Analisis Perbandingan Algoritma AES dan ChaCha20-Poly1305 dalam Enkripsi Konten *Livestreaming*

Maulana Anindita A.A. 18219086
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
18219086@std.stei.itb.ac.id

***Abstrak*—Maraknya media live streaming memunculkan banyak resiko keamanan seperti serangan man-in-the-middle, eavesdropping, dan DDoS. Salah satu cara untuk mengamankan sebuah live stream saat berlangsung adalah dengan mengenkripsi data yang ditransmisi secara real-time. Makalah ini akan membandingkan kinerja antara dua algoritma enkripsi yaitu algoritma AES dengan mode cipher block chaining dan algoritma ChaCha20-Poly1305. Penelitian dilakukan dengan mengukur kinerja dari kedua algoritma tersebut dalam mengenkripsi file.**

Keywords—livestream, enkripsi, kinerja

I. PENDAHULUAN

Pada era digital ini, terdapat banyak media untuk membuat dan menikmati berbagai konten melalui akses internet. Salah satu media yang paling diminati untuk membuat dan menikmati konten adalah media *live streaming*. Menurut Encyclopedia Britannica [1], *live streaming* atau siaran langsung merupakan transmisi informasi dalam format video secara real-time melalui internet. Salah satu *live streaming platform* yang paling populer saat ini adalah platform Twitch yang memiliki kurang lebih 2,5 juta rata-rata penonton dan 4 juta penonton maksimum per Mei 2023 [2].

Dengan maraknya platform live streaming, banyak juga risiko keamanan yang muncul dalam implementasinya. Risiko seperti serangan *man-in-the-middle*, *eavesdropping*, dan DDoS dapat terjadi pada *live stream* yang tidak diamankan. Demi mengamankan sebuah *live stream* dari risiko serangan, salah satu cara yang dapat dilakukan adalah melakukan enkripsi data yang ditransmisi saat live stream berlangsung. Terdapat beberapa algoritma enkripsi yang dapat digunakan untuk mengamankan transmisi data, beberapa diantaranya adalah algoritma *block cipher* AES dan algoritma *stream cipher* ChaCha20-Poly1305 yang akan dibahas lebih lanjut pada bagian berikutnya.

II. LANDASAN TEORI

A. AES Block Cipher

Advanced Encryption Standard (AES) merupakan sebuah algoritma *block cipher* kunci simetris yang dikemukakan oleh NIST pada tahun 2001. Algoritma AES merupakan algoritma iteratif yang dibuat dengan basis dua teknik utama untuk enkripsi dan dekripsi data yaitu *substitution and permutation network* (SPN). AES dapat menerima *block plaintext* sebesar 128 bit atau 16 byte yang direpresentasikan sebagai matrix 4x4. Fitur penting lain dari algoritma AES adalah fitur iterasi perubahan data (*number of rounds*). Jumlah iterasi ini bergantung oleh besarnya ukuran kunci yang diberikan. Terdapat tiga ukuran kunci yang dapat digunakan untuk enkripsi dan dekripsi data yaitu 128, 192, atau 256 bits. Untuk ukuran kunci 128 bits, AES akan melakukan 10 kali iterasi, 12 kali untuk ukuran 192 bit, dan 14 kali untuk 256 bit [3].

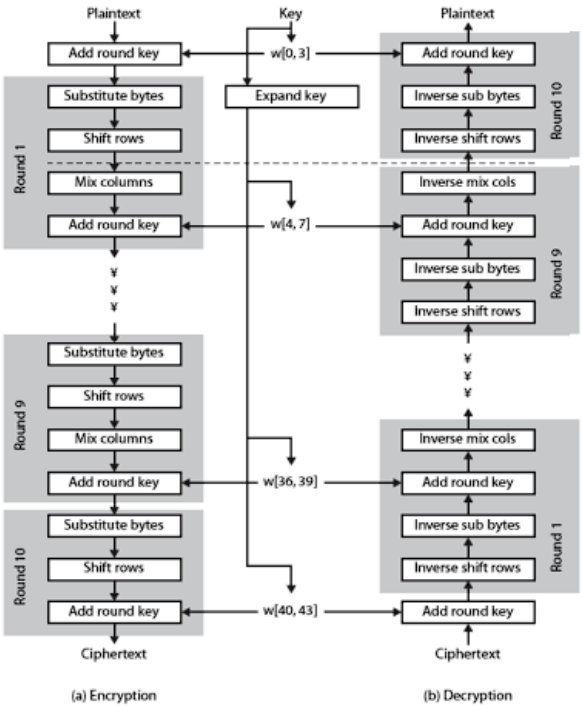

Fig. 1 Arsitektur AES [3]

