit	awal	11.33
40byte.txt	40*byte*	2 bit	awal	9.06
rata-rata				20.07
8byte.txt	8 *byte*	2 bit	tengah	60.94
16byte.txt	16*byte*	2 bit	tengah	23.44
24byte.txt	24*byte*	2 bit	tengah	13.02
32byte.txt	32*byte*	2 bit	tengah	12.5
40byte.txt	40*byte*	2 bit	tengah	10
rata -rata				23.98
8byte.txt	8 *byte*	2 bit	akhir	51.56
16byte.txt	16*byte*	2 bit	akhir	24.22
24byte.txt	24*byte*	2 bit	akhir	15.1
32byte.txt	32*byte*	2 bit	akhir	11.72

40byte.txt	40*byte*	2 bit	akhir	11.88
rata-rata				22.90
rata-rata total				23.39

Dari perubahan 1 bit di awal dari *chipertext* terhadap *plaintext* dengan *key* yang sama diperoleh rata-rata nilai 25.38 kemudian untuk perubahan 1 bit di tengah rata-rata nilai *avalanche effectnya* 25.26, untuk perubahan 1 bit di akhir 22.73.

Kemudian untuk perubahan 2 bit di awal rata-rata nilainya adalah 20.07, di tengah adalah 23.98 dan di akhir adalah 22.90.

Nilai rata-rata total adalah sebesar 23.39. Nilai tersebut masih kurang dari nilai seharusnya yang diharapkan yaitu 45 persen yang merupakan nilai ideal untuk *avalanche effect.*

Kemudian yang keempat yaitu perubahan *byte* dan posisi pada *chipertext* terhadap *plainteks* dengan *key* yang sama yaitu ramadhanku dengan karakter pengganti ‘a’ dapat dilihat pada tabel 4.8.

Tabel 4.8 Perubahan *byte* dan posisi pada *chipertext* terhadap *plainteks* dengan *key* dekripsi yang sama (*key*: ramadhanku, karakter pengganti 'a')

Nama *file*	Ukuran *file*	Jumlah perubahan bit/*byte*	Posisi perubahan	*Avalanche effect*
8byte.txt	8 *byte*	1 byte	awal	39.06
16byte.txt	16*byte*	1 byte	awal	22.66
24byte.txt	24*byte*	1 byte	awal	15.1
32byte.txt	32*byte*	1 byte	awal	11.33
40byte.txt	40*byte*	1 byte	awal	9.06
rata-rata				19.44
8byte.txt	8 *byte*	1 byte	tengah	54.69
16byte.txt	16*byte*	1 byte	tengah	21.88
24byte.txt	24*byte*	1 byte	tengah	20.83
32byte.txt	32*byte*	1 byte	tengah	12.11
40byte.txt	40*byte*	1 byte	tengah	8.44

rata -rata				23.59
8byte.txt	8 *byte*	1 byte	akhir	56.25
16byte.txt	16*byte*	1 byte	akhir	22.66
24byte.txt	24*byte*	1 byte	akhir	15.1
32byte.txt	32*byte*	1 byte	akhir	12.5
40byte.txt	40*byte*	1 byte	akhir	10.94
rata-rata				23.49
8byte.txt	8 *byte*	2 byte	awal	57.81
16byte.txt	16*byte*	2 byte	awal	33.59
24byte.txt	24*byte*	2 byte	awal	22.4
32byte.txt	32*byte*	2 byte	awal	16.8
40byte.txt	40*byte*	2 byte	awal	13.44
rata-rata				28.81
8byte.txt	8 *byte*	2 byte	tengah	53.13
16byte.txt	16*byte*	2 byte	tengah	21.09
24byte.txt	24*byte*	2 byte	tengah	18.75
32byte.txt	32*byte*	2 byte	tengah	13.67
40byte.txt	40*byte*	2 byte	tengah	10.31
rata -rata				23.39
8byte.txt	8 *byte*	2 byte	akhir	42.19
16byte.txt	16*byte*	2 byte	akhir	27.34
24byte.txt	24*byte*	2 byte	akhir	15.63
32byte.txt	32*byte*	2 byte	akhir	15.23
40byte.txt	40*byte*	2 byte	akhir	9.06
rata-rata				21.89
rata-rata total				23.44

Dari perubahan 1 *byte* di awal dari *chipertext* terhadap *plaintext* dengan *key* yang sama diperoleh rata-rata nilai 19.44, kemudian untuk perubahan 1 *byte* di tengah rata-rata nilai *avalanche effect*nya 23.59, untuk perubahan 1 *byte* di akhir 23.49.

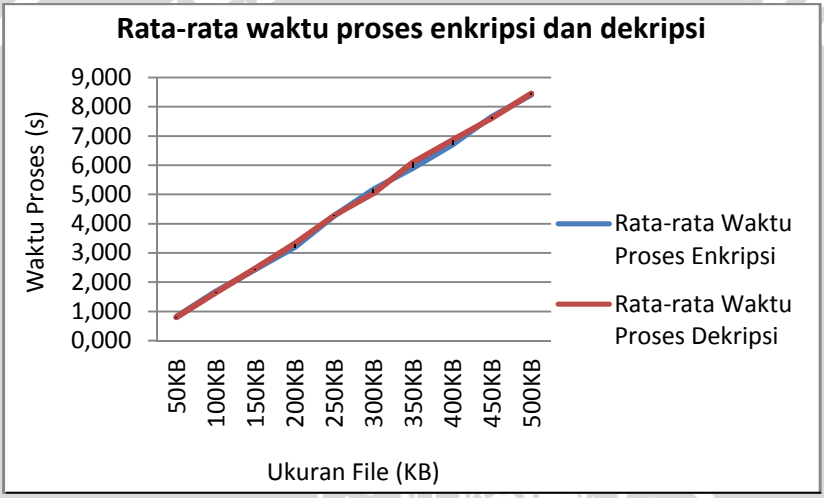
Kemudian untuk perubahan 2 *byte* di awal rata-rata nilainya adalah 28.81, di tengah adalah 23.39 dan di akhir adalah 21.89.

Nilai rata-rata total adalah sebesar 23.44. Nilai tersebut masih kurang dari nilai seharusnya yang diharapkan yaitu 45 persen yang merupakan nilai ideal untuk *avalanche effect.*

4.5 Analisis hasil

Pada subbab ini akan dilakukan analisis terhadap hasil uji yang didapatkan pada subbab 4.4.

Pertama akan dilakukan analisis pada hasil uji coba waktu proses, dapat dilihat pada grafik 4.1.

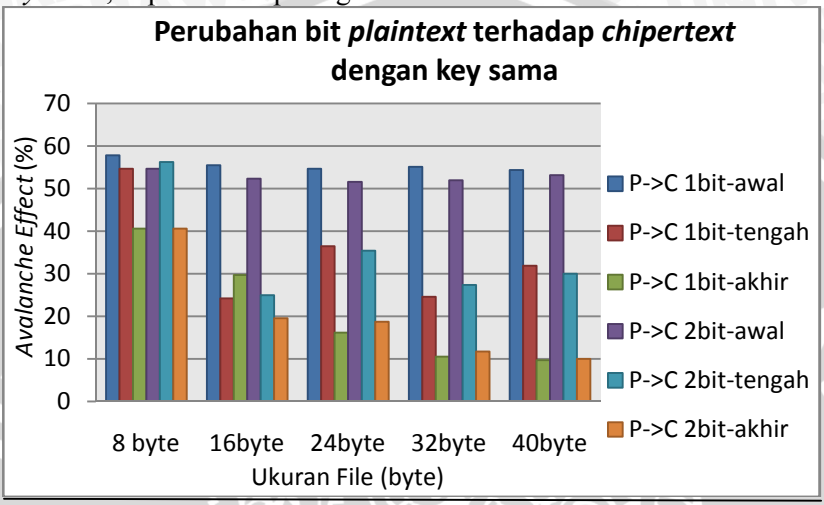


Grafik 4.1 Rata-rata waktu proses enkripsi dan dekripsi

Grafik rata-rata waktu dekripsi berada agak sejajar dengan grafik rata-rata waktu enkripsi. Hal ini mengindikasikan bahwa rata-rata waktu dekripsi hampir sama dengan rata-rata waktu enkripsi.

Kemudian yang kedua akan dilakukan analisis pada hasil uji *avalanche effect*, pada pembuatan grafik dari hasil uji *avalanche effect* didapatkan 6 buah grafik (dapat dilihat pada lampiran) dimana terdapat 2 bentuk grafik yang unik. Yaitu bentuk yang cenderung menurun dan bentuk yang cenderung naik turun di atas angka 40. Bentuk yang cenderung menurun dimiliki oleh grafik perubahan *plaintext* terhadap *chipertext* dan perubahan *chipertext* terhadap *plaintext* sedangkan bentuk yang kedua dimiliki oleh grafik perubahan *key* terhadap *chipertext.*

Kemudian akan dilakukan analisis pada hasil perhitungan nilai *avalanche effect* perubahan bit plaintext terhadap chipertext dengan *key* sama, dapat dilihat pada grafik 4.2.



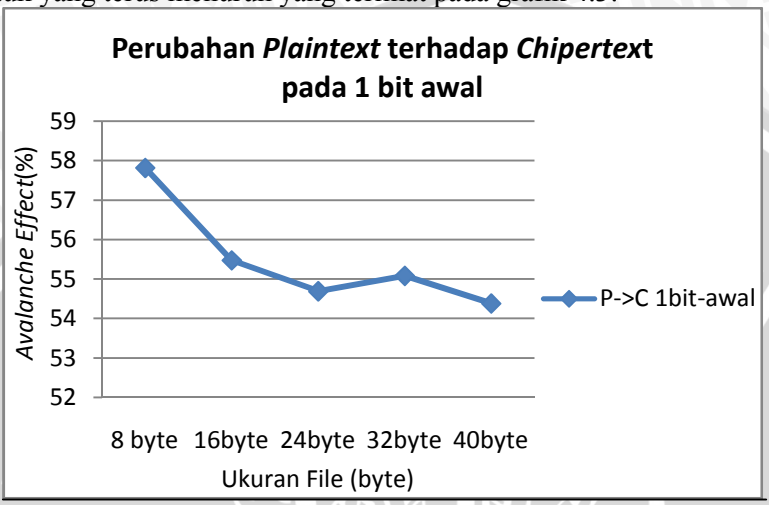
Grafik 4.2 perubahan bit *plaintext* terhadap *chipertext*

Pada grafik perubahan bit *plaintext* terhadap *chipertext* (dengan pengecualian pada grafik kelima pada tiap ukuran *file*). dapat dilihat bentuk grafik pada umumnya adalah menurun dan nilai maksimum adalah pada ukuran 8 *byte*, hal ini dikarenakan algoritma *CAST-128* mengolah *plaintext* per blok yang berisi 64 bit atau 8 *byte*. Sehingga apabila dilakukan perubahan pada satu bit pada blok tertentu maka hanya isi blok tersebut yang berubah, sedangkan blok lainnya tidak berubah sehingga nilai *avalanche effect* pada ukuran *file* yang besar cenderung lebih kecil.

