Perancangan Algoritma *Block Cipher* Menggunakan *Cryptographically Secure Pseudorandom Number Generator*

Gagas Praharsa Bahar - 13520016[1]
Rayhan Kinan Muhannad – 13520065[2]
Aira Thalca Avila Putra – 13520101[3]

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha no. 10 Bandung 40132, Indonesia
[1]13520016@std.stei.itb.ac.id
[2]13520065@std.stei.itb.ac.id
[3]13520101@std.stei.itb.ac.id

***Abstract*—Algoritma GKA adalah sebuah algoritma blok cipher yang terinspirasi dari block cipher umum di dunia seperti DES. Blok dan kunci dari algoritma GKA berukuran sama yaitu 128 bit. Algoritma GKA memanfaatkan jaringan Feistel 16 putaran dalam implementasinya. Fungsi putaran pada jaringan Feistel berisi jaringan substitusi-permutasi yang beroperasi dalam *byte*. Algoritma ini memiliki tingkat keamanan yang cukup baik karena menerapkan prinsip *diffusion* dan *confusion*.**

Keywords*—Kriptografi, *block cipher*, *modern cipher

I. PENDAHULUAN

Pada era informasi seperti sekarang ini, pertukaran informasi secara digital tentu saja menjadi sebuah kegiatan yang sangat penting bagi keberlangsungan hidup manusia. Di era digital seperti sekarang ini, proses pertukaran informasi yang disimbolkan sebagai deret angka atau huruf terjadi dengan sangat cepat. Namun, dengan pesatnya perkembangan infrastruktur pertukaran informasi, kemungkinan adanya kebocoran informasi ke pihak ketiga yang tidak diinginkan menjadi jauh lebih besar. Oleh karenanya, ilmu kriptografi hadir untuk mempersulit pihak yang tidak berkepentingan untuk mengakses sebuah informasi.

Block cipher adalah salah satu jenis *cipher* dimana plainteks dibagi menjadi blok-blok bit dengan panjang sama sebelum dijadikan input dari sebuah algoritma terkait. Dengan menggunakan *block cipher*, panjang blok cipherteks akan sama panjangnya dengan panjang blok plainteks. *Block cipher* juga memiliki beberapa mode operasi yang berbeda, yang mempengaruhi bagaimana masing-masing blok diproses.

Untuk mendesain sebuah *block cipher* yang kuat, diperlukan algoritma *cipher* yang kuat sebagai bagian inti dari *block cipher* itu sendiri. Terdapat beberapa prinsip serta teknik yang dapat digunakan untuk memperkuat *cipher* blok, seperti *confusion* dan *diffusion*, substitusi, permutasi, jaringan feistel, dan pembangkitan kunci dengan kuat. Terlebih lagi, pembangkitan skema substitusi serta permutasi dapat dibangkitkan dengan pembangkit angka acak yang *cryptographically secure* atau aman secara kriptografi.

II. DASAR TEORI

A. Data Encryption Standard

Data Encryption Standard (DES) adalah salah satu standar algoritma *symmetric-key block cipher* yang digunakan untuk mengenkripsi data elektronik. Algoritma DES dikembangkan pada tahun 1970 an oleh IBM dan diadopsi sebagai algoritma standar untuk melakukan enkripsi oleh pemerintahan Amerika Serikat pada tahun 1977 *sebagai Federal Information Processing Standard*. Algoritma DES menggunakan 56-bit *key* yang merupakan hasil *truncation* dari *initial key* sebesar 64-bit untuk melakukan enkripsi/dekripsi 64-bit *block* dari suatu data.

Terdapat beberapa tahap dalam proses enkripsi/dekripsi pada algoritma DES. Langkah pertama dalam algoritma DES adalah melakukan pembuangan seluruh bit ke-8 dalam setiap *byte* pada *external key* yang dimasukkan oleh pengguna. Hal ini akan menyebabkan panjang keseluruhan *external key* mengecil dari 64-bit menjadi 56-bit. Setelah *external key* dilakukan operasi *truncation*, maka langkah selanjutnya adalah melakukan *initial bit permutation* pada *plaintext block* yang dimasukkan oleh pengguna. Langkah berikutnya yang dilakukan adalah memasukkan *plaintext* ke dalam *Feistel Network* dengan jumlah pengulangan sebanyak 16 kali. Di dalam *Feistel Network*, *plaintext* yang dimasukkan akan dilakukan operasi *substitution*, *permutation*, *mixing*, dan *swap* dengan menggunakan *internal key* yang berbeda - beda pada setiap perulangan. Setelah dilakukan pengulangan sebanyak 16 kali pada *Feistel Network*, maka langkah terakhir yang dilakukan adalah melakukan *final permutation* yang dimana permutasi tersebut merupakan invers dari *initial permutation* untuk mengembalikan tatanan bit seperti pada awal proses enkripsi/dekripsi. Perbedaan antara proses enkripsi dengan proses dekripsi pada algoritma DES adalah pada proses dekripsi, urutan pengerjaan langkah dilakukan serta menggunakan urutan pembangkitan *internal key* pada perulangan Feistel Network dilakukan dari langkah terakhir ke langkah pertama (dieksekusi secara terbalik).

Di era modern ini, algoritma DES sudah menjadi tidak aman untuk melakukan enkripsi/dekripsi data sensitif. Beberapa *cryptanalyst* dari distributed.net dan *Electronic Frontier Foundation* berhasil memecahkan enkripsi algoritma DES dalam waktu 22 jam dan 15 menit. Hal ini dikarenakan algoritma DES hanya aman jika dilakukan kriptanalisis dengan menggunakan teknik *differential cryptanalysis*, tetapi tidak aman jika dilakukan dengan teknik *brute force*. Oleh karena itu, untuk mengatasi kelemahan oleh serangan *brute force* pada algoritma DES, dikembangkan suatu standar algoritma baru yang dinamakan dengan *Advanced Encryption Standard (*AES) dan *Triple* DES (3DES). Pada masa kini, penggunaan algoritma DES hanya diaplikasikan pada beberapa *legacy system* untuk melakukan proses enkripsi-dekripsi.

B. Feistel Network

Feistel Network atau *Feistel Cipher* adalah suatu teknik atau *framework* yang digunakan dalam melakukan *symmetric-key block cipher*. *Feistel Network* diciptakan oleh Horst Feistel pada tahun 1973 ketika beliau sedang bekerja pada IBM. *Feistel Network* dikembangkan dari konsep operasi enkripsi yang dilakukan secara berulang kali dengan menggunakan *subkey* yang berbeda pada setiap iterasi. Jika dibandingkan dengan teknik *Substitution-Permutation Network*, teknik *Feistel Network* relatif lebih aman dari serangan kriptografi dan lebih mudah untuk diimplementasikan oleh pengembang *cipher*. Hal ini dikarenakan proses enkripsi dan dekripsi dari *Feistel Network* cenderung mirip dan pengembang *cipher* tidak perlu mengimplementasikan *inverse round function* yang digunakan pada proses dekripsi, sehingga *round function* yang digunakan dapat dirancang sekompleks mungkin untuk menghindari serangan kriptografi.

Terdapat beberapa langkah dalam operasi enkripsi/dekripsi dengan menggunakan *Feistel Network*. Langkah pertama yang dilakukan adalah membagi *plaintext* pada *block* masukkan menjadi dua bagian yang sama besar, disebut sebagai *left* dan *right*. Kemudian, salah satu bagian tersebut (tergantung jenis operasi enkripsi/dekripsi) dioperasikan dengan menggunakan *round function*, dimana *round function* tersebut dapat berupa operasi *substitution* maupun *permutation*. Terdapat beberapa motivasi dari pengaplikasian *round function* pada tahap ini. Tujuan dari operasi *substitution* pada *round function* adalah untuk menciptakan *confusion* pada *ciphertext* yang dihasilkan. Prinsip *confusion* pada *ciphertext* merupakan konsep dimana hubungan statistik yang ada diantara *plaintext*, *ciphertext*, dan juga *internal key*. Sedangkan itu, operasi *permutation* pada *round function* adalah untuk menciptakan *diffusion* pada *ciphertext* yang dihasilkan. Prinsip *diffusion* di dalam *ciphertext* merupakan konsep dimana satu bit di dalam suatu *plaintext* dapat memengaruhi banyak bit di dalam *cipher text*. Sebagai efek dari *diffusion*, pengubahan kecil yang terjadi pada suatu *plaintext* sebanyak satu atau dua bit dapat menghasilkan pengubahan pada bit - bit *ciphertext* yang tidak dapat diprediksi oleh *cryptanalyst*. Pada setiap perulangan *cipher*, salah satu bagian *plaintext* (tergantung jenis operasi enkripsi/dekripsi) dilakukan operasi XOR dengan hasil operasi *round function* pada pengulangan sebelumnya. Kemudian, bagian tersebut ditukar dengan lawannya (*left* menjadi *right* serta *right* menjadi *left*).

Pengaplikasian *round function* pada setiap pengulangan *cipher* memerlukan suatu *subkey* untuk memperkuat *cipher* yang diaplikasikan. *Subkey* (atau *internal key*) umumnya diderivasi dari *key scheduling function* menggunakan *external key* yang dimasukkan oleh pengguna. *Key scheduling function* tersebut haruslah memiliki periode fungsi yang besar untuk menghindari adanya perulangan *internal key* pada operasi enkripsi/dekripsi.

Umumnya beberapa *cipher* yang menggunakan *Feistel Network* mengulang operasi *cipher* sebanyak 8 hingga 16 kali. Semakin besar jumlah pengulangan, maka semakin besar pula efek *confusion* dan *diffusion* pada *ciphertext*. Hal tersebut akan menyebabkan *ciphertext* yang dihasilkan semakin sulit untuk dipecahkan oleh *cryptanalyst*.

C. Cryptographically Secure Pseudorandom Number Generator

Cryptographically Secure Pseudorandom Number Generator (CSPRNG) merupakan suatu klasifikasi *Pseudorandom Number Generator* dimana urutan angka yang dihasilkan tidak dapat diprediksi secara mudah serta tingkat *randomness* yang dihasilkan terdistribusi secara merata jika dilihat secara statistik. Umumnya, tingkat keamanan dari CSPRNG berasal dari *secret key* yang digunakan sebagai *seed* nilai awal pada urutan angka acak. Suatu algoritma CSPRNG yang sama jika di-*seed* dengan nilai *secret key* yang sama akan menghasilkan suatu urutan angka acak yang sama pula. Oleh karena itu, nilai *secret key* yang digunakan tidak boleh terekspos pada publik.

Keamanan suatu algoritma CSPRNG dari serangan kriptografi terletak pada tingkat kompleksitas untuk melakukan *reverse engineering* dari urutan angka tersebut. Terdapat beberapa tipe CSPRNG jika dilihat dari sisi algoritma yang digunakan untuk membangkitkan angka acak. Tipe pertama adalah CSPRNG yang menggunakan fungsi *hash* atau *encryption* yang sudah terbukti aman dari serangan kriptografi, seperti algoritma RSA. Tipe kedua adalah CSPRNG yang menggunakan permasalahan *number theory* yang termasuk klasifikasi *NP-Hard*. Tipe terakhir adalah CSPRNG yang menggunakan *special design* seperti penggunaan *entropy* dan *evolutionary algorithm*.

Pada makalah ini, penulis akan berfokus pada CSPRNG bertipe *number theoretic* sebagai algoritma utama yang digunakan untuk melakukan proses enkripsi/dekripsi pada *cipher*. Hal ini dikarenakan CSPRNG dengan tipe tersebut termasuk mudah untuk diimplementasikan pada *cipher* serta memiliki tingkat kompleksitas yang cukup rendah (umumnya memiliki kompleksitas linear). Penulis menggunakan dua algoritma CSPRNG dengan tipe *number theoretic* yang akan diaplikasikan pada *cipher*, yaitu *Blum-Blum-Shub Random Number Generator* dan *Blum-Micali-Random Number Generator*. Kedua CSPRNG tersebut menggunakan operasi modularitas dan prinsip kekongruenan untuk membuat urutan angka yang *cryptographically secure*.

D. Mode Operasi Cipher Blok

Pada cipher blok, terdapat beberapa mode operasi yang dapat digunakan. Mode-mode operasi tersebut dibedakan berdasarkan cara blok dioperasikan sebelum dilakukan proses enkripsi atau dekripsi oleh fungsi yang berada pada algoritmanya itu sendiri. Pada pembahasan kali ini, akan dibahas tentang lima mode operasi, yaitu:

- ECB (*Electronic Code Book*)
  ECB merupakan mode awal dari sebuah operasi cipher blok, dimana setiap blok plainteks dienkripsi secara individual dan independen dari blok lainnya, sehingga hasil enkripsi suatu blok tidak akan mempengaruhi enkripsi blok lainnya.

- CBC (*Cipher Block Chaining*)
  Pada CBC, enkripsi dilakukan dengan mempertimbangkan blok plainteks sebelumnya. Hal ini diraih dengan cara melakukan operasi XOR dengan hasil enkripsi sebelumnya. Untuk blok pertama, operasi XOR dilakukan dengan sebuah *initialization vector* yang ditentukan sebelumnya.

- CFB (*Cipher Feedback*)
  Pada CFB, cipherteks sebelumnya dienkripsi terlebih dahulu sebelum dilakukan XOR dengan plainteks blok sekarang. Hasil dari operasi ini kemudian akan diumpan kepada iterasi berikutnya. Mode ini dapat mengenkripsi data yang lebih kecil daripada ukuran blok.

- OFB (*Output Feedback*)
  Pada OFB, *keystream* atau pembangkitan kunci yang berasal dari *initialization vector* dibangkitkan secara independen, sehingga tidak berpengaruh pada blok, hanya berpengaruh pada kunci sebelumnya saja.

- CTR (*Counter*)
  Pada CTR, tidak dilakukan perantaian pada proses enkripsi maupun dekripsi, melainkan menggunakan sebuah nilai blok bit yang berukuran sama dengan ukuran blok plainteks. Karena tidak dilakukan perantaian, mode CTR dapat mengenkripsi secara tidak sekuensial.

III. RANCANGAN BLOCK CIPHER

A. Spesifikasi Algoritma

Algoritma GKA merupakan algoritma blok cipher yang memanfaatkan prinsip *Diffusion* dan *Confusion* serta menggunakan jaringan feistel dan fungsi putaran serta pembangkitan kunci untuk setiap putaran. Block Cipher ini menggunakan jaringan Feistel 16 putaran dengan ukuran kunci 128 bit dan ukuran blok 128 bit. Sebelum melakukan 16 putaran, blok akan di *shuffle (initial permutation)* yang beroperasi dalam satuan bit. Setelah 16 putaran dilakukan, blok yang dihasilkan akan di-*unshuffle* dengan menggunakan *inverse permutation* dari *initial permutation* pada saat melakukan *shuffle*.

B. Pembangkitan Kunci

Dalam setiap putaran pada jaringan feistel, dibangkitkan sebuah kunci putaran berukuran 128 bit. Kunci putaran dibangkitkan dengan menerima masukan jumlah putaran yang sudah dilakukan yang selanjutnya akan disebut sebagai *x* dan kunci *external key* yang merupakan blok berukuran 128 bit.

Kunci putaran dibangkitkan dengan cara mengubah kunci *external key* menjadi bilangan bulat yang selanjutnya akan disebut sebagai *y*. Nilai *x* dan *y* kemudian dimasukkan ke dalam fungsi *pseudo-random generator* yang memanfaatkan algoritma Blum-Blum-Slub. Berikut implementasi dari *pseudorandom generator* yang memanfaatkan algoritma Blum-Blum-Slub.

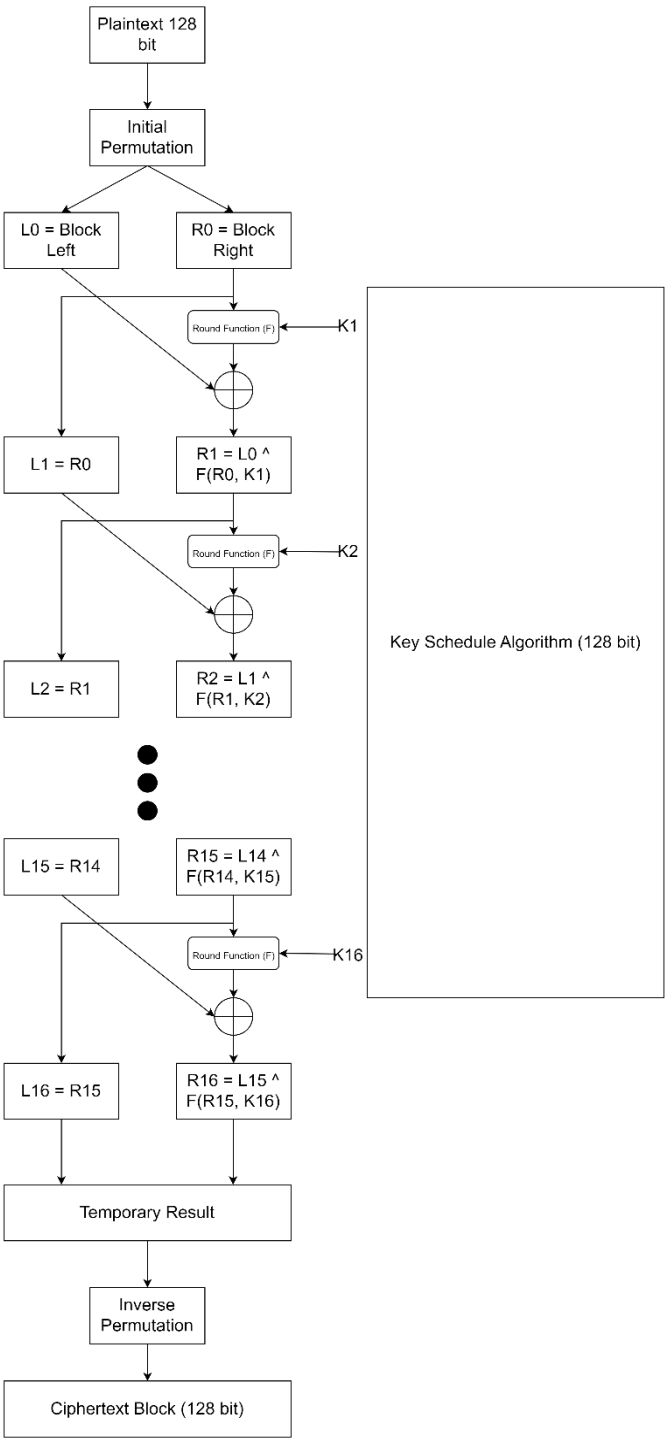
```
def generate_number_bbs(seed: int, index: int)
-> int:
# Blum-Blum-Shub Pseudo-Random Number Generator
(Key Expansion)
    exponent = binary_exponentiation(
        2, index + 1, lcm((FIRST_PRIME - 1),
(SECOND_PRIME - 1))
    )
    result = binary_exponentiation(seed,
exponent, FIRST_PRIME * SECOND_PRIME)

    return result
```