Dalam sebuah iterasi proses enkripsi setiap *block*, terdapat empat sub-proses yang berlaku:

1. *SubBytes*: Setiap byte dari data masukan disubstitusikan dengan byte dari *substitution box* (S-

box) yang sesuai. Tahap ini menghasilkan non-linearitas dan *confusion* pada data hasil.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Fig 2. S-Box Algoritma AES [3]

2. *ShiftRows*: Setiap baris dari data digeser secara secara siklis. Baris pertama tidak digeser, baris kedua digeser ke kiri sekali, baris ketiga digeser ke kiri dua kali, dan baris keempat digeser ke kiri tiga kali. Tahap ini memberikan *diffusion* dan menyebarkan data pada setiap baris.
3. *MixColumns*: Setiap kolom dari data diacak menggunakan sebuah matrix pengacakan. Tahap ini memberikan *diffusion* lebih dalam serta memastikan bahwa setiap byte dalam kolom bergantung pada empat byte lainnya.
4. *AddRoundKey*: Pada tahapan ini, dilakukan operasi XOR antara data dengan *round key*. *Round key* sendiri dapat didapatkan dari kunci awal menggunakan sebuah *Key-Scheduling Algorithm* (KSA). Tahapan ini memberikan kunci unik pada data dan membuat hasil enkripsi bergantung pada kunci.

Keempat proses ini akan dilaksanakan berulang pada setiap iterasi kecuali iterasi terakhir dimana proses *MixColumns* dilewati. Hasil data dari iterasi terakhir merupakan data yang telah terenkripsi.

Untuk menangani dan memproses enkripsi dari beberapa jumlah block, Terdapat lima metode *block cipher* utama yang dapat digunakan yaitu *Electronic Code Book* (ECB), *Cipher Block Chaining* (CBC), *Cipher Feedback* (CFB), *Output Feedback* (OFB), dan *Counter Mode* (CTR) [4].

1. *Electronic Codebook* (ECB)

   ECB menggunakan konsep kunci simetris yang paling sederhana, yaitu setiap blok dienkripsi satu persatu dengan kunci yang sama. Metode enkripsi yang sederhana inilah yang membuat ECB menjadi metode yang paling mudah dan cepat untuk diimplementasikan. Namun, ECB memiliki kekurangan dimana *block* plaintext yang identik akan menghasilkan *block* ciphertext yang identik juga. Hal ini menyebabkan pola enkripsi dari ECB mudah ditebak. Berikut adalah rumus yang merepresentasikan proses enkripsi dan dekripsi dari metode ECB.

   $$\text{Enkripsi: } Ci = Ek(Pi)$$

   $$\text{Dekripsi: } Pi = Dk(Ci)$$

2. *Cipher Block Chaining* (CBC)

   Metode CBC memberikan keamanan lebih untuk ukuran data yang lebih besar. Hal ini karena Metode CBC akan melakukan operasi XOR antara block ciphertext dengan sebuah Initialization Value (IV) sebelum melakukan enkripsi. Untuk melakukan enkripsi block selanjutnya, hasil dari enkripsi block ciphertext ini akan di-XOR-kan dengan block ciphertext sebelumnya. Hal ini akan berlaku sampai enkripsi block terakhir. Dengan ini, jika data yang sama dienkripsi beberapa kali, maka akan menghasilkan ciphertext yang unik berdasarkan IV yang digunakan. Berikut adalah proses enkripsi dan dekripsi metode CBC.

   $$\text{Enkripsi: } Ek(Pi \oplus Ci{-}1)$$

   $$\text{Dekripsi: } Dk(Ci) \oplus Ci{-}1$$

3. *Cipher Feedback* (CFB)

   Dalam metode CFB, algoritma *block cipher* digunakan sebagai fungsi *feedback* untuk membangkitkan sebuah *keystream* yang akan di-XOR-kan pada data. Mekanisme *feedback* ini menyebabkan proses enkripsi dan dekripsi yang sinkron, serta menjamin bahwa setiap byte dari ciphertext bergantung pada byte ciphertext sebelumnya. Berikut adalah rumus dari proses enkripsi dan dekripsi metode CFB.

   $$\text{Enkripsi: } Pi \oplus Ms[Ek(Ri{-}1)]$$

   $$\text{Dekripsi: } Ci \oplus Ms[Ek(Ri{-}1)]$$

   Dimana Ms[Ek(Ri)] adalah s bit paling signifikan dari nilai enkripsi Ri. Seperti metode CBC, hasil dari enkripsi *block* plaintext akan bergantung pada *block* ciphertext sebelumnya.

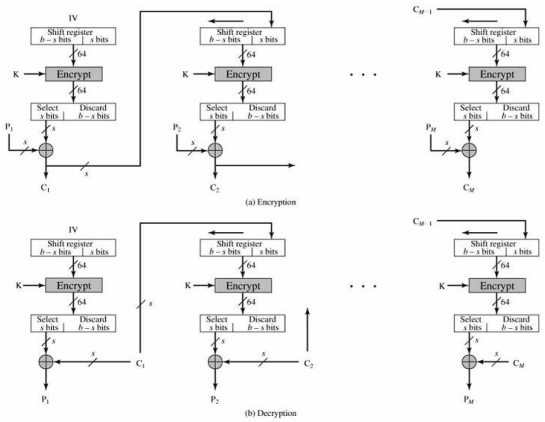

Fig 2. Proses Metode *Cipher Feedback* [4]

4. *Output Feedback* (OFB)

   Pada dasarnya, metode OFB merupakan metode hampir sama dengan metode CFB dimana algoritma dapat membangkitkan sebuah *keystream* yang akan di-XOR-kan pada data untuk menghasilkan ciphertext. Hanya saja, dalam metode OFB, proses enkripsi dan dekripsi saling paralel dengan satu sama lain.

5. *Counter Mode* (CTR)

   Dalam mode CTR, sebuah *counter* R digunakan bersamaan dengan nilai *nonce* untuk membangkitkan *keystream*. *Keystream* tersebut lalu akan di-XOR-kan dengan data *plaintext* untuk menghasilkan *ciphertext*. Berikut adalah rumus dari enkripsi dan dekripsi metode CTR.

$$\text{Enkripsi: } Ci = Pi \oplus Ek(Ri-1)$$

$$\text{Dekripsi: } Pi = Ci \oplus Ek(Ri-1)$$

B. ChaCha20-Poly1305

Algoritma ChaCha20 merupakan sejumlah algoritma *stream cipher* yang berupa varian dari algoritma *stream cipher* Salsa20. Algoritma ChaCha20 memiliki basis desain yang sama seperti Salsa20, hanya saja algoritma ChaCha20 menambahkan jumlah *diffusion* pada tiap *round* untuk menambahkan keamanan pada proses enkripsi [5]. Sedangkan, Poly1305 merupakan algoritma *message authentication code* (MAC) yang menggunakan operasi perkalian polinomial untuk menghasilkan MAC yang aman dan terjamin keasliannya [7].

ChaCha20 membangun *keystream* dengan mengaplikasikan ChaCha20 *block function* kepada kunci, *nonce*, dan sebuah *block counter*. Data *plaintext* akan dienkripsi menggunakan *keystream* tersebut, dengan *block* i dari data tersebut di-XOR-kan dengan hasil dari ChaCha20 *block function* dan dievaluasi menggunakan *block counter* i. Berikut adalah *syntax* dari ChaCha20 *block function*.

$$\{0,1\}^{256} * \{0,1\}^{32} * \{0,1\}^{96} \rightarrow \{0,1\}^{512}$$

ChaCha20 *block function* menerima input kunci berukuran 32 byte, *block number* berukuran 4 byte, dan nonce berukuran 12 byte, dan menghasilkan 64 byte yang bersifat pseudo-random. Sedangkan, Poly1305 menerima masukan kunci yang berukuran 16 byte dan sebuah pesan dengan panjang bebas dan mengeluarkan *digest* yang berukuran 16 byte pada pesan. Keluaran dari Poly1305 adalah hasil potongan dari sebuah polinomial. Koefisien dari polinomial tersebut ditentukan oleh pesan masukan dengan setiap 16 byte dari pesan tersebut di-*encode* ke nilai integer modulo 2^{130} - 5. Ketika dienkripsi dengan *one-time pad* (OTP), hasil *digest* dari pesan akan berbentuk MAC yang bersifat *information-theoretic* [6].

III. DESAIN PENELITIAN

Penelitian dilakukan dengan membandingkan performa kedua algoritma dalam mengenkripsi konten *live stream*. Hal ini karena performa dari sebuah algoritma enkripsi akan menentukan kemampuan algoritma tersebut untuk memproses dan mengenkripsi data *real-time*. Performa dari masing-masing algoritma enkripsi akan diukur menggunakan kecepatan enkripsi. Semakin cepat proses enkripsi, maka semakin baik performa dari algoritma enkripsi. Program penelitian dibuat menggunakan bahasa pemrograman python dengan bantuan *library cryptography* yang sudah mencakup kedua algoritma AES dan ChaCha20-Poly1305 di dalamnya [8]. Penelitian dilakukan menggunakan perangkat lunak dan keras sebagai berikut

- OS: Windows 10 64-bit
- Processor: AMD Ryzen 5 3500U
- RAM: 8 GB

Konten *live stream* yang digunakan untuk pengujian program merupakan *file* video .mp4 yang berdurasi 53 menit dan berukuran 61,2 MB. Hal ini dilakukan untuk mensimulasikan enkripsi konte *live stream* yang kebanyakan berlangsung selama lebih dari satu jam.

Program penelitian dibagi menjadi dua fungsi utama:

1. *Measure AES Performance*

   Fungsi ini mengukur kecepatan algoritma AES dalam enkripsi *live stream*. Pada fungsi ini, metode *block cipher* yang digunakan adalah metode *Cipher Block Cipher* (CBC). Hal ini dikarenakan metode CBC merupakan metode yang memberikan keamanan tinggi terhadap hasil enkripsi. Berikut adalah *source code* dari fungsi pengukuran kecepatan algoritma AES.

```
from cryptography.hazmat.primitives.ciphers import
Cipher, algorithms, modes
from cryptography.hazmat.backends import
default_backend
from cryptography.hazmat.primitives.ciphers.aead
import ChaCha20Poly1305
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2
import PBKDF2HMAC
from cryptography.hazmat.backends import
```

```
default_backend
import os
import time

def measure_aes_performance(stream, key,
chunk_size):
    init_vector = os.urandom(16)
    cipher = Cipher(algorithms.AES(key),
modes.CBC(init_vector), backend=default_backend())

    start_time = time.perf_counter()
    encryptor = cipher.encryptor()

    while True:
        chunk = stream.read(chunk_size)
        if not chunk:
            break

        ciphertext = encryptor.update(chunk)

    end_time = time.perf_counter()

    encryption_speed = (stream.tell() // (end_time
- start_time)) // 1000000
    return encryption_speed
```

Fungsi ini menerima masukan *stream* yaitu nama dari *file live stream* yang akan dienkripsi, kunci yang akan digunakan, dan *chunk_size* yang merepresentasikan ukuran data yang ditransmisi saat *live stream* berlangsung. Dalam fungsi ini, *file* akan dibagi menjadi beberapa *chunk* yang berukuran sama dengan nilai *chunk_size*, lalu akan diproses dan dienkripsi per *chunk* dari file tersebut. Proses ini akan dilakukan berulang hingga *chunk* terakhir dari file. Kecepatan dari enkripsi akan dihitung dengan menentukan *start_time* yaitu waktu sebelum proses enkripsi dimulai, dan *end_time* yaitu waktu setelah proses enkripsi selesai. Lalu, jumlah byte yang diproses akan dibagi dengan jumlah waktu yang berlangsung selama proses enkripsi. Hasil dari operasi ini adalah kecepatan enkripsi dalam satuan *byte per second*. Agar hasil lebih mudah dibaca, maka hasil kecepatan akan dikonversi menjadi satuan *megabyte per second*.

2. *Measure ChaCha20-Poly1305 Performance*

Fungsi ini mengukur kecepatan algoritma ChaCha20-Poly1305 dalam enkripsi *live stream*. Berikut adalah *source code* dari fungsi pengukuran algoritma ChaCha20-Poly1305.

```
from cryptography.hazmat.primitives.ciphers
import Cipher, algorithms, modes
from cryptography.hazmat.backends import
default_backend
from
cryptography.hazmat.primitives.ciphers.aead
import ChaCha20Poly1305
from cryptography.hazmat.primitives import
hashes
from
cryptography.hazmat.primitives.kdf.pbkdf2
import PBKDF2HMAC
from cryptography.hazmat.backends import
default_backend
import os
import time

def
measure_chacha20_poly1305_performance(stream,
password, chunk_size):
    salt = b'salt'
    backend = default_backend()
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=backend
    )
    key = kdf.derive(password)
    cipher = ChaCha20Poly1305(key)

    start_time = time.perf_counter()

    while True:
        chunk = stream.read(chunk_size)
        if not chunk:
            break

        nonce = os.urandom(12)
        ciphertext = cipher.encrypt(nonce,
chunk, None)

    end_time = time.perf_counter()

    encryption_speed = (stream.tell() //
(end_time - start_time)) // 1000000
    return encryption_speed
```