Contohnya saja pada perubahan *plaintext* terhadap *chipertext* pada satu bit awal. Perubahan pada *chipertext* akan diilustrasikan sebagai berikut(yang dicetak tebal menunjukkan blok yang berubah):

8 *byte* = **blok1**
16 *byte* = **blok1**-blok2
24 *byte* = **blok1**-blok2-blok3
32 *byte* = **blok1**-blok2-blok3-blok4
40 *byte* = **blok1**-blok2-blok3-blok4-blok5

Dari ilustrasi tersebut tentu saja dapat diperoleh jawaban dari bentuk yang terus menurun yang terlihat pada grafik 4.3.



Grafik 4.3 Perubahan *Plaintext* terhadap *Chipertex*t pada 1 bit awal

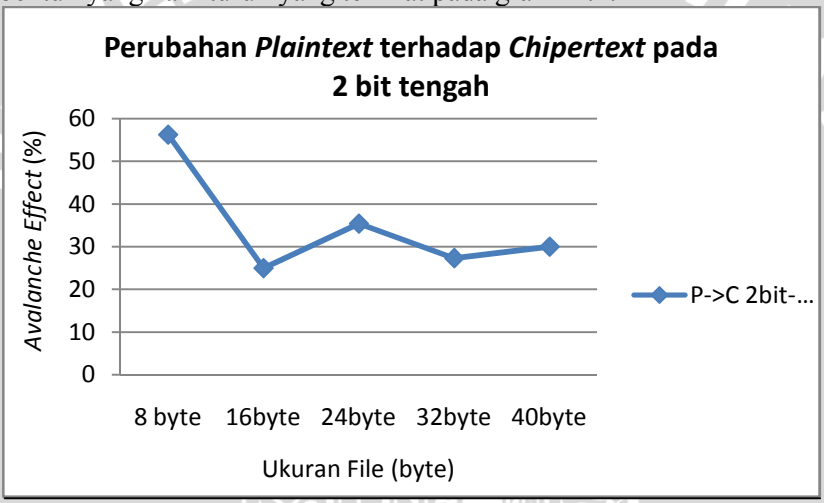
Akan tetapi hal itu tidak terjadi pada perubahan 2 bit pada posisi tengah (grafik perubahan bit *plaintext* terhadap *chipertext* grafik kelima pada tiap ukuran *file*), bentuk grafiknya cenderung naik turun, bisa dilihat pada grafik di atas pada ukuran 8 dan 16 *byte* grafiknya akan sama, kemudian pada 24 *byte* turun lalu pada ukuran 32 *byte* naik lagi dan pada 40 *byte* turun lagi. Hal ini disebabkan pada perubahan 2 bit pada posisi tengah pada *file* yang mempunyai blok genap akan melibatkan 2 blok dan otomatis 2 blok tadi akan berubah. Sedangkan pada *file* yang memiliki blok ganjil misalnya 24 *bytes* yang memiliki 3 blok hanya akan melibatkan 1 blok saja yang berubah.

Misalnya saja pada grafik perubahan bit *plaintext* terhadap *chipertext* (grafik kelima pada tiap ukuran *file*), pada ukuran 16 *byte* 2 bit tengah yang akan diubah yaitu bit terakhir dari blok pertama dan bit pertama dari blok ke dua, maka blok delapan dan sembilan ini akan berubah saat dilakukan enkripsi, beda halnya dengan *file* yang berukuran 24 *byte*. Karena terdiri dari 3 blok, tentu saja 2 bit yang akan diubah akan berada pada blok yang di tengah yaitu blok kedua, sehingga perubahan hanya terjadi pada blok ke dua saja.

Contohnya saja pada perubahan *plaintext* terhadap *chipertext* pada dua bit tengah. Perubahan pada *chipertext* akan diilustrasikan sebagai berikut(yang dicetak tebal menunjukkan blok yang berubah):

8 *byte* = **blok1**
16*byte* = **blok1-blok2**
24*byte* = blok1-**blok2**-blok3
32*byte* = blok1-**blok2-blok3**-blok4
40*byte* = blok1-blok2-**blok3**-blok4-blok5

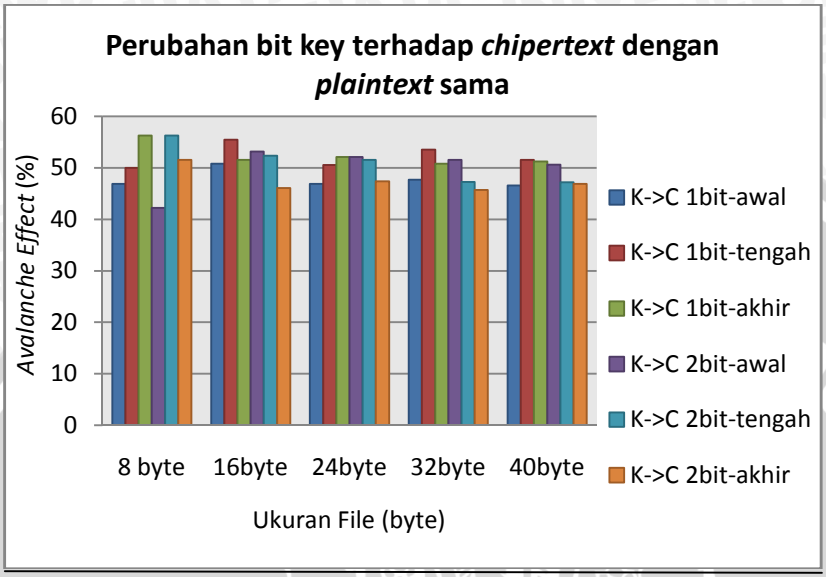
Dari ilustrasi tersebut tentu saja dapat diperoleh jawaban dari bentuk yang naik turun yang terlihat pada grafik 4.4.



Grafik 4.4 Perubahan *Plaintext* terhadap *Chipertext* pada 2 bit tengah

Bentuk grafik selanjutnya adalah bentuk grafik yang cenderung naik turun. Grafik ini dimiliki oleh perubahan *key* terhadap *chipertext* baik itu perubahan bit maupun perubahan *byte*.

Hal ini terjadi karena perubahan *key* terhadap *chipertext* tidak langsung berpengaruh pada blok sebuah *plaintext*. Akan tetapi berpengaruh pada kunci *masking* dan kunci rotasi pada jaringan *feistel*. Sehingga pada ukuran besar nilai *avalanche effect* tidak semakin kecil seperti grafik grafik perubahan bit *plaintext* terhadap *chipertext*. Hal ini dapat dilihat pada perubahan bit *key* terhadap *chipertext*. Dan hal tersebut dapat dilihat pada grafik 4.5.



Grafik 4.5 Perubahan bit *key* terhadap *chipertext*

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

1. Telah diimplementasikan sebuah perangkat lunak yang menggunakan algoritma kriptografi *CAST-128* kemudian dari uji waktu proses algoritma *CAST-128,* diperoleh kesimpulan bahwa waktu proses enkripsi untuk ukuran *file* 50KB rata-rata waktu enkripsi adalah sebesar 0,807 detik, untuk ukuran *file* 500KB rata-rata waktu enkripsi adalah sebesar 8,412 detik kemudian rata-rata waktu dekripsi untuk ukuran *file* 50KB adalah sebesar 0,794 detik, untuk ukuran *file* 500KB adalah sebesar 8,451 detik.
2. Dari uji *avalanche effect* dapat disimpulkan bahwa nilai rata-rata perubahan yang menghasilkan nilai *avalanche effect* yang ideal (45% sampai 60%) adalah pada perubahan *key,* dimana pada perubahan *byte* menghasilkan nilai 50,12% dan pada perubahan bit menghasilkan nilai 49,79%.
3. Kemudian untuk nilai rata-rata perubahan yang menghasilkan nilai *avalanche effect* yang kurang ideal adalah pada perubahan *chipertext dan plaintext,* dimana pada perubahan *byte plaintext* menghasilkan nilai 36,15%, pada perubahan bit *plaintext* menghasilkan nilai 36,48%, pada perubahan *byte chipertext* menghasilkan nilai 23,44% dan pada perubahan bit *chipertext* menghasilkan nilai 23,39%.

Dengan nilai *avalanche effect* yang dihasilkan pada pengujian terhadap *key*, dapat disimpulkan bahwa algoritma *CAST-128* cukup tahan terhadap serangan kriptanalisis.

5.2 Saran

Saran yang mungkin bisa jadi bahan pertimbangan adalah pada pengujian *avalanche effect* posisi pengujian bit/*byte* bisa di buat lebih bervariasi lagi. Begitu juga pada jumlah bit/*byte* yang diuji bisa ditambah lagi untuk mendapatkan hasil yang lebih akurat.


UNIVERSITAS BRAWIJAYA

DAFTAR PUSTAKA

Adams, Casrlisle M. *Constructing Symmetric Ciphers Using the CAST Design Procedure*, Entrust Technologies, Kanada

Anonymous. Ascii Table and description. http://www.asciitable.com/ diakses tanggal 20 Maret 2012 Pukul 20.00 WIB

Anonymous. RAID information–definition. http://www.accs.com/p_and_p/RAID/Definitions.html diakses tanggal 20 Maret 2012 Pukul 21.00 WIB.

Ariesanda, Boyke. Analisis dan Pengembangan Merkle-Damgård *Structure,* ITB.Bandung.

Ariyanto, Endro.2008. Analisa Implementasi Algortima Stream Cipher Sosemanuk dan Dicing dalam Proses Enkripsi Data. Departemen Teknik Informatika Institut Teknologi Telkom, Bandung

Ariyus, Doni.2007. Keamanan Multimedia,Andi.Yogyakarta.

Ariyus, Doni.2008. Kriptografi,Andi.Yogyakarta

As'ad, Nabila.2010. Studi Analisis Algortima CAST dan Implementasinya dalam PGP, Program Studi Teknik Informatika, Institut Teknologi Bandung. Bandung.

Gunawan, Ade.2006. Studi mengenai Algoritma simetri CAST-128 dan aplikasinya. Program Studi Teknik Informatika, Institut Teknologi Bandung. Bandung.

Krishnamurthy.2009. *Encryption Quality Analysis and Security Evaluation of CAST-128 Algorithm and its Modified Version using Digital Images*. International Journal of Network Security & Its Applications (IJNSA), Vol.1, No 1.

Munir, Rinaldi.2004. Bahan Kuliah IF5054 Kriptografi. Departemen Teknik Informatika, Institut Teknologi Bandung. Bandung

Rudianto.2004. Analisis Keamanan Algoritma Kriptografi RC6. Jurusan Teknik Informatika, ITB. Bandung.

Schneier, Bruce .1996. Applied Cryptography - Protocol, Algorithm, and Source Code in C, second edition, JohnWilley & Sons. Hoboken.

LAMPIRAN 1

```
SBOX I
$s1 = array(
0x30fb40d4, 0x9fa0ff0b, 0x6beccd2f, 0x3f258c7a, 0x1e213f2f,
0x9c004dd3, 0x6003e540, 0xcf9fc949, 0xbfd4af27, 0x88bbbdb5,
0xe2034090, 0x98d09675, 0x6e63a0e0, 0x15c361d2, 0xc2e7661d,
0x22d4ff8e, 0x28683b6f, 0xc07fd059, 0xff2379c8, 0x775f50e2,
0x43c340d3, 0xdf2f8656, 0x887ca41a, 0xa2d2bd2d, 0xa1c9e0d6,
0x346c4819, 0x61b76d87, 0x22540f2f, 0x2abe32e1, 0xaa54166b,
0x22568e3a, 0xa2d341d0, 0x66db40c8, 0xa784392f, 0x004dff2f,
0x2db9d2de, 0x97943fac, 0x4a97c1d8, 0x527644b7, 0xb5f437a7,
0xb82cbaef, 0xd751d159, 0x6ff7f0ed, 0x5a097a1f, 0x827b68d0,
0x90ecf52e, 0x22b0c054, 0xbc8e5935, 0x4b6d2f7f, 0x50bb64a2,
0xd2664910, 0xbee5812d, 0xb7332290, 0xe93b159f, 0xb48ee411,
0x4bff345d, 0xfd45c240, 0xad31973f, 0xc4f6d02e, 0x55fc8165,
0xd5b1caad, 0xa1ac2dae, 0xa2d4b76d, 0xc19b0c50, 0x882240f2,
0x0c6e4f38, 0xa4e4bfd7, 0x4f5ba272, 0x564c1d2f, 0xc59c5319,
0xb949e354, 0xb04669fe, 0xb1b6ab8a, 0xc71358dd, 0x6385c545,
0x110f935d, 0x57538ad5, 0x6a390493, 0xe63d37e0, 0x2a54f6b3,
0x3a787d5f, 0x6276a0b5, 0x19a6fcdf, 0x7a42206a, 0x29f9d4d5,
0xf61b1891, 0xbb72275e, 0xaa508167, 0x38901091, 0xc6b505eb,
0x84c7cb8c, 0x2ad75a0f, 0x874a1427, 0xa2d1936b, 0x2ad286af,
0xaa56d291, 0xd7894360, 0x425c750d, 0x93b39e26, 0x187184c9,
0x6c00b32d, 0x73e2bb14, 0xa0bebc3c, 0x54623779, 0x64459eab,
0x3f328b82, 0x7718cf82, 0x59a2cea6, 0x04ee002e, 0x89fe78e6,
0x3fab0950, 0x325ff6c2, 0x81383f05, 0x6963c5c8, 0x76cb5ad6,
0xd49974c9, 0xca180dcf, 0x380782d5, 0xc7fa5cf6, 0x8ac31511,
0x35e79e13, 0x47da91d0, 0xf40f9086, 0xa7e2419e, 0x31366241,
0x051ef495, 0xaa573b04, 0x4a805d8d, 0x548300d0, 0x00322a3c,
0xbf64cddf, 0xba57a68e, 0x75c6372b, 0x50afd341, 0xa7c13275,
0x915a0bf5, 0x6b54bfab, 0x2b0b1426, 0xab4cc9d7, 0x449ccd82,
0xf7fbf265, 0xab85c5f3, 0x1b55db94, 0xaad4e324, 0xcfa4bd3f,
0x2deaa3e2, 0x9e204d02, 0xc8bd25ac, 0xeadf55b3, 0xd5bd9e98,
0xe31231b2, 0x2ad5ad6c, 0x954329de, 0xadbe4528, 0xd8710f69,
0xaa51c90f, 0xaa786bf6, 0x22513f1e, 0xaa51a79b, 0x2ad344cc,
0x7b5a41f0, 0xd37cfbad, 0x1b069505, 0x41ece491, 0xb4c332e6,
0x032268d4, 0xc9600acc, 0xce387e6d, 0xbf6bb16c, 0x6a70fb78,
0x0d03d9c9, 0xd4df39de, 0xe01063da, 0x4736f464, 0x5ad328d8,
0xb347cc96, 0x75bb0fc3, 0x98511bfb, 0x4ffbcc35, 0xb58bcf6a,
0xe11f0abc, 0xbfc5fe4a, 0xa70aec10, 0xac39570a, 0x3f04442f,
0x6188b153, 0xe0397a2e, 0x5727cb79, 0x9ceb418f, 0x1cacd68d,
0x2ad37c96, 0x0175cb9d, 0xc69dff09, 0xc75b65f0, 0xd9db40d8,
0xec0e7779, 0x4744ead4, 0xb11c3274, 0xdd24cb9e, 0x7e1c54bd,
0xf01144f9, 0xd2240eb1, 0x9675b3fd, 0xa3ac3755, 0xd47c27af,
0x51c85f4d, 0x56907596, 0xa5bb15e6, 0x580304f0, 0xca042cf1,
0x011a37ea, 0x8dbfaadb, 0x35ba3e4a, 0x3526ffa0, 0xc37b4d09,
0xbc306ed9, 0x98a52666, 0x5648f725, 0xff5e569d, 0x0ced63d0,
0x7c63b2cf, 0x700b45e1, 0xd5ea50f1, 0x85a92872, 0xaf1fbda7,
0xd4234870, 0xa7870bf3, 0x2d3b4d79, 0x42e04198, 0x0cd0ede7,
0x26470db8, 0xf881814c, 0x474d6ad7, 0x7c0c5e5c, 0xd1231959,
0x381b7298, 0xf5d2f4db, 0xab838653, 0x6e2f1e23, 0x83719c9e,
0xbd91e046, 0x9a56456e, 0xdc39200c, 0x20c8c571, 0x962bda1c,
0xe1e696ff, 0xb141ab08, 0x7cca89b9, 0x1a69e783, 0x02cc4843,
0xa2f7c579, 0x429ef47d, 0x427b169c, 0x5ac9f049, 0xdd8f0f00,
```

```
0x5c8165bf);

SBOX II
$s2 = array(
0x1f201094, 0xef0ba75b, 0x69e3cf7e, 0x393f4380, 0xfe61cf7a,
0xeec5207a, 0x55889c94, 0x72fc0651, 0xada7ef79, 0x4e1d7235,
0xd55a63ce, 0xde0436ba, 0x99c430ef, 0x5f0c0794, 0x18dcdb7d,
0xa1d6eff3, 0xa0b52f7b, 0x59e83605, 0xee15b094, 0xe9ffd909,
0xdc440086, 0xef944459, 0xba83ccb3, 0xe0c3cdfb, 0xd1da4181,
0x3b092ab1, 0xf997f1c1, 0xa5e6cf7b, 0x01420ddb, 0xe4e7ef5b,
0x25a1ff41, 0xe180f806, 0x1fc41080, 0x179bee7a, 0xd37ac6a9,
0xfe5830a4, 0x98de8b7f, 0x77e83f4e, 0x79929269, 0x24fa9f7b,
0xe113c85b, 0xacc40083, 0xd7503525, 0xf7ea615f, 0x62143154,
0x0d554b63, 0x5d681121, 0xc866c359, 0x3d63cf73, 0xcee234c0,
0xd4d87e87, 0x5c672b21, 0x071f6181, 0x39f7627f, 0x361e3084,
0xe4eb573b, 0x602f64a4, 0xd63acd9c, 0x1bbc4635, 0x9e81032d,
0x2701f50c, 0x99847ab4, 0xa0e3df79, 0xba6cf38c, 0x10843094,
0x2537a95e, 0xf46f6ffe, 0xa1ff3b1f, 0x208cfb6a, 0x8f458c74,
0xd9e0a227, 0x4ec73a34, 0xfc884f69, 0x3e4de8df, 0xef0e0088,
0x3559648d, 0x8a45388c, 0x1d804366, 0x721d9bfd, 0xa58684bb,
0xe8256333, 0x844e8212, 0x128d8098, 0xfed33fb4, 0xce280ae1,
0x27e19ba5, 0xd5a6c252, 0xe49754bd, 0xc5d655dd, 0xeb667064,
0x77840b4d, 0xa1b6a801, 0x84db26a9, 0xe0b56714, 0x21f043b7,
0xe5d05860, 0x54f03084, 0x066ff472, 0xa31aa153, 0xdadc4755,
0xb5625dbf, 0x68561be6, 0x83ca6b94, 0x2d6ed23b, 0xeccf01db,
0xa6d3d0ba, 0xb6803d5c, 0xaf77a709, 0x33b4a34c, 0x397bc8d6,
0x5ee22b95, 0x5f0e5304, 0x81ed6f61, 0x20e74364, 0xb45e1378,
0xde18639b, 0x881ca122, 0xb96726d1, 0x8049a7e8, 0x22b7da7b,
0x5e552d25, 0x5272d237, 0x79d2951c, 0xc60d894c, 0x488cb402,
0x1ba4fe5b, 0xa4b09f6b, 0x1ca815cf, 0xa20c3005, 0x8871df63,
0xb9de2fcb, 0x0cc6c9e9, 0x0beeff53, 0xe3214517, 0xb4542835,
0x9f63293c, 0xee41e729, 0x6e1d2d7c, 0x50045286, 0x1e6685f3,
0xf33401c6, 0x30a22c95, 0x31a70850, 0x60930f13, 0x73f98417,
0xa1269859, 0xec645c44, 0x52c877a9, 0xcdff33a6, 0xa02b1741,
0x7cbad9a2, 0x2180036f, 0x50d99c08, 0xcb3f4861, 0xc26bd765,
0x64a3f6ab, 0x80342676, 0x25a75e7b, 0xe4e6d1fc, 0x20c710e6,
0xcdf0b680, 0x17844d3b, 0x31eef84d, 0x7e0824e4, 0x2ccb49eb,
0x846a3bae, 0x8ff77888, 0xee5d60f6, 0x7af75673, 0x2fdd5cdb,
0xa11631c1, 0x30f66f43, 0xb3faec54, 0x157fd7fa, 0xef8579cc,
0xd152de58, 0xdb2ffd5e, 0x8f32ce19, 0x306af97a, 0x02f03ef8,
0x99319ad5, 0xc242fa0f, 0xa7e3ebb0, 0xc68e4906, 0xb8da230c,
0x80823028, 0xdcdef3c8, 0xd35fb171, 0x088a1bc8, 0xbec0c560,
0x61a3c9e8, 0xbca8f54d, 0xc72feffa, 0x22822e99, 0x82c570b4,
0xd8d94e89, 0x8b1c34bc, 0x301e16e6, 0x273be979, 0xb0ffeaa6,
0x61d9b8c6, 0x00b24869, 0xb7ffce3f, 0x08dc283b, 0x43daf65a,
0xf7e19798, 0x7619b72f, 0x8f1c9ba4, 0xdc8637a0, 0x16a7d3b1,
0x9fc393b7, 0xa7136eeb, 0xc6bcc63e, 0x1a513742, 0xef6828bc,
0x520365d6, 0x2d6a77ab, 0x3527ed4b, 0x821fd216, 0x095c6e2e,
0xdb92f2fb, 0x5eea29cb, 0x145892f5, 0x91584f7f, 0x5483697b,
0x2667a8cc, 0x85196048, 0x8c4bacea, 0x833860d4, 0x0d23e0f9,
0x6c387e8a, 0x0ae6d249, 0xb284600c, 0xd835731d, 0xdcb1c647,
0xac4c56ea, 0x3ebd81b3, 0x230eabb0, 0x6438bc87, 0xf0b5b1fa,
0x8f5ea2b3, 0xfc184642, 0x0a036b7a, 0x4fb089bd, 0x649da589,
0xa345415e, 0x5c038323, 0x3e5d3bb9, 0x43d79572, 0x7e6dd07c,
0x06dfdf1e, 0x6c6cc4ef, 0x7160a539, 0x73bfbe70, 0x83877605,
0x4523ecf1);
```

```
SBOX III
$s3 = array(
0x8defc240, 0x25fa5d9f, 0xeb903dbf, 0xe810c907, 0x47607fff,
0x369fe44b, 0x8c1fc644, 0xaececa90, 0xbeb1f9bf, 0xeefbcaea,
0xe8cf1950, 0x51df07ae, 0x920e8806, 0xf0ad0548, 0xe13c8d83,
0x927010d5, 0x11107d9f, 0x07647db9, 0xb2e3e4d4, 0x3d4f285e,
0xb9afa820, 0xfade82e0, 0xa067268b, 0x8272792e, 0x553fb2c0,
0x489ae22b, 0xd4ef9794, 0x125e3fbc, 0x21fffcee, 0x825b1bfd,
0x9255c5ed, 0x1257a240, 0x4e1a8302, 0xbae07fff, 0x528246e7,
0x8e57140e, 0x3373f7bf, 0x8c9f8188, 0xa6fc4ee8, 0xc982b5a5,
0xa8c01db7, 0x579fc264, 0x67094f31, 0xf2bd3f5f, 0x40fff7c1,
0x1fb78dfc, 0x8e6bd2c1, 0x437be59b, 0x99b03dbf, 0xb5dbc64b,
0x638dc0e6, 0x55819d99, 0xa197c81c, 0x4a012d6e, 0xc5884a28,
0xccc36f71, 0xb843c213, 0x6c0743f1, 0x8309893c, 0x0feddd5f,
0x2f7fe850, 0xd7c07f7e, 0x02507fbf, 0x5afb9a04, 0xa747d2d0,
0x1651192e, 0xaf70bf3e, 0x58c31380, 0x5f98302e, 0x727cc3c4,
0x0a0fb402, 0x0f7fef82, 0x8c96fdad, 0x5d2c2aae, 0x8ee99a49,
0x50da88b8, 0x8427f4a0, 0x1eac5790, 0x796fb449, 0x8252dc15,
0xefbd7d9b, 0xa672597d, 0xada840d8, 0x45f54504, 0xfa5d7403,
0xe83ec305, 0x4f91751a, 0x925669c2, 0x23efe941, 0xa903f12e,
0x60270df2, 0x0276e4b6, 0x94fd6574, 0x927985b2, 0x8276dbcb,
0x02778176, 0xf8af918d, 0x4e48f79e, 0x8f616ddf, 0xe29d840e,
0x842f7d83, 0x340ce5c8, 0x96bbb682, 0x93b4b148, 0xef303cab,
0x984faf28, 0x779faf9b, 0x92dc560d, 0x224d1e20, 0x8437aa88,
0x7d29dc96, 0x2756d3dc, 0x8b907cee, 0xb51fd240, 0xe7c07ce3,
0xe566b4a1, 0xc3e9615e, 0x3cf8209d, 0x6094d1e3, 0xcd9ca341,
0x5c76460e, 0x00ea983b, 0xd4d67881, 0xfd47572c, 0xf76cedd9,
0xbda8229c, 0x127dadaa, 0x438a074e, 0x1f97c090, 0x081bdb8a,
0x93a07ebe, 0xb938ca15, 0x97b03cff, 0x3dc2c0f8, 0x8d1ab2ec,
0x64380e51, 0x68cc7bfb, 0xd90f2788, 0x12490181, 0x5de5ffd4,
0xdd7ef86a, 0x76a2e214, 0xb9a40368, 0x925d958f, 0x4b39fffa,
0xba39aee9, 0xa4ffd30b, 0xfaf7933b, 0x6d498623, 0x193cbcfa,
0x27627545, 0x825cf47a, 0x61bd8ba0, 0xd11e42d1, 0xcead04f4,
0x127ea392, 0x10428db7, 0x8272a972, 0x9270c4a8, 0x127de50b,
0x285ba1c8, 0x3c62f44f, 0x35c0eaa5, 0xe805d231, 0x428929fb,
0xb4fcdf82, 0x4fb66a53, 0x0e7dc15b, 0x1f081fab, 0x108618ae,
0xfcfd086d, 0xf9ff2889, 0x694bcc11, 0x236a5cae, 0x12deca4d,
0x2c3f8cc5, 0xd2d02dfe, 0xf8ef5896, 0xe4cf52da, 0x95155b67,
0x494a488c, 0xb9b6a80c, 0x5c8f82bc, 0x89d36b45, 0x3a609437,
0xec00c9a9, 0x44715253, 0x0a874b49, 0xd773bc40, 0x7c34671c,
0x02717ef6, 0x4feb5536, 0xa2d02fff, 0xd2bf60c4, 0xd43f03c0,
0x50b4ef6d, 0x07478cd1, 0x006e1888, 0xa2e53f55, 0xb9e6d4bc,
0xa2048016, 0x97573833, 0xd7207d67, 0xde0f8f3d, 0x72f87b33,
0xabcc4f33, 0x7688c55d, 0x7b00a6b0, 0x947b0001, 0x570075d2,
0xf9bb88f8, 0x8942019e, 0x4264a5ff, 0x856302e0, 0x72dbd92b,
0xee971b69, 0x6ea22fde, 0x5f08ae2b, 0xaf7a616d, 0xe5c98767,
0xcf1febd2, 0x61efc8c2, 0xf1ac2571, 0xcc8239c2, 0x67214cb8,
0xb1e583d1, 0xb7dc3e62, 0x7f10bdce, 0xf90a5c38, 0x0ff0443d,
0x606e6dc6, 0x60543a49, 0x5727c148, 0x2be98a1d, 0x8ab41738,
0x20e1be24, 0xaf96da0f, 0x68458425, 0x99833be5, 0x600d457d,
0x282f9350, 0x8334b362, 0xd91d1120, 0x2b6d8da0, 0x642b1e31,
0x9c305a00, 0x52bce688, 0x1b03588a, 0xf7baefd5, 0x4142ed9c,
0xa4315c11, 0x83323ec5, 0xdfef4636, 0xa133c501, 0xe9d3531c,
0xee353783);
```

```
SBOX IV
$s4 = array(
0x9db30420, 0x1fb6e9de, 0xa7be7bef, 0xd273a298, 0x4a4f7bdb,
0x64ad8c57, 0x85510443, 0xfa020ed1, 0x7e287aff, 0xe60fb663,
0x095f35a1, 0x79ebf120, 0xfd059d43, 0x6497b7b1, 0xf3641f63,
0x241e4adf, 0x28147f5f, 0x4fa2b8cd, 0xc9430040, 0x0cc32220,
0xfdd30b30, 0xc0a5374f, 0x1d2d00d9, 0x24147b15, 0xee4d111a,
0x0fca5167, 0x71ff904c, 0x2d195ffe, 0x1a05645f, 0x0c13fefe,
0x081b08ca, 0x05170121, 0x80530100, 0xe83e5efe, 0xac9af4f8,
0x7fe72701, 0xd2b8ee5f, 0x06df4261, 0xbb9e9b8a, 0x7293ea25,
0xce84ffdf, 0xf5718801, 0x3dd64b04, 0xa26f263b, 0x7ed48400,
0x547eebe6, 0x446d4ca0, 0x6cf3d6f5, 0x2649abdf, 0xaea0c7f5,
0x36338cc1, 0x503f7e93, 0xd3772061, 0x11b638e1, 0x72500e03,
0xf80eb2bb, 0xabe0502e, 0xec8d77de, 0x57971e81, 0xe14f6746,
0xc9335400, 0x6920318f, 0x081dbb99, 0xffc304a5, 0x4d351805,
0x7f3d5ce3, 0xa6c866c6, 0x5d5bcca9, 0xdaec6fea, 0x9f926f91,
0x9f46222f, 0x3991467d, 0xa5bf6d8e, 0x1143c44f, 0x43958302,
0xd0214eeb, 0x022083b8, 0x3fb6180c, 0x18f8931e, 0x281658e6,
0x26486e3e, 0x8bd78a70, 0x7477e4c1, 0xb506e07c, 0xf32d0a25,
0x79098b02, 0xe4eabb81, 0x28123b23, 0x69dead38, 0x1574ca16,
0xdf871b62, 0x211c40b7, 0xa51a9ef9, 0x0014377b, 0x041e8ac8,
0x09114003, 0xbd59e4d2, 0xe3d156d5, 0x4fe876d5, 0x2f91a340,
0x557be8de, 0x00eae4a7, 0x0ce5c2ec, 0x4db4bba6, 0xe756bdff,
0xdd3369ac, 0xec17b035, 0x06572327, 0x99afc8b0, 0x56c8c391,
0x6b65811c, 0x5e146119, 0x6e85cb75, 0xbe07c002, 0xc2325577,
0x893ff4ec, 0x5bbfc92d, 0xd0ec3b25, 0xb7801ab7, 0x8d6d3b24,
0x20c763ef, 0xc366a5fc, 0x9c382880, 0x0ace3205, 0xaac9548a,
0xeca1d7c7, 0x041afa32, 0x1d16625a, 0x6701902c, 0x9b757a54,
0x31d477f7, 0x9126b031, 0x36cc6fdb, 0xc70b8b46, 0xd9e66a48,
0x56e55a79, 0x026a4ceb, 0x52437eff, 0x2f8f76b4, 0x0df980a5,
0x8674cde3, 0xedda04eb, 0x17a9be04, 0x2c18f4df, 0xb7747f9d,
0xab2af7b4, 0xefc34d20, 0x2e096b7c, 0x1741a254, 0xe5b6a035,
0x213d42f6, 0x2c1c7c26, 0x61c2f50f, 0x6552daf9, 0xd2c231f8,
0x25130f69, 0xd8167fa2, 0x0418f2c8, 0x001a96a6, 0x0d1526ab,
0x63315c21, 0x5e0a72ec, 0x49bafefd, 0x187908d9, 0x8d0dbd86,
0x311170a7, 0x3e9b640c, 0xcc3e10d7, 0xd5cad3b6, 0x0caec388,
0xf73001e1, 0x6c728aff, 0x71eae2a1, 0x1f9af36e, 0xcfcbd12f,
0xc1de8417, 0xac07be6b, 0xcb44a1d8, 0x8b9b0f56, 0x013988c3,
0xb1c52fca, 0xb4be31cd, 0xd8782806, 0x12a3a4e2, 0x6f7de532,
0x58fd7eb6, 0xd01ee900, 0x24adffc2, 0xf4990fc5, 0x9711aac5,
0x001d7b95, 0x82e5e7d2, 0x109873f6, 0x00613096, 0xc32d9521,
0xada121ff, 0x29908415, 0x7fbb977f, 0xaf9eb3db, 0x29c9ed2a,
0x5ce2a465, 0xa730f32c, 0xd0aa3fe8, 0x8a5cc091, 0xd49e2ce7,
0x0ce454a9, 0xd60acd86, 0x015f1919, 0x77079103, 0xdea03af6,
0x78a8565e, 0xdee356df, 0x21f05cbe, 0x8b75e387, 0xb3c50651,
0xb8a5c3ef, 0xd8eeb6d2, 0xe523be77, 0xc2154529, 0x2f69efdf,
0xafe67afb, 0xf470c4b2, 0xf3e0eb5b, 0xd6cc9876, 0x39e4460c,