FIRST_PRIME dan SECOND_PRIME pada algoritma tersebut merupakan dua bilangan prima yang sangat besar sehingga pembangkitan kunci sulit untuk dipecahkan oleh kriptanalisis.

C. Skema Jaringan Feistel

Algoritma GKA memanfaatkan jaringan feistel untuk melakukan enkripsi dekripsi. Untuk setiap putaran enkripsi, input berupa 128 bit akan dibagi menjadi dua bagian, yaitu blok *left* dan blok *right* dengan 64 bit pertama menjadi blok *left* dan 64 bit kedua menjadi blok *right*. Blok *right* kemudian akan dimasukkan ke dalam fungsi putaran *f* bersama dengan internal *key* yang dibangkitkan pada setiap putaran. Hasil dari fungsi putaran tersebut berukuran 64 bit dan dilakukan XOR dengan blok *left*. Hasil XOR tersebut kemudian digabungkan dengan blok *right* untuk menghasilkan blok berukuran 128 bit dengan 64 bit pertama merupakan blok *right* dan 64 bit terakhir adalah blok hasil XOR itu sendiri. Hasil penggabungan tersebut akan menjadi masukkan untuk putaran selanjutnya.

Dalam setiap putaran dekripsi, input berupa 128 bit akan dibagi menjadi dua bagian, yaitu blok *left* dan blok *right* dengan 64 bit pertama menjadi blok *left* dan 64 bit kedua menjadi blok *right*. Blok *left* kemudian akan dimasukkan ke dalam fungsi putaran *f* bersama dengan internal *key* yang dibangkitkan pada setiap putaran. Hasil dari fungsi putaran tersebut berukuran 64 bit dan dilakukan XOR dengan blok *right*. Hasil XOR tersebut kemudian digabungkan dengan blok *left* untuk menghasilkan blok berukuran 128 bit dengan 64 bit pertama merupakan hasil XOR dan 64 bit terakhir adalah blok *left* itu sendiri. Hasil penggabungan tersebut akan menjadi masukkan untuk putaran selanjutnya. Skema Jaringan Feistel dapat dilihat pada gambar dibawah ini.



Gambar 1. Skema *Feistel Network*

D. Fungsi Putaran

Fungsi putaran menerima input berupa blok konten berukuran 64 bit dan internal *key* berukuran 128 bit. Fungsi putaran akan melakukan substitusi *byte* dan permutasi *byte* secara berurutan. Untuk melakukan pembangkitan s-box dan p-box, penulis menggunakan algoritma Blum-Micali untuk membangkitkan bilangan *pseudorandom*. Berikut ini merupakan implementasi dari Blum-Micali *pseudorandom number generator*.

```
def generate_list_bm(seed: int, n: int) ->
list[int]:
    # Blum-Micali Pseudo-Random Number
Generator (Round Function)
    list_of_random_int = [seed]
    for i in range(n):
        list_of_random_int.append(
            binary_exponentiation(
                PRIMITIVE_ROOT,
                list_of_random_int[i],
                LARGE_PRIME
            )
        )

    return list_of_random_int[1:]
```

Bilangan PRIMITIVE_ROOT haruslah berupa *primitive root modulo* dari bilangan LARGE_PRIME. Bilangan LARGE_PRIME itu sendiri haruslah berupa bilangan prima yang sangat besar sehingga perhitungan *discrete logarithm* dari modulo tersebut menjadi sangat lama.

Substitusi *byte* dilakukan dengan memanfaatkan s-box berukuran 16 × 16 yang di-*generate* berdasarkan *internal* key menggunakan algoritma Blum-Micali yang bertujuan untuk mendapatkan urutan bilangan yang *pseudorandom.* Setiap nilai yang terdapat di dalam matriks merupakan bilangan berukuran 8 bit. Setiap *byte* pada *block* konten kemudian disubstitusi berdasarkan s-box tersebut. Substitusi pada setiap *byte* dilakukan dengan cara membagi *byte* tersebut menjadi 2 bagian berukuran 4 bit. Bagian pertama berisi bit 0-3 (asumsikan *a)*, bagian pertama berisi bit 4 - 7 (asumsikan *b*). Hasil substitusi dari *byte* ini adalah nilai yang terdapat pada s-box baris ke-*a* kolom ke-*b*. Sebagai contoh, akan dilakukan substitusi pada ***byte* 00110101.** *Byte* tersebut dibagi menjadi 2 dengan bagian pertama 0011 dan bagian kedua 0101. Maka hasil substitusi adalah nilai byte yang ada pada baris ke-3 kolom ke-5.

Permutasi *byte* dilakukan dengan memanfaatkan p-box yang merupakan *array* dengan panjang 16 elemen yang di-*generate* berdasarkan *internal key* menggunakan algoritma Blum-Micali untuk membangkitkan bilangan *pseudorandom* dan juga algoritma Fisher-Yates untuk melakukan permutasi. Setiap *byte* pada *block* konten akan ditukar berdasarkan *index* yang dihasilkan oleh urutan bilangan *pseudorandom*. Berikut ini merupakan implementasi dari algoritma Fisher-Yates untuk membangkitkan *pseudorandom permutation* pada setiap *bytes* pada *block* konten.

```
def shuffle_bytes(content: bytes, seed: int) ->
bytes:
    # Fisher-Yates Shuffle Algorithm
    arr = bytearray(content)
    length = len(content)
    random_int = generate_list_bm(seed, length)

    for i in range(length - 1, 0, -1):
        j = random_int[i] % (i + 1)
        arr[i], arr[j] = arr[j], arr[i]

    return bytes(arr)
```

IV. Eksperimen dan Pembahasan Hasil

A. Analisis Waktu

Dalam menganalisis waktu enkripsi dan dekripsi pesan, digunakan mode ECB (*Electronic Code Book*) agar hasil yang didapatkan setara antar perbedaan ukuran plainteks. Pengujian juga dilakukan dengan menggunakan plainteks dengan ukuran 32 *byte*, 256 *byte*, 1024 *byte*, serta 2048 *byte*. Semua plainteks berada dalam format huruf "ABCD" secara berulang untuk memudahkan pembuatan plainteks uji. Kunci yang digunakan adalah "abcdefghijklmnop".

TABLE I. Tabel Pengujian Analisis Waktu

Ukuran *Plaintext*	Waktu Enkripsi/Dekripsi

32 *byte*	Enkripsi: 0.925 detik
	`Time encrypt: 0.9256789684295654 s`
	Dekripsi: 0.918 detik
	`Time decrypt: 0.918828010559082 s`

256 *byte*	Enkripsi: 7.49 detik
	`Time encrypt: 7.490213394165039 s`
	Dekripsi: 7.34 detik
	`Time decrypt: 7.34660530090332 s`

1024 *byte*	Enkripsi: 29.918 detik
	`Time encrypt: 29.918490409851074 s`
	Dekripsi: 29.518 detik
	`Time decrypt: 29.518614053726196 s`

2048 *byte*	Enkripsi: 58.8 detik
	`Time encrypt: 58.80809545516968 s`
	Dekripsi: 58.5 detik
	`Time decrypt: 58.53667378425598 s`

Dari hasil pengujian diatas, terlihat bahwa enkripsi dan dekripsi tidak jauh berbeda pada *runtime*-nya. Hal ini disebabkan karena algoritma enkripsi dan dekripsi pada dasarnya tidak jauh berbeda.

Hasil pengujian juga menunjukkan bahwa waktu komputasi naik secara linear (apabila plainteks berukuran 8 kali lipat maka waktu enkripsi juga akan naik 8 kali lipat). Hal ini disebabkan karena enkripsi dilakukan per-blok, sehingga waktu enkripsi per blok tidak akan jauh berbeda.

Dari hasil pengujian ini, diketahui pula bahwa algoritma semacam ini tidak cocok untuk mengenkripsi *file* dengan ukuran besar karena akan memakan waktu yang cukup lama. Namun, karena algoritma ini cukup *secure*, algoritma ini tetap dapat digunakan sebagai algoritma enkripsi pesan singkat yang penting, seperti pertukaran *key*.