Cara kerja fungsi ini hampir sama seperti fungsi pengukuran kecepatan AES, hanya saja fungsi ini melakukan penurunan kunci dari sebuah masukan *password* menggunakan *key derivation function* (KDF).

Terakhir, penelitian akan dilakukan dengan membandingkan kecepatan enkripsi kedua algoritma pada berbagai nilai *chunk_size*. Hal ini untuk melihat apakah terdapat perubahan terhadap performa kedua algoritma enkripsi ketika ukuran *chunk* yang diproses semakin membesar.

IV. Analisis dan Pembahasan

Berikut adalah hasil pengukuran kecepatan kedua algoritma yang didapatkan setelah pengujian program penelitian.

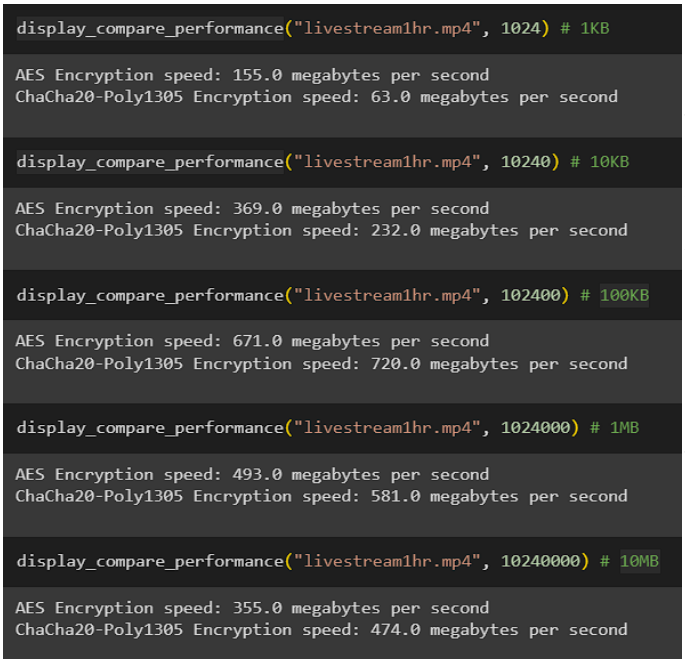

Fig 3. Hasil Pengukuran Kecepatan Kedua Algoritma

Tabel 1. Tabel Hasil Pengukuran Kecepatan Kedua Algoritma

Chunk Size	Kecepatan	
	AES	ChaCha20-Poly1305
1KB	155 MB/s	63 MB/s
10KB	369 MB/s	232 MB/s
100KB	671 MB/s	720 MB/s
1MB	577 MB/s	604 MB/s
10MB	355 MB/s	474 MB/s

Berdasarkan hasil pengukuran kecepatan di atas, dapat dilihat bahwa pada ukuran *chunk* rendah (1 KB s.d. 10 KB), algoritma AES dapat mengenkripsi *file* lebih cepat daripada algoritma ChaCha20-Poly1305. Namun, pada ukuran *chunk* tinggi (100 KB s.d. 10 MB), dapat dilihat bahwa algoritma ChaCha20-Poly1305 dapat mengenkripsi *file* dengan kecepatan yang lebih tinggi. Hal ini dapat terjadi karena untuk ukuran *chunk* yang lebih kecil, AES dengan metode CBC cenderung memiliki performa yang lebih baik dibandingkan ChaCha20-Poly1305. Hal ini disebabkan karena AES merupakan algoritma *block cipher* yang beroperasi pada *block* dengan ukuran tetap (128 bit atau 16 byte). Dengan ukuran *chunk* yang lebih kecil, proses enkripsi dapat lebih baik berpadanan dengan ukuran blok AES, sehingga pemrosesan menjadi lebih efisien. *Overhead* yang terkait dengan langkah inisialisasi dan finalisasi pada setiap *chunk* relatif lebih kecil dibandingkan dengan ukuran *chunk* secara keseluruhan, yang menguntungkan AES.