0x1fda8538, 0x1987832f, 0xca007367, 0xa99144f8, 0x296b299e,
0x492fc295, 0x9266beab, 0xb5676e69, 0x9bd3ddda, 0xdf7e052f,
0xdb25701c, 0x1b5e51ee, 0xf65324e6, 0x6afce36c, 0x0316cc04,
0x8644213e, 0xb7dc59d0, 0x7965291f, 0xccd6fd43, 0x41823979,
0x932bcdf6, 0xb657c34d, 0x4edfd282, 0x7ae5290c, 0x3cb9536b,
0x851e20fe, 0x9833557e, 0x13ecf0b0, 0xd3ffb372, 0x3f85c5c1,
0x0aef7ed2);
```

```
SBOX V
$s5 = array(
0x7ec90c04, 0x2c6e74b9, 0x9b0e66df, 0xa6337911, 0xb86a7fff,
0x1dd358f5, 0x44dd9d44, 0x1731167f, 0x08fbf1fa, 0xe7f511cc,
0xd2051b00, 0x735aba00, 0x2ab722d8, 0x386381cb, 0xacf6243a,
0x69befd7a, 0xe6a2e77f, 0xf0c720cd, 0xc4494816, 0xccf5c180,
0x38851640, 0x15b0a848, 0xe68b18cb, 0x4caadeff, 0x5f480a01,
0x0412b2aa, 0x259814fc, 0x41d0efe2, 0x4e40b48d, 0x248eb6fb,
0x8dba1cfe, 0x41a99b02, 0x1a550a04, 0xba8f65cb, 0x7251f4e7,
0x95a51725, 0xc106ecd7, 0x97a5980a, 0xc539b9aa, 0x4d79fe6a,
0xf2f3f763, 0x68af8040, 0xed0c9e56, 0x11b4958b, 0xe1eb5a88,
0x8709e6b0, 0xd7e07156, 0x4e29fea7, 0x6366e52d, 0x02d1c000,
0xc4ac8e05, 0x9377f571, 0x0c05372a, 0x578535f2, 0x2261be02,
0xd642a0c9, 0xdf13a280, 0x74b55bd2, 0x682199c0, 0xd421e5ec,
0x53fb3ce8, 0xc8adedb3, 0x28a87fc9, 0x3d959981, 0x5c1ff900,
0xfe38d399, 0x0c4eff0b, 0x062407ea, 0xaa2f4fb1, 0x4fb96976,
0x90c79505, 0xb0a8a774, 0xef55a1ff, 0xe59ca2c2, 0xa6b62d27,
0xe66a4263, 0xdf65001f, 0x0ec50966, 0xdfdd55bc, 0x29de0655,
0x911e739a, 0x17af8975, 0x32c7911c, 0x89f89468, 0x0d01e980,
0x524755f4, 0x03b63cc9, 0x0cc844b2, 0xbcf3f0aa, 0x87ac36e9,
0xe53a7426, 0x01b3d82b, 0x1a9e7449, 0x64ee2d7e, 0xcddbb1da,
0x01c94910, 0xb868bf80, 0x0d26f3fd, 0x9342ede7, 0x04a5c284,
0x636737b6, 0x50f5b616, 0xf24766e3, 0x8eca36c1, 0x136e05db,
0xfef18391, 0xfb887a37, 0xd6e7f7d4, 0xc7fb7dc9, 0x3063fcdf,
0xb6f589de, 0xec2941da, 0x26e46695, 0xb7566419, 0xf654efc5,
0xd08d58b7, 0x48925401, 0xc1bacb7f, 0xe5ff550f, 0xb6083049,
0x5bb5d0e8, 0x87d72e5a, 0xab6a6ee1, 0x223a66ce, 0xc62bf3cd,
0x9e0885f9, 0x68cb3e47, 0x086c010f, 0xa21de820, 0xd18b69de,
0xf3f65777, 0xfa02c3f6, 0x407edac3, 0xcbb3d550, 0x1793084d,
0xb0d70eba, 0x0ab378d5, 0xd951fb0c, 0xded7da56, 0x4124bbe4,
0x94ca0b56, 0x0f5755d1, 0xe0e1e56e, 0x6184b5be, 0x580a249f,
0x94f74bc0, 0xe327888e, 0x9f7b5561, 0xc3dc0280, 0x05687715,
0x646c6bd7, 0x44904db3, 0x66b4f0a3, 0xc0f1648a, 0x697ed5af,
0x49e92ff6, 0x309e374f, 0x2cb6356a, 0x85808573, 0x4991f840,
0x76f0ae02, 0x083be84d, 0x28421c9a, 0x44489406, 0x736e4cb8,
0xc1092910, 0x8bc95fc6, 0x7d869cf4, 0x134f616f, 0x2e77118d,
0xb31b2be1, 0xaa90b472, 0x3ca5d717, 0x7d161bba, 0x9cad9010,
0xaf462ba2, 0x9fe459d2, 0x45d34559, 0xd9f2da13, 0xdbc65487,
0xf3e4f94e, 0x176d486f, 0x097c13ea, 0x631da5c7, 0x445f7382,
0x175683f4, 0xcdc66a97, 0x70be0288, 0xb3cdcf72, 0x6e5dd2f3,
0x20936079, 0x459b80a5, 0xbe60e2db, 0xa9c23101, 0xeba5315c,
0x224e42f2, 0x1c5c1572, 0xf6721b2c, 0x1ad2fff3, 0x8c25404e,
0x324ed72f, 0x4067b7fd, 0x0523138e, 0x5ca3bc78, 0xdc0fd66e,
0x75922283, 0x784d6b17, 0x58ebb16e, 0x44094f85, 0x3f481d87,
0xfcfeae7b, 0x77b5ff76, 0x8c2302bf, 0xaaf47556, 0x5f46b02a,
0x2b092801, 0x3d38f5f7, 0x0ca81f36, 0x52af4a8a, 0x66d5e7c0,
0xdf3b0874, 0x95055110, 0x1b5ad7a8, 0xf61ed5ad, 0x6cf6e479,
0x20758184, 0xd0cefa65, 0x88f7be58, 0x4a046826, 0x0ff6f8f3,
0xa09c7f70, 0x5346aba0, 0x5ce96c28, 0xe176eda3, 0x6bac307f,
0x376829d2, 0x85360fa9, 0x17e3fe2a, 0x24b79767, 0xf5a96b20,
0xd6cd2595, 0x68ff1ebf, 0x7555442c, 0xf19f06be, 0xf9e0659a,
0xeeb9491d, 0x34010718, 0xbb30cab8, 0xe822fe15, 0x88570983,
0x750e6249, 0xda627e55, 0x5e76ffa8, 0xb1534546, 0x6d47de08,
0xefe9e7d4);
```

```
SBOX VI
$s6 = array(
0xf6fa8f9d, 0x2cac6ce1, 0x4ca34867, 0xe2337f7c, 0x95db08e7,
0x016843b4, 0xeced5cbc, 0x325553ac, 0xbf9f0960, 0xdfa1e2ed,
0x83f0579d, 0x63ed86b9, 0x1ab6a6b8, 0xde5ebe39, 0xf38ff732,
0x8989b138, 0x33f14961, 0xc01937bd, 0xf506c6da, 0xe4625e7e,
0xa308ea99, 0x4e23e33c, 0x79cbd7cc, 0x48a14367, 0xa3149619,
0xfec94bd5, 0xa114174a, 0xeaa01866, 0xa084db2d, 0x09a8486f,
0xa888614a, 0x2900af98, 0x01665991, 0xe1992863, 0xc8f30c60,
0x2e78ef3c, 0xd0d51932, 0xcf0fec14, 0xf7ca07d2, 0xd0a82072,
0xfd41197e, 0x9305a6b0, 0xe86be3da, 0x74bed3cd, 0x372da53c,
0x4c7f4448, 0xdab5d440, 0x6dba0ec3, 0x083919a7, 0x9fbaeed9,
0x49dbcfb0, 0x4e670c53, 0x5c3d9c01, 0x64bdb941, 0x2c0e636a,
0xba7dd9cd, 0xea6f7388, 0xe70bc762, 0x35f29adb, 0x5c4cdd8d,
0xf0d48d8c, 0xb88153e2, 0x08a19866, 0x1ae2eac8, 0x284caf89,
0xaa928223, 0x9334be53, 0x3b3a21bf, 0x16434be3, 0x9aea3906,
0xefe8c36e, 0xf890cdd9, 0x80226dae, 0xc340a4a3, 0xdf7e9c09,
0xa694a807, 0x5b7c5ecc, 0x221db3a6, 0x9a69a02f, 0x68818a54,
0xceb2296f, 0x53c0843a, 0xfe893655, 0x25bfe68a, 0xb4628abc,
0xcf222ebf, 0x25ac6f48, 0xa9a99387, 0x53bddb65, 0xe76ffbe7,
0xe967fd78, 0x0ba93563, 0x8e342bc1, 0xe8a11be9, 0x4980740d,
0xc8087dfc, 0x8de4bf99, 0xa11101a0, 0x7fd37975, 0xda5a26c0,
0xe81f994f, 0x9528cd89, 0xfd339fed, 0xb87834bf, 0x5f04456d,
0x22258698, 0xc9c4c83b, 0x2dc156be, 0x4f628daa, 0x57f55ec5,
0xe2220abe, 0xd2916ebf, 0x4ec75b95, 0x24f2c3c0, 0x42d15d99,
0xcd0d7fa0, 0x7b6e27ff, 0xa8dc8af0, 0x7345c106, 0xf41e232f,
0x35162386, 0xe6ea8926, 0x3333b094, 0x157ec6f2, 0x372b74af,
0x692573e4, 0xe9a9d848, 0xf3160289, 0x3a62ef1d, 0xa787e238,
0xf3a5f676, 0x74364853, 0x20951063, 0x4576698d, 0xb6fad407,
0x592af950, 0x36f73523, 0x4cfb6e87, 0x7da4cec0, 0x6c152daa,
0xcb0396a8, 0xc50dfe5d, 0xfcd707ab, 0x0921c42f, 0x89dff0bb,
0x5fe2be78, 0x448f4f33, 0x754613c9, 0x2b05d08d, 0x48b9d585,
0xdc049441, 0xc8098f9b, 0x7dede786, 0xc39a3373, 0x42410005,
0x6a091751, 0x0ef3c8a6, 0x890072d6, 0x28207682, 0xa9a9f7be,
0xbf32679d, 0xd45b5b75, 0xb353fd00, 0xcbb0e358, 0x830f220a,
0x1f8fb214, 0xd372cf08, 0xcc3c4a13, 0x8cf63166, 0x061c87be,
0x88c98f88, 0x6062e397, 0x47cf8e7a, 0xb6c85283, 0x3cc2acfb,
0x3fc06976, 0x4e8f0252, 0x64d8314d, 0xda3870e3, 0x1e665459,
0xc10908f0, 0x513021a5, 0x6c5b68b7, 0x822f8aa0, 0x3007cd3e,
0x74719eef, 0xdc872681, 0x073340d4, 0x7e432fd9, 0x0c5ec241,
0x8809286c, 0xf592d891, 0x08a930f6, 0x957ef305, 0xb7fbffbd,
0xc266e96f, 0x6fe4ac98, 0xb173ecc0, 0xbc60b42a, 0x953498da,
0xfba1ae12, 0x2d4bd736, 0x0f25faab, 0xa4f3fceb, 0xe2969123,
0x257f0c3d, 0x9348af49, 0x361400bc, 0xe8816f4a, 0x3814f200,
0xa3f94043, 0x9c7a54c2, 0xbc704f57, 0xda41e7f9, 0xc25ad33a,
0x54f4a084, 0xb17f5505, 0x59357cbe, 0xedbd15c8, 0x7f97c5ab,
0xba5ac7b5, 0xb6f6deaf, 0x3a479c3a, 0x5302da25, 0x653d7e6a,
0x54268d49, 0x51a477ea, 0x5017d55b, 0xd7d25d88, 0x44136c76,
0x0404a8c8, 0xb8e5a121, 0xb81a928a, 0x60ed5869, 0x97c55b96,
0xeaec991b, 0x29935913, 0x01fdb7f1, 0x088e8dfa, 0x9ab6f6f5,