B. Analisis Efek Longsoran

Dalam menganalisis efek longsoran, digunakan plainteks berukuran 32 bytes agar proses enkripsi/dekripsi tidak terlalu lambat. Hasil percobaan diberikan dalam bentuk luaran program dalam *byte*, karena sulit untuk menampilkannya dalam bentuk karakter. Pengujian dilakukan dengan kunci "abcdefghijklmnop". Berikut adalah tabel yang berisi hasil percobaan:

TABLE II. Tabel Pengujian Efek Longsoran Mode ECB

No	Hasil Percobaan

1	`Content: b'Rayhan Kinan sedang memakan kue '`
	`Encrypted: b'\xf7\x01\x9cFw\x90\x0em\xb7\x0eOF\x93\xf1e\xcb\x07\xf6\xdfq\xda\xa1\xa7\xc3y\xf8\t\x02}\x113\xd8'`
	`Time encrypt: 0.9183506965637207 s`
	`Decrypted: b'Rayhan Kinan sedang memakan kue '`
	`Time decrypt: 0.9180827140808105 s`

2	`Input content: Sayhan Kinan sedang memakan kue`
	`Content: b'Sayhan Kinan sedang memakan kue '`
	`Encrypted: b'\xddV<\xbc\xdd\xd5\xe3\xe9\xe1\xccx\n\xcd\xc12\xf9\x07\xf6\xdfq\xda\xa1\xa7\xc3y\xf8\t\x02}\x113\xd8'`
	`Time encrypt: 0.9175562858581543 s`
	`Decrypted: b'Sayhan Kinan sedang memakan kue '`
	`Time decrypt: 0.9184818267822266 s`

3	`Input content: Tayhan Kinan sedang memakan kue`
	`Content: b'Tayhan Kinan sedang memakan kue '`
	`Encrypted: b'\xa5\x8c\n\x86m[\xe4U\xcf\xfd\xbc\xfd\x14\x83\x7f\x99\x07\xf6\xdfq\xda\xa1\xa7\xc3y\xf8\t\x02}\x113\xd8'`
	`Time encrypt: 0.9164025783538818 s`
	`Decrypted: b'Tayhan Kinan sedang memakan kue '`
	`Time decrypt: 0.9253418445587158 s`

Dari hasil pengujian diatas, terlihat bahwa satu bit saja yang berubah pada masukan (diraih dengan cara mengubah satu huruf dengan huruf tetangganya) dapat mengubah hasil yang terenkripsi cukup besar, khususnya pada *byte* yang berada pada blok tersebut. Blok yang berbeda dengan blok yang diubah tidak akan mendapat pengaruhnya, dikarenakan enkripsi dilakukan dengan mode ECB.

Untuk menguji efek longsoran pada lebih dari satu blok, dapat digunakan salah satu mode lainnya, yaitu CBC:

TABLE III. Tabel Pengujian Efek Longsoran Mode CBC

No	Hasil Percobaan

1	`Input content: Rayhan Kinan sedang memakan kue`
	`Content: b'Rayhan Kinan sedang memakan kue '`
	`Encrypted: b'2c\x06cg\xc9\x80\xear\xe2&B\xcaI\xc8\xcb\xc2wD\xd5/S\rP\x00lf\x11\xc6\xb9\x0e\xb1'`
	`Time encrypt: 0.905311107635498 s`
	`Decrypted: b'Rayhan Kinan sedang memakan kue '`
	`Time decrypt: 0.9065244197845459 s`

2	`Input content: Sayhan Kinan sedang memakan kue` `Content: b'Sayhan Kinan sedang memakan kue '` `Encrypted: b'!\'\xe4\xeen\xc9\x94,r\x98)\x98\x83\x053\x9e!\xa7\x1b\xc5W`\xa9\x89\xef"\x8d\x9e\x1c\x9f\x9bx'` `Time encrypt: 0.9109454154968262 s` `Decrypted: b'Sayhan Kinan sedang memakan kue '` `Time decrypt: 0.9163923263549805 s`

3	`Input content: Tayhan Kinan sedang memakan kue`
	`Content: b'Tayhan Kinan sedang memakan kue '`
	`Encrypted: b'\xd9U\x1a\xfe\x992\x947\xa8\xee\x9b\x8fq\x07\x9aT\xb1\x87(\xfam\xd1\xfc\x98\xb5s3\x0f\xd9\\\xa2\xce'`
	`Time encrypt: 0.9142308235168457 s`
	`Decrypted: b'Tayhan Kinan sedang memakan kue '`
	`Time decrypt: 0.9209928512573242 s`

Dengan mode CBC, terlihat bahwa efek longsoran berlangsung secara sangat masif walaupun hanya terdapat satu bit saja yang berubah. Dapat dilihat apabila dibandingkan dari

konten yang terenkripsi, semua *byte*-nya sangat berbeda dengan tetangga plainteksnya.

Dari hasil pengujian diatas, diketahui bahwa algoritma enkripsi ini cukup baik dalam menerapkan prinsip *diffusion* dan *confusion* karena dua plainteks yang berdekatan dapat menjadi dua cipherteks yang sangat amat berbeda.

C. Analisis Ruang Kunci

Dalam inisalisasi algoritma, dibutuhkan kunci yang akan digunakan dalam algoritma ini. Panjang kunci yang dibutuhkan adalah sepanjang 16 *bytes* atau 128 bit. Apabila dihitung, kemungkinan kunci yang ada sebanyak 2^{128}, atau sekitar $3 * 10^{38}$ kemungkinan kunci.

Selain dari kemungkinan kunci eksternal yang akan digunakan dalam sistem, terdapat juga *initialization vector* yang harus digunakan pada mode CBC, CFB, OFB dan CTR, yang masing-masing memiliki kunci sepanjang 128 bit pula (kecuali CTR, hanya 64 bit).

Apabila kemungkinan keduanya digabungkan, maka ruang kombinasi kunci serta *initialization vector* yang mungkin ada sebanyak 2^{256} kombinasi, atau sebanyak 10^{77} kombinasi. Kombinasi sebanyak ini akan sangat sulit untuk dilakukan *brute force* untuk menentukan kunci ataupun *initialization vector*.

V. KESIMPULAN DAN SARAN

A. Kesimpulan

Block cipher adalah salah satu metode kriptografi yang cukup marak digunakan untuk mengenkripsi bagian dari informasi. Dalam pengembangan sebuah *block cipher*, banyak teknik yang dapat dilakukan untuk memperkuat algoritma enkripsi agar lebih sulit untuk dipecahkan oleh para kriptanalis, seperti pengulangan, substitusi, serta permutasi dari plainteks yang terkait. Satu algoritma block cipher juga dapat digunakan dalam banyak mode operasi, sehingga memperbanyak kemungkinan *cipher* juga. Semua hal ini dilakukan untuk memperkuat algoritma agar tidak mudah dipecahkan.

B. Saran

Untuk penelitian-penelitian selanjutnya, dapat digunakan algoritma yang lebih sangkil atau bahasa pemrograman yang lebih cepat, dikarenakan waktu yang dibutuhkan untuk mengenkripsi serta mendekripsi pesan cukup lambat pada algoritma yang ada sekarang.

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan puji dan syukur kepada Tuhan Yang Maha Esa atas berkah dan rahmatnya sehingga makalah yang berjudul "Perancangan Algoritma Block Cipher Menggunakan Cryptographically Secure Pseudorandom Generator" dapat diselesaikan dengan baik dan lancar. Penulis juga mengucapkan terima kasih kepada bapak Rinaldi Munir selaku dosen pengampu mata kuliah IF4020 Kriptografi atas ilmu dan bimbingannya.

REFERENSI

[1] Munir, Rinaldi. 2023. Block Cipher (Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2022-2023/13-Block-Cipher-Bagian1-2023.pdf . Diakses pada 1 Maret 2023.

[2] Munir, Rinaldi. 2023. Perancangan Block Cipher. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2022-2023/14-Prancangan-block-cipher-2023.pdf . Diakses pada 1 Maret 2023.

[3] Blum, Lenore; Blum, Manuel; Shub, Michael. 1986. A Simple Unpredictable Pseudo-Random Number Generator. https://shub.ccny.cuny.edu/articles/1986-A_simple_unpredictable_pseudo-random_number_generator.pdf. Diakses pada 27 Februari 2023.

[4] Blum, Manuel; Micali, Silvio. 1984. How To Generate Cryptographically Strong Sequences Of Pseudo Random Bits. http://www.csee.wvu.edu/~xinl/library/papers/comp/Blum_FOCS1982.pdf. Diakses pada 27 Februari 2023.

[5] Knuth, Donald; 1969. Seminumerical Algorithms. The Art of Computer Programming. Vol. 2. Reading, MA: Addison–Wesley. pp. 139–140. Diakses pada 28 Februari 2023.

PERNYATAAN

Dengan ini kami menyatakan bahwa makalah yang kami tulis ini adalah tulisan sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.