Di sisi lain, ChaCha20 merupakan *stream cipher* yang beroperasi pada aliran data kontinu. ChaCha20 tidak memiliki ukuran *block* yang tetap, dan enkripsinya dapat dilakukan byte-per-byte atau bit-per-bit. Ketika ukuran chunk semakin besar, ChaCha20 mulai menunjukkan keunggulannya dibandingkan AES. ChaCha20 dirancang untuk sangat parallel dan efisien dalam skenario *live streaming*. Dengan ukuran *chunk* yang lebih besar, *overhead* dari langkah inisialisasi dan finalisasi menjadi relatif lebih kecil, sehingga ChaCha20 dapat dengan efisien memproses aliran data.

Secara ringkas, untuk ukuran chunk yang lebih kecil, AES dengan mode CBC dapat memiliki keunggulan karena berpadanan dengan ukuran blok. Namun, ketika ukuran chunk semakin besar, desain streaming dan sifat paralel ChaCha20 membuatnya lebih efisien, sehingga memiliki performa yang lebih baik dibandingkan AES.

Berdasarkan hasil, juga dapat dilihat bahwa kecepatan enkripsi kedua algoritma mencapai puncak pada ukuran *chunk* 100 KB, lalu menurun pada *chunk* size yang lebih tinggi. Pada ukuran 100 KB, kedua algoritma mencapai keseimbangan optimal antara paralelisasi, *overhead* enkripsi, dan metode *block* cipher. Namun, setelah titik tersebut, peningkatan *overhead* pemrosesan untuk chunk yang lebih besar mungkin mulai melebihi kemampuan paralelisasi, sehingga mengakibatkan penurunan kinerja.

V. KESIMPULAN

Berdasarkan hasil dan pembahasan dari perbandingan pengukuran kinerja antara algoritma AES dengan metode *Cipher Block Chaining* (CBC) dan algoritma ChaCha20-Poly1305. Dapat disimpulkan bahwa algoritma ChaCha20-Poly1305 memiliki kinerja lebih baik dibanding algoritma AES pada ukuran *chunk* yang lebih tinggi. Selain itu, ditemukan bahwa kedua algoritma menghasilkan kinerja maksimum pada ukuran *chunk* 100 KB.

ACKNOWLEDGEMENTS

Dengan ini, penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa atas ridho-Nya yang telah memberi penulis kesempatan untuk menyelesaikan kuliah Kriptografi dan Koding. Penulis juga ingin mengucapkan terima kasih sebesar-besarnya kepada Pak Rinaldi Munit selaku dosen mata kuliah Kriptografi dan Koding yang telah membimbing penulis dalam belajar dan memperluas wawasan penulis terhadap ilmu kriptografi. Terakhir, penulis ingin mengucapkan terima kasih kepada rekan-rekan belajar mata kuliah Kriptografi dan Koding yang telah senantiasa membantu penulis dalam suka dan duka dalam menjalani perkuliahan Kriptografi dan Koding.

REFERENCES

[1] Rogers, K. (2023, May 18). livestreaming. Encyclopedia Britannica. https://www.britannica.com/technology/livestreaming

[2] Twitch Statistics & charts · Twitchtracker. (n.d.). https://twitchtracker.com/statistics

[3] Abdullah, A. M. (2017). Advanced encryption standard (AES) algorithm to encrypt and decrypt data. Cryptography and Network Security, 16, 1-11.

[4] Almuhammadi, S., & Al-Hejri, I. (2017, April). A comparative analysis of AES common modes of operation. In 2017 IEEE 30th Canadian conference on electrical and computer engineering (CCECE) (pp. 1-4). IEEE.

[5] Bernstein, D. J. (2008, January). ChaCha, a variant of Salsa20. In Workshop record of SASC (Vol. 8, No. 1, pp. 3-5).

[6] Procter, G. (2014). A Security Analysis of the Composition of ChaCha20 and Poly1305. Cryptology ePrint Archive.

[7] Bernstein, D. J. (2005). The Poly1305-AES message-authentication code. In Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers 12 (pp. 32-49). Springer Berlin Heidelberg.

[8] Cryptography. PyPI. (n.d.). https://pypi.org/project/cryptography

PERNYATAAN

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

Bandung, 22 Mei 2023

Maulana Anindita A. A.