0x3b4cbf9f, 0x4a5de3ab, 0xe6051d35, 0xa0e1d855, 0xd36b4cf1,
0xf544edeb, 0xb0e93524, 0xbebb8fbd, 0xa2d762cf, 0x49c92f54,
0x38b5f331, 0x7128a454, 0x48392905, 0xa65b1db8, 0x851c97bd,
0xd675cf2f);
```

```
SBOX VII
$s7 = array(
0x85e04019, 0x332bf567, 0x662dbfff, 0xcfc65693, 0x2a8d7f6f,
0xab9bc912, 0xde6008a1, 0x2028da1f, 0x0227bce7, 0x4d642916,
0x18fac300, 0x50f18b82, 0x2cb2cb11, 0xb232e75c, 0x4b3695f2,
0xb28707de, 0xa05fbcf6, 0xcd4181e9, 0xe150210c, 0xe24ef1bd,
0xb168c381, 0xfde4e789, 0x5c79b0d8, 0x1e8bfd43, 0x4d495001,
0x38be4341, 0x913cee1d, 0x92a79c3f, 0x089766be, 0xbaeeadf4,
0x1286becf, 0xb6eacb19, 0x2660c200, 0x7565bde4, 0x64241f7a,
0x8248dca9, 0xc3b3ad66, 0x28136086, 0x0bd8dfa8, 0x356d1cf2,
0x107789be, 0xb3b2e9ce, 0x0502aa8f, 0x0bc0351e, 0x166bf52a,
0xeb12ff82, 0xe3486911, 0xd34d7516, 0x4e7b3aff, 0x5f43671b,
0x9cf6e037, 0x4981ac83, 0x334266ce, 0x8c9341b7, 0xd0d854c0,
0xcb3a6c88, 0x47bc2829, 0x4725ba37, 0xa66ad22b, 0x7ad61f1e,
0x0c5cbafa, 0x4437f107, 0xb6e79962, 0x42d2d816, 0x0a961288,
0xe1a5c06e, 0x13749e67, 0x72fc081a, 0xb1d139f7, 0xf9583745,
0xcf19df58, 0xbec3f756, 0xc06eba30, 0x07211b24, 0x45c28829,
0xc95e317f, 0xbc8ec511, 0x38bc46e9, 0xc6e6fa14, 0xbae8584a,
0xad4ebc46, 0x468f508b, 0x7829435f, 0xf124183b, 0x821dba9f,
0xaff60ff4, 0xea2c4e6d, 0x16e39264, 0x92544a8b, 0x009b4fc3,
0xaba68ced, 0x9ac96f78, 0x06a5b79a, 0xb2856e6e, 0x1aec3ca9,
0xbe838688, 0x0e0804e9, 0x55f1be56, 0xe7e5363b, 0xb3a1f25d,
0xf7debb85, 0x61fe033c, 0x16746233, 0x3c034c28, 0xda6d0c74,
0x79aac56c, 0x3ce4e1ad, 0x51f0c802, 0x98f8f35a, 0x1626a49f,
0xeed82b29, 0x1d382fe3, 0x0c4fb99a, 0xbb325778, 0x3ec6d97b,
0x6e77a6a9, 0xcb658b5c, 0xd45230c7, 0x2bd1408b, 0x60c03eb7,
0xb9068d78, 0xa33754f4, 0xf430c87d, 0xc8a71302, 0xb96d8c32,
0xebd4e7be, 0xbe8b9d2d, 0x7979fb06, 0xe7225308, 0x8b75cf77,
0x11ef8da4, 0xe083c858, 0x8d6b786f, 0x5a6317a6, 0xfa5cf7a0,
0x5dda0033, 0xf28ebfb0, 0xf5b9c310, 0xa0eac280, 0x08b9767a,
0xa3d9d2b0, 0x79d34217, 0x021a718d, 0x9ac6336a, 0x2711fd60,
0x438050e3, 0x069908a8, 0x3d7fedc4, 0x826d2bef, 0x4eeb8476,
0x488dcf25, 0x36c9d566, 0x28e74e41, 0xc2610aca, 0x3d49a9cf,
0xbae3b9df, 0xb65f8de6, 0x92aeaf64, 0x3ac7d5e6, 0x9ea80509,
0xf22b017d, 0xa4173f70, 0xdd1e16c3, 0x15e0d7f9, 0x50b1b887,
0x2b9f4fd5, 0x625aba82, 0x6a017962, 0x2ec01b9c, 0x15488aa9,
0xd716e740, 0x40055a2c, 0x93d29a22, 0xe32dbf9a, 0x058745b9,
0x3453dc1e, 0xd699296e, 0x496cff6f, 0x1c9f4986, 0xdfe2ed07,
0xb87242d1, 0x19de7eae, 0x053e561a, 0x15ad6f8c, 0x66626c1c,
0x7154c24c, 0xea082b2a, 0x93eb2939, 0x17dcb0f0, 0x58d4f2ae,
0x9ea294fb, 0x52cf564c, 0x9883fe66, 0x2ec40581, 0x763953c3,
0x01d6692e, 0xd3a0c108, 0xa1e7160e, 0xe4f2dfa6, 0x693ed285,
0x74904698, 0x4c2b0edd, 0x4f757656, 0x5d393378, 0xa132234f,
0x3d321c5d, 0xc3f5e194, 0x4b269301, 0xc79f022f, 0x3c997e7e,
0x5e4f9504, 0x3ffafbbd, 0x76f7ad0e, 0x296693f4, 0x3d1fce6f,
0xc61e45be, 0xd3b5ab34, 0xf72bf9b7, 0x1b0434c0, 0x4e72b567,
0x5592a33d, 0xb5229301, 0xcfd2a87f, 0x60aeb767, 0x1814386b,
0x30bcc33d, 0x38a0c07d, 0xfd1606f2, 0xc363519b, 0x589dd390,
0x5479f8e6, 0x1cb8d647, 0x97fd61a9, 0xea7759f4, 0x2d57539d,
0x569a58cf, 0xe84e63ad, 0x462e1b78, 0x6580f87e, 0xf3817914,
0x91da55f4, 0x40a230f3, 0xd1988f35, 0xb6e318d2, 0x3ffa50bc,
0x3d40f021, 0xc3c0bdae, 0x4958c24c, 0x518f36b2, 0x84b1d370,
0x0fedce83, 0x878ddada, 0xf2a279c7, 0x94e01be8, 0x90716f4b,
0x954b8aa3);
```

```
SBOX VIII
$s8 = array(
0xe216300d, 0xbbddfffc, 0xa7ebdabd, 0x35648095, 0x7789f8b7,
0xe6c1121b, 0x0e241600, 0x052ce8b5, 0x11a9cfb0, 0xe5952f11,
0xece7990a, 0x9386d174, 0x2a42931c, 0x76e38111, 0xb12def3a,
0x37ddddfc, 0xde9adeb1, 0x0a0cc32c, 0xbe197029, 0x84a00940,
0xbb243a0f, 0xb4d137cf, 0xb44e79f0, 0x049eedfd, 0x0b15a15d,
0x480d3168, 0x8bbbde5a, 0x669ded42, 0xc7ece831, 0x3f8f95e7,
0x72df191b, 0x7580330d, 0x94074251, 0x5c7dcdfa, 0xabbe6d63,
0xaa402164, 0xb301d40a, 0x02e7d1ca, 0x53571dae, 0x7a3182a2,
0x12a8ddec, 0xfdaa335d, 0x176f43e8, 0x71fb46d4, 0x38129022,
0xce949ad4, 0xb84769ad, 0x965bd862, 0x82f3d055, 0x66fb9767,
0x15b80b4e, 0x1d5b47a0, 0x4cfde06f, 0xc28ec4b8, 0x57e8726e,
0x647a78fc, 0x99865d44, 0x608bd593, 0x6c200e03, 0x39dc5ff6,
0x5d0b00a3, 0xae63aff2, 0x7e8bd632, 0x70108c0c, 0xbbd35049,
0x2998df04, 0x980cf42a, 0x9b6df491, 0x9e7edd53, 0x06918548,
0x58cb7e07, 0x3b74ef2e, 0x522fffb1, 0xd24708cc, 0x1c7e27cd,
0xa4eb215b, 0x3cf1d2e2, 0x19b47a38, 0x424f7618, 0x35856039,
0x9d17dee7, 0x27eb35e6, 0xc9aff67b, 0x36baf5b8, 0x09c467cd,
0xc18910b1, 0xe11dbf7b, 0x06cd1af8, 0x7170c608, 0x2d5e3354,
0xd4de495a, 0x64c6d006, 0xbcc0c62c, 0x3dd00db3, 0x708f8f34,
0x77d51b42, 0x264f620f, 0x24b8d2bf, 0x15c1b79e, 0x46a52564,
0xf8d7e54e, 0x3e378160, 0x7895cda5, 0x859c15a5, 0xe6459788,
0xc37bc75f, 0xdb07ba0c, 0x0676a3ab, 0x7f229b1e, 0x31842e7b,
0x24259fd7, 0xf8bef472, 0x835ffcb8, 0x6df4c1f2, 0x96f5b195,
0xfd0af0fc, 0xb0fe134c, 0xe2506d3d, 0x4f9b12ea, 0xf215f225,
0xa223736f, 0x9fb4c428, 0x25d04979, 0x34c713f8, 0xc4618187,
0xea7a6e98, 0x7cd16efc, 0x1436876c, 0xf1544107, 0xbedeee14,
0x56e9af27, 0xa04aa441, 0x3cf7c899, 0x92ecbae6, 0xdd67016d,
0x151682eb, 0xa842eedf, 0xfdba60b4, 0xf1907b75, 0x20e3030f,
0x24d8c29e, 0xe139673b, 0xefa63fb8, 0x71873054, 0xb6f2cf3b,
0x9f326442, 0xcb15a4cc, 0xb01a4504, 0xf1e47d8d, 0x844a1be5,
0xbae7dfdc, 0x42cbda70, 0xcd7dae0a, 0x57e85b7a, 0xd53f5af6,
0x20cf4d8c, 0xcea4d428, 0x79d130a4, 0x3486ebfb, 0x33d3cddc,
0x77853b53, 0x37effcb5, 0xc5068778, 0xe580b3e6, 0x4e68b8f4,
0xc5c8b37e, 0x0d809ea2, 0x398feb7c, 0x132a4f94, 0x43b7950e,
0x2fee7d1c, 0x223613bd, 0xdd06caa2, 0x37df932b, 0xc4248289,
0xacf3ebc3, 0x5715f6b7, 0xef3478dd, 0xf267616f, 0xc148cbe4,
0x9052815e, 0x5e410fab, 0xb48a2465, 0x2eda7fa4, 0xe87b40e4,
0xe98ea084, 0x5889e9e1, 0xefd390fc, 0xdd07d35b, 0xdb485694,
0x38d7e5b2, 0x57720101, 0x730edebc, 0x5b643113, 0x94917e4f,
0x503c2fba, 0x646f1282, 0x7523d24a, 0xe0779695, 0xf9c17a8f,
0x7a5b2121, 0xd187b896, 0x29263a4d, 0xba510cdf, 0x81f47c9f,
0xad1163ed, 0xea7b5965, 0x1a00726e, 0x11403092, 0x00da6d77,
0x4a0cdd61, 0xad1f4603, 0x605bdfb0, 0x9eedc364, 0x22ebe6a8,
0xcee7d28a, 0xa0e736a0, 0x5564a6b9, 0x10853209, 0xc7eb8f37,
0x2de705ca, 0x8951570f, 0xdf09822b, 0xbd691a6c, 0xaa12e4f2,
0x87451c0f, 0xe0f6a27a, 0x3ada4819, 0x4cf1764f, 0x0d771c2b,
0x67cdb156, 0x350d8384, 0x5938fa0f, 0x42399ef3, 0x36997b07,
0x0e84093d, 0x4aa93e61, 0x8360d87b, 0x1fa98b0c, 0x1149382c,
0xe97625a5, 0x0614d1b7, 0x0e25244b, 0x0c768347, 0x589e8d82,
0x0d2059d1, 0xa466bb1e, 0xf8da0a82, 0x04f19130, 0xba6e4ec0,
0x99265164, 0x1ee7230d, 0x50b2ad80, 0xeaee6801, 0x8db2a283,
0xea8bf59e);
```

LAMPIRAN 2

Tabel Pengujian waktu Enkripsi

percobaan 1

Nama *File*	Ukuran *File*	Waktu Proses Enkripsi (s)
50KB.txt	50KB	0.798
100KB.txt	100KB	1.582
150KB.txt	150KB	2.387
200KB.txt	200KB	3.212
250KB.txt	250KB	3.916
300KB.txt	300KB	5.026
350KB.txt	350KB	5.951
400KB.txt	400KB	6.657
450KB.txt	450KB	7.563
500KB.txt	500KB	8.324

percobaan 2

Nama *File*	Ukuran *File*	Waktu Proses Enkripsi (s)
50KB.txt	50KB	0.827
100KB.txt	100KB	1.619
150KB.txt	150KB	2.364
200KB.txt	200KB	3.147
250KB.txt	250KB	4.41
300KB.txt	300KB	5.01
350KB.txt	350KB	5.594
400KB.txt	400KB	6.46
450KB.txt	450KB	7.561
500KB.txt	500KB	8.507

percobaan 3

Nama *File*	Ukuran *File*	Waktu Proses Enkripsi (s)
50KB.txt	50KB	0.826
100KB.txt	100KB	1.756
150KB.txt	150KB	2.364
200KB.txt	200KB	3.103
250KB.txt	250KB	4.22
300KB.txt	300KB	5.165
350KB.txt	350KB	5.881
400KB.txt	400KB	6.784
450KB.txt	450KB	7.947
500KB.txt	500KB	8.84

percobaan 4

Nama *File*	Ukuran *File*	Waktu Proses Enkripsi (s)
50KB.txt	50KB	0.807
100KB.txt	100KB	1.612
150KB.txt	150KB	2.621
200KB.txt	200KB	3.274
250KB.txt	250KB	4.08
300KB.txt	300KB	5.327
350KB.txt	350KB	6.236
400KB.txt	400KB	7.013
450KB.txt	450KB	7.676
500KB.txt	500KB	8.173

percobaan 5

Nama *File*	Ukuran *File*	Waktu Proses Enkripsi (s)
50KB.txt	50KB	0.777
100KB.txt	100KB	1.774
150KB.txt	150KB	2.414
200KB.txt	200KB	3.267
250KB.txt	250KB	4.737
300KB.txt	300KB	5.341
350KB.txt	350KB	5.827
400KB.txt	400KB	6.649
450KB.txt	450KB	7.562
500KB.txt	500KB	8.214

LAMPIRAN 3

Tabel Pengujian waktu Dekripsi

percobaan 1

Nama *File*	Ukuran *File*	Waktu Proses Dekripsi(s)
50KB.txt	50KB	0.803
100KB.txt	100KB	1.585
150KB.txt	150KB	2.363
200KB.txt	200KB	3.163
250KB.txt	250KB	4.064
300KB.txt	300KB	4.766
350KB.txt	350KB	5.921
400KB.txt	400KB	6.544
450KB.txt	450KB	7.446
500KB.txt	500KB	8.286

percobaan 2

Nama *File*	Ukuran *File*	Waktu Proses Dekripsi(s)
50KB.txt	50KB	0.767
100KB.txt	100KB	1.605
150KB.txt	150KB	2.397
200KB.txt	200KB	3.328
250KB.txt	250KB	4.313
300KB.txt	300KB	4.922
350KB.txt	350KB	6.066
400KB.txt	400KB	6.806
450KB.txt	450KB	7.515
500KB.txt	500KB	8.668

percobaan 3

Nama *File*	Ukuran *File*	Waktu Proses Dekripsi(s)
50KB.txt	50KB	0.796
100KB.txt	100KB	1.672
150KB.txt	150KB	2.572
200KB.txt	200KB	3.539
250KB.txt	250KB	4.569
300KB.txt	300KB	5.213
350KB.txt	350KB	6.02
400KB.txt	400KB	7.114
450KB.txt	450KB	7.606
500KB.txt	500KB	8.331

percobaan 4

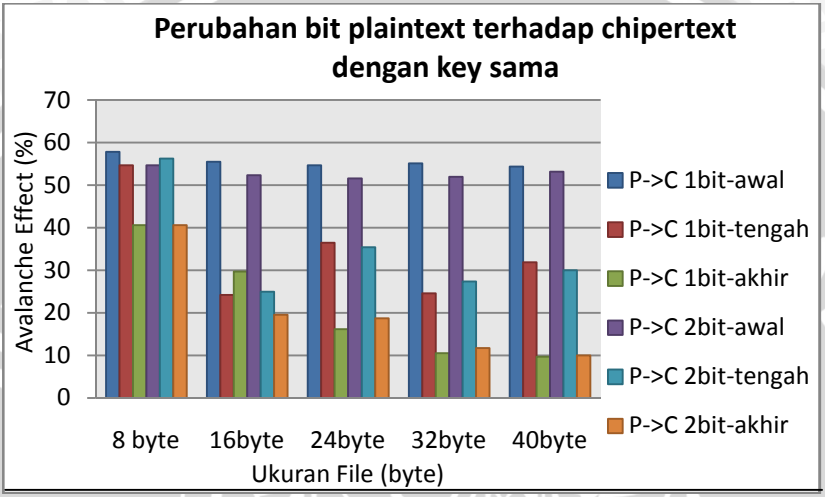
Nama *File*	Ukuran *File*	Waktu Proses Dekripsi(s)
50KB.txt	50KB	0.805
100KB.txt	100KB	1.604
150KB.txt	150KB	2.427
200KB.txt	200KB	3.258
250KB.txt	250KB	4.278
300KB.txt	300KB	5.444
350KB.txt	350KB	6.413
400KB.txt	400KB	6.833
450KB.txt	450KB	7.833
500KB.txt	500KB	8.334

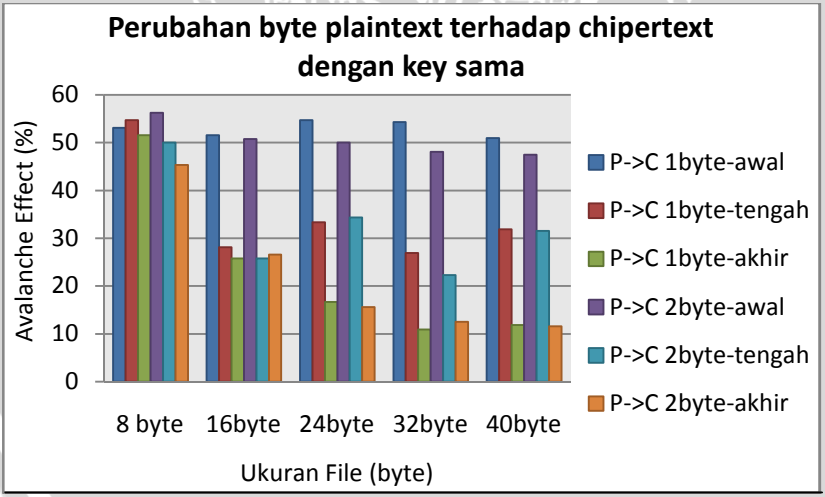
percobaan 5

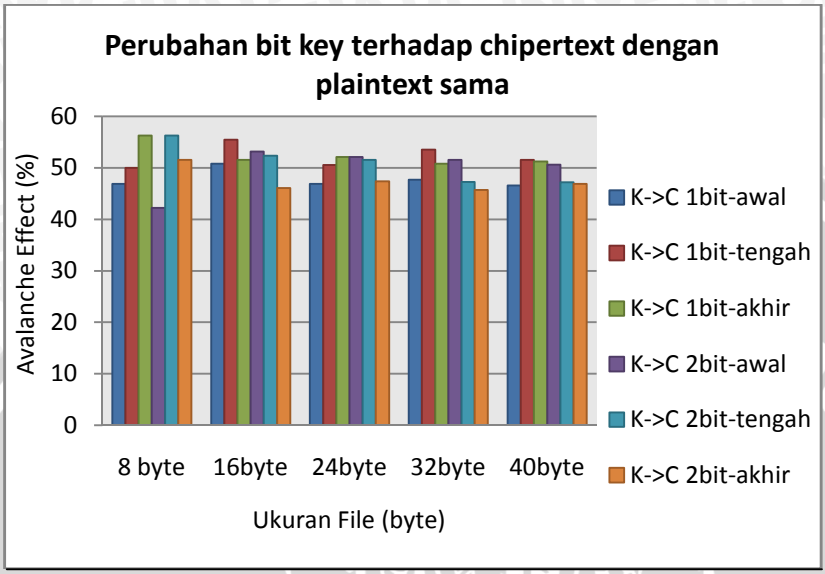
Nama *File*	Ukuran *File*	Waktu Proses Dekripsi(s)
50KB.txt	50KB	0.799
100KB.txt	100KB	1.719
150KB.txt	150KB	2.544
200KB.txt	200KB	3.255
250KB.txt	250KB	4.179
300KB.txt	300KB	4.833
350KB.txt	350KB	6.073
400KB.txt	400KB	7.032
450KB.txt	450KB	7.623
500KB.txt	500KB	8.637

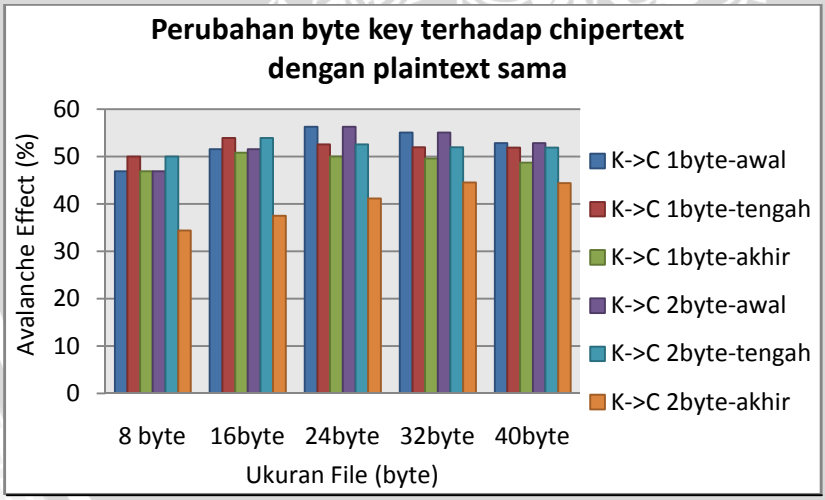
LAMPIRAN 4

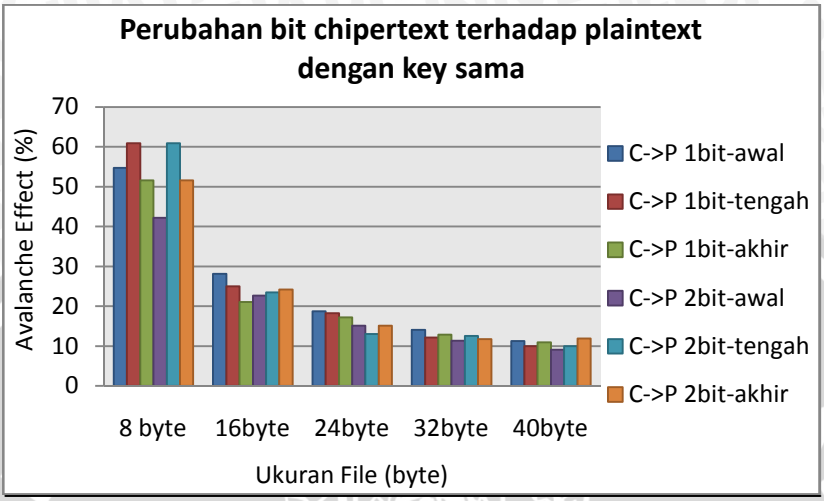
Grafik Pengujian *avalanche effect*

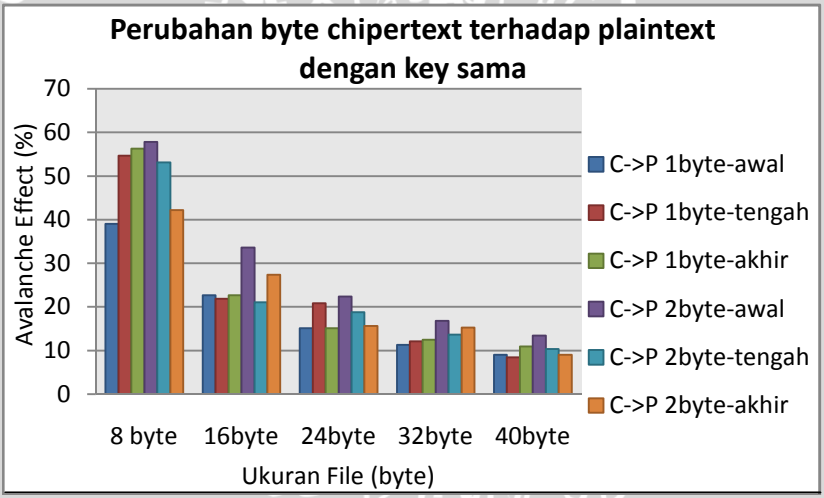





Perubahan bit key terhadap chipertext dengan plaintext sama
Avalanche Effect (%)
60
50
40
30
20
10
0
8 byte
16byte
24byte
32byte
40byte
Ukuran File (byte)
K->C 1bit-awal
K->C 1bit-tengah
K->C 1bit-akhir
K->C 2bit-awal
K->C 2bit-tengah
K->C 2bit-akhir


Perubahan byte key terhadap chipertext dengan plaintext sama
Avalanche Effect (%)
60
50
40
30
20
10
0
8 byte
16byte
24byte
32byte
40byte
Ukuran File (byte)
K->C 1byte-awal
K->C 1byte-tengah
K->C 1byte-akhir
K->C 2byte-awal
K->C 2byte-tengah
K->C 2byte-akhir


Perubahan bit chipertext terhadap plaintext dengan key sama
Avalanche Effect (%)
70
60
50
40
30
20
10
0
8 byte
16byte
24byte
32byte
40byte
Ukuran File (byte)
C->P 1bit-awal
C->P 1bit-tengah
C->P 1bit-akhir
C->P 2bit-awal
C->P 2bit-tengah
C->P 2bit-akhir


Perubahan byte chipertext terhadap plaintext dengan key sama
Avalanche Effect (%)
70
60
50
40
30
20
10
0
8 byte
16byte
24byte
32byte
40byte
Ukuran File (byte)
C->P 1byte-awal
C->P 1byte-tengah
C->P 1byte-akhir
C->P 2byte-awal
C->P 2byte-tengah
C->P 2byte-akhir