Analisis Kompleksitas pada Algoritma Enkripsi RSA dan AES dengan Optimasi Kode Huffman

Jonathan Harijadi - 13524144
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jonathan.harijadi@gmail.com , 13524144@std.stei.itb.ac.id

***Abstract*—Kriptografi adalah ilmu untuk mengamankan informasi dengan mengubahnya menjadi bentuk yang tidak bisa dibaca oleh orang lain. Cara untuk mengamankan informasi adalah melalui algoritma enkripsi. Beberapa algoritma enkripsi yang sering digunakan adalah algoritma Rivest–Shamir–Adleman (RSA) dan algoritma Advanced Encryption Standard (AES). Menggunakan Kode Huffman, sebuah teknik kompresi data, algoritma RSA dan AES dapat dioptimasi sehingga akan mengurangi jumlah data dan waktu yang digunakan. Makalah ini membahas perubahan pada kompleksitas algoritma pada RSA dan AES setelah digabungkan dengan Kode Huffman.**

***Kata Kunci*—Kriptografi, RSA, AES, Kode Huffman, Kompleksitas Algoritma.**

I. PENDAHULUAN

Dalam era digital saat ini, ada banyak data-data pribadi yang perlu diamankan. Salah satu cara untuk mengamankan data-data ini adalah menggunakan metode enkripsi. Dua metode enkripsi yang cukup popular adalah algoritma Rivest–Shamir–Adleman (RSA) dan algoritma Advanced Encryption Standard (AES). RSA adalah algoritma enkripsi asimetris yang menggunakan metode faktorisasi sehingga menghasilkan nilai yang sangat besar, sedangkan AES adalah algoritma enkripsi simetris yang menggunakan metode transformasi blok. Algoritma enkripsi ini biasanya akan menambah kompleksitasnya untuk menaikkan tingkat keamanannya. Selain itu data-data yang beredar pada era digital ini terus bertambah banyak. Oleh karena itu ukuran data yang perlu dienkripsi juga meningkat dengan pesat. Ukuran data yang besar tentu akan menambah waktu yang dibutuhkan untuk mengenkripsi data tersebut.

Salah satu cara untuk mengani masalah ini adalah dengan melakukan kompresi data terlebih dahulu sebelum dienkripsi. Dengan melakukan kompresi data terlebih dahulu, ukuran data yang perlu diproses algoritma enkripsi juga akan berkurang. Dalam makalah ini, algoritma kompresi data yang digunakan adalah algoritma kode Huffman. Kode Huffman merupakan algoritma kompresi data lossless yang artinya data hasil dekompresi akan sama dengan data sebelum di kompresi sehingga tidak ada data yang hilang. Makalah ini memilih algoritma kode Huffman dikarenakan penerapannya yang relatif lebih sederhana dibandingkan dengan algoritma kompresi data lain. Makalah ini bertujuan untuk mencari tahu seberapa besar pengaruh optimasi kode Huffman terhadap kompleksitas algoritma enkripsi RSA dan AES.

II. LANDASAN TEORI

A. Rivest–Shamir–Adleman (RSA)

Algoritma RSA adalah algoritma enkripsi asimetris. Pada tahun 1976 algoritma RSA dibuat oleh tiga peneliti dari MIT (Massachussets Institute of Technology) yang bernama Ron Rivest, Adi Shamir, dan Leonard Adleman. Dalam algoritma asimetris, kunci yang digunakan untuk mengenkripsi berbeda dengan kunci untuk mendekripsi data. Kunci-kunci ini dibuat menggunakan metode faktorisasi sehingga kunci yang dihasilkan akan memiliki nilai yang sangat besar. Berikut adalah alur kerja pembuatan kunci enkripsi dan dekripsi dari algoritma RSA:

1. Pilih 2 bilangan prima p dan q yang unik atau tidak sama, bilangan ini perlu dirahasiakan,
2. Hitung n yang berupa hasil kali antara p dan q (n = pq), nilai n ini tidak perlu dirahasiakan,
3. Hitung phi(n) yang berupa hasil kali antara p-1 dan q-1 (phi(n) = (p-1) * (q-1)), nilai phi(n) ini perlu dirahasiakan,
4. Pilih sebuah bilangan bulat e yang relatif prima terhadap nilai phi(n) atau Pembagi Bersama Terbesar(PBB) antara nilai e dengan phi(n) dan memenuhi syarat 1 < e < phi(n), nilai e akan menjadi kunci publik yang dibagikan,
5. Hitung kunci dekripsi d menggunakan kekongruenan e * d ≡ 1 (mod phi(n)) atau menggunakan persamaan,

$$d = \frac{1+km}{e} \quad (1)$$

Setelah mendapatkan kunci enkripsi, enkripsi data p menjadi *cipher text* c dapat dilakukan dengan persamaan ini,

$$c = p^e \bmod n \quad (2)$$

Sedangkan untuk dekripsi data menggunakan kunci d, dapat menggunakan persamaan ini,

$$p = c^d \bmod n \quad (3)$$

B. Miller-Rabin Primality Test

Algoritma Miller-Rabin Primality Test adalah algoritma probabilistik yang menguji apakah bilangan bulat positif yang diuji merupakan bilangan prima atau tidak. Algoritma ini berfungsi untuk bilangan bulat positif dengan nilai lebih dari tiga. Algoritma ini tidak dapat menjamin hasil yang diberikan akurat, tetapi jika dilakukan beberapa kali dengan basis yang berbeda maka hasil yang diberikan akan memiliki tingkat kepercayaan yang sangat tinggi. Algoritma ini memanfaatkan teorema Fermat, yang menyatakan bahwa jika p adalah bilangan prima dan a adalah bilangan bulat yang relatif prima terhadap p, maka:

$$a^{p-1} \equiv 1 \bmod p \quad (4).$$

Berikut adalah Langkah-langkah menjalankan algoritma Miller-Rabin Primality Test:

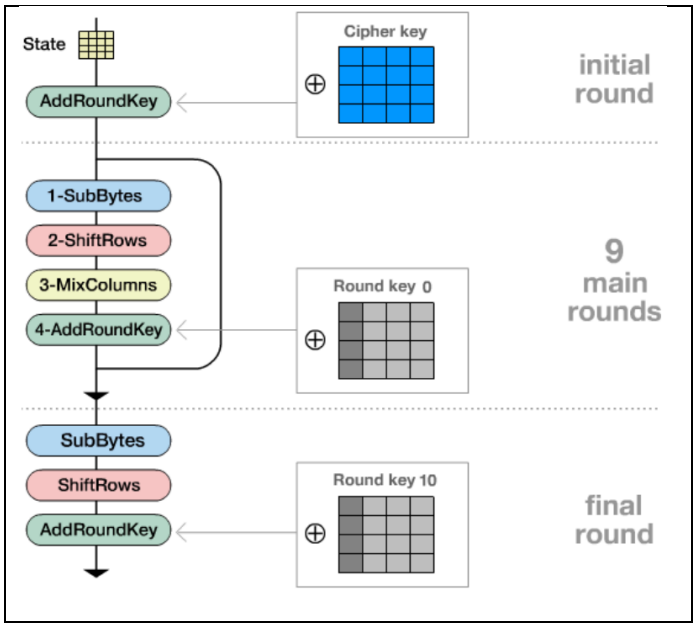
1. Mengecek apakah n ≤ 3, jika iya maka hentikan algoritma,
2. Tuliskan n dalam bentuk persamaan $n - 1 = 2^r \cdot d$ dengan d adalah bilangan ganjil,
3. Pilih bilangan acak a dengan syarat 2 < a < n – 1,
4. Periksa apakah $a^d \equiv \pm 1 \ (mod\ n)$.
5. Jika langkah ke 4 tidak terpenuhi maka n merupakan bilangan komposit, sedangkan jika langkah ke 4 terpenuhi maka bilangan n mungkin prima dan ulangi lagi dari langkah ke 3 dengan menggunakan a yang berbeda.

Langkah ke 5 atau langkah terakhir dapat terus diulangi hingga semua a di antara 2 dengan nilai n – 1 terpenuhi atau diulangi beberapa kali saja sampai kemungkinan bilangan n prima sangat tinggi. Pengulangan ini akan dilakukan hingga kemungkinan memberikan hasil yang salah atau *false positive*, paling besar hanya $\left(\frac{1}{4}\right)^k$, dengan k adalah jumlah pengulangan yang dilakukan.

C. Advanced Encryption Standard (AES)

Algoritma enkripsi AES dibuat oleh Vincent Rijmen dan Joan Daemen. Alogoritma AES merupakan algoritma enkripsi simetris yang menggunakan substitusi dan permutasi blok data. AES memiliki panjang kunci yang beragam seperti 128, 192, dan 256 bit. Tetapi, panjang kunci yang sering digunakan merupakan 128 bit dan 256 bit. Sedangkan kunci dengan panjang 192 bit sangat jarang digunakan.

Cara kerja AES adalah dengan mengelompokkan plainteks atau data lain yang telah dipecah menjadi bit-bit menjadi beberapa blok. Setiap blok pada AES memiliki ukuran total 128 bit yang disimpan dalam matriks 4x4 dengan bentuk heksadesimal. Proses AES memiliki 3 tahap utama yaitu tahap awal, tahap pengulangan, dan tahap akhir. Tahap pengulangan pada algoritma AES berubah sesuai dengan jumlah ronde yang akan dilakukan algoritma AES, tahap pengulangan akan memiliki 9 putaran jika menggunakan AES dengan 10 ronde.



Gambar 2.1. Tahap Enkripsi AES dengan 10 Ronde
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2014-2015/Makalah2-2015/Makalah2_Kripto_IF4020_2015_024.pdf)

Pada tahap awal, blok data yang telah disimpan sebelumnya akan di transformasi melalui operasi AddRoundKey. Proses ini akan melakukan penggabungan XOR antara blok data dengan *cipher key* yang ditentukan. Setelah operasi ini blok data akan diproses ke tahap yang selanjutnya yaitu tahap pengulangan.

Dalam tahap pengulangan, blok data akan ditransformasi dengan tiga operasi baru yaitu SubBytes, ShiftRows, dan MixColumns. Setelah tiga operasi tadi, transformasi blok data akan dilanjutkan dengan operasi AddRoundKey. Keempat operasi ini akan terus diungali hingga jumlah ronde AES yang dikurangi satu.

Operasi pertama yang dilakukan dalam tahap pengulangan adalah SubBytes. Pada operasi subbytes, setiap elemen dalam blok data akan ditukarkan dengan elemen yang ada dalam *S-box*. *S-box* ini merupakan suatu matriks 16x16 yang memiliki nilai byte. Nilai yang ditapatkan pada elemen blok akan dijadikan koordinat dalam matriks *S-box* dan ditukarkan dengan elemen yang ada pada koordinat tersebut. Contoh elemen blok data memiliki nilai 5a, maka nilai tersebut akan ditukarkan dengan nilai matriks *S-box* pada baris ke-5 dan kolom ke- a. Selanjutnya adalah operasi ShiftRows, pada operasi ini blok data akan dilakukan pergeseran ke kiri secara sirkular. Pergeseran ini akan dilakukan pada setiap baris dan diulangi sesuai dengan indeks baris tersebut, yaitu dilakukan 0 kali pada baris indeks 0 hingga dilakukan 3 kali pada baris indeks 3. Selanjutnya adalah operasi MixColumns, dalam operasi ini setiap kolom pada blok data akan dikalikan dengan polinom a(x) mod (x^4 +1). Terakhir adalah operasi AddRoundKey yang merupakan operasi XOR dengan round key. Round key yang digunakan merupakan hasil

dari operasi pengubahan pada *cipher key* atau *round key* sebelumnya.

Pada tahap terakhir, hasil transformasi blok data yang telah dilakukan pada tahp-tahap sebelumnya akan diubah dengan operasi SubBytes, ShiftRows, dan AddRoundKeys. Operasi ini dilakukan sama dengan operasi pada tahap pengulangan, tetapi hanya menghilangkan operasi MixColumns.

D. Kode Huffman

Algoritma kode Huffman ditemukan oleh David A. Huffman pada tahun 1950. Algoritma ini merupakan algoritma kompresi data lossless yang relatif lebih mudah untuk dipahami dan dilakukan dibandingkan dengan algoritma kompresi data lainnya. Algoritma kompresi data lossless merupakan algoritma yang mengecilkan ukuran suatu data tanpa menghilangkan data sama sekali.

Cara kerja kode Huffman ini cukup sederhana yaitu mengubah setiap karakter pada suatu data menjadi bentuk bit. Karakter yang sering muncul akan dijadikan bit yang lebih pendek, sedangkan karakter yang jarang muncul akan diberikan bit yang lebih panjang. Dibandingkan dengan encoding karakter pada umumnya seperti karakter American Standard Code for Information Interchange (ASCII) ataupun sandi morse, algoritma kode huffman dapat mengkonversi data dengan optimal untuk data yang akan dikompresi sehingga menghasilkan ukuran data yang jauh lebih kecil.

Algoritma kode huffman biasanya dibuat menggunakan konsep pohon. Alur kerja kode Huffman adalah sebagai berikut:

1. Catatlah frekuensi kemunculan masing-masing karakter pada data yang dikompresi termasuk spasi,
2. Pasangkan dua buah karakter dengan frekuensi kemunculan terendah sebagai daun dan membuat hasil gabungannya menjadi orangtua kedua daun tersebut dan simpul orangtua ini akan memiliki nilai frekuensi gabungan kedua daun tersebut,
3. Pilih karakter dengan frekuensi terkecil berikutnya dan taruh di sebelah kiri simpul orangtua dan gabungkan kedua daun ini menghasilkan simpul orangtua yang baru dengan total nilai frekuensi kedua daun tersebut,
4. Ulangi Langkah 2 dan 3 hingga seluruh karakter sudah diproses.

Setiap sisi yang di kiri pohon akan diberi tanda 0 dan sisi kanan akan diberi tanda 1. Representasi bit karakter tertentu dapat dicari dengan mengikuti alur pohon yang perlu dilalui dari puncak ke karakter yang dicari. Contoh 110 menunjukkan, untuk pergi ke karakter tersebut dari simpul puncak, perlu melalui ranting ke kanan dua kali lalu ke kiri sekali.

E. Kompleksitas Algoritma

Kompleksitas algoritma adalah jumlah waktu dan ruang memori yang dibutuhkan untuk menjalankan suatu algoritma. Kompleksitas algoritma dapat dibedakan menjadi kompleksitas waktu dan kompleksitas ruang. Pada makalah ini, hanya menggunakan konsep kompleksitas waktu saja.

Untuk mengukur kecepatan pertumbuhan suatu algoritma ketika ukuran data masukan membesar, biasanya menggunakan kompleksitas waktu asimptotik. Notasi kompleksitas yang biasa digunakan berupa Big-O (O), Big-Ω (Omega), dan Big-Θ (Theta). Big-O merupakan batas atas suatu algoritma atau proses yang perlu dilakukan pada skenario terburuk. Sedangkan Big- Ω merupakan batas bawak suatu algoritma atau proses yang perlu dilakukan pada skenario terbaik. Terakhir Big- Θ adalah rata-rata waktu yang dibutuhkan dalam menjalankan suatu algoritma. Penulisan kompleksitas waktu biasanya hanya menggunakan Big-O saja, dengan contoh; O(1) menunjukkan konstan, O(log n) bertumbuh secara logaritmik, O(n) bertumbuh secara linear, dan O(f(n)) yang bertumbuh sesuai f(n).

III. ANALISA DAN PEMBAHASAN

A. Kompleksitas Awal

Agar dapat dibandingkan, kompleksitas algoritma RSA dan AES harus dihitung terlebih dahulu sebelum dilakukan optimasi menggunakan kompresi data kode huffman. Kompleksitas waktu algoritma kode huffman juga perlu diketahui sehingga dapat ditambahkan pada hasil akhir algoritma enkripsi setelah di optimasi dengan kode huffman. Dengan menghitung kompleksitas algoritma pada setiap tahap dalam proses enkripsi dan dekripsi, kompleksitas waktu kedua algoritma enkripsi dapat diketahui.

A.1. Kompleksitas Algoritma RSA

Pada algoritma RSA, alur kerja RSA dapat dibagi menjadi tiga yaitu tahap pembuatan kunci, tahap proses enkripsi, dan tahap proses dekripsi. Dalam proses pembuatan kunci akan dilakukan beberapa langkah berikut,

1. Pilih bilangan prima p dan q acak dengan $p \neq q$,
2. Menghitung $n = p * q$,
3. Menghitung $phi(n) = (p - 1) * (q - 1)$,
4. Pilih bilangan kunci publik e, e relatif prima dengan phi(n), $1 < e < phi(n)$,
5. Hitung bilangan kunci privat d, menggunakan invers modulus e mod phi(n) atau persamaan $d = \frac{1+km}{e}$.

Dalam tahap pembuatan kunci enkripsi dan kunci dekripsi, hal yang paling banyak memakan waktu adalah langkah pertama. Pada langkah pertama perlu dilakukan pengecekan apakah suatu bilangan acak yang dibuat merupakan prima atau tidak. Pengecekan ini dapat digunakan menggunakan algoritma Miller-Rabin Primality Test. Kompleksitas waktu untuk algoritma Miller-Rabin Primality Test adalah $O(k * (\log n)^3)$ dengan nilai k adalah jumlah pengulangan yang dilakukan algoritma miller-rabin dengan basis berbeda, biasanya nilai k sangat kecil sehingga secara asimtotik, k sering dianggap konstanta. Oleh karena itu kompleksitas waktu algoritma miller-rabin sering ditulis $O((\log n)^3)$.

Selanjutnya pada langkah ke 2 dan ke 3 nilai p dan q biasanya memiliki nilai yang sangat besar sehingga komputer tidak dapat mengalikan kedua bilangan sekaligus. Oleh karena

itu, p dan q yang masing-masing memiliki Panjang bit sekitar n/2 akan memiliki kompleksitas waktu O($\left(\frac{n}{2}\right)^2$) atau ditulis O($n^2$).

Terakhir pada pencarian kunci e dan kunci d, seperti halnya dengan masalah pada pencarian nilai n dan m, nilai p dan q sangatlah besar sehingga tidak efisien atau bahkan bisa salah jika menggunakan persamaan $d = \frac{1+km}{e}$. Untuk mencari kunci e sebaiknya memilih nilai e secara acak lalu diperiksa menggunakan algoritma Euclidean untuk menentukan apakah nilai e relatif prima dengan phi(n). Kompleksitas waktu algoritma euclidean adalah O(n^2), sehingga waktu kompleksitas total untuk mencari kunci e adalah O(k * n^2) dengan k merupakan jumlah pengulangan algoritma euclidean hingga mendapat e yang sesuai. Tetapi, dikarenakan nilai k sangat sedikit, K dapat dianggap sebagai konstanta sehingga kompleksitas waktu mencari kunci e adalah O(n^2). Sama halnya untuk kunci d, kunci d dapat dicari menggunakan Extended Euclidean Algorithm untuk mencari invers modulo dari persamaan e * d ≡ 1 (mod phi(n)). Kompleksitas waktu dari Extended Euclidean Algorithm juga sebanding dengan algoritma euclidean yaitu O(n^2).

Dari kompleksitas waktu setiap langkahnya dapat disimpulkan bahwa kompleksitas pembuatan kunci enkripsi sekaligus kunci dekripsi adalah O(n^3). Hal ini dikarenakan tahap pertama yang memiliki kompleksitas waktu O(n^3), adalah tahap dengan kompleksitas waktu yang paling tinggi dibandingkan dengan kompleksitas waktu langkah yang lain. Selanjutnya adalah proses enkripsi dalam algoritma RSA.

Pada proses enkripsi algoritma RSA, digunakan persamaan $c = p^e \ mod \ n$, untuk mengubah p yang merupakan data yang ingin dienkripsi menjadi c atau *cipher text*. Kompleksitas waktu eksponensiasi modular ($p^e \ mod \ n$) adalah O((log N)²) dan kompleksitas jumlah perkalian modular adalah O(log e). Tetapi, log N dapat diubah menjadi n, jika n merupakan panjang modulus N dan log e dapat diubah juga menjadi n, dikarenakan e bisa memiliki panjang bit yang sama dengan N. Setelah perubahan ini, kompleksitas waktu proses enkripsi algoritma RSA berubah menjadi O(n * n^2) atau O(n^3).

Selanjutnya adalah proses dekripsi algoritma RSA, persamaan yang digunakan mirip dengan saat proses enkripsi yaitu $p = c^d \ mod \ n$. Sama halnya dengan saat proses enkripsi, pada proses dekripsi dilakukan eksponensiasi modular yang menghasilkan kompleksitas waktu total O(n^3) untuk seluruh proses dekripsinya.

A.2. Kompleksitas Algoritma AES

Dalam algoritma AES terdapat 4 operasi yang dapat dilakukan yaitu SubBytes, ShiftRows, MixColumns, dan AddRoundKey. Proses enkripsi dan dekripsi pada algoritma AES hanya menggunakan keempat operasi itu saja dengan perbedaan utama antara proses enkripsi dengan proses dekripsi adalah perbedaan urutan penggunaan keempat operasi ini. Pada tahap awal proses enkripsi hanya menggunakan operasi AddRoundKey, lalu pada tahap pengulangan digunakan keempat operasi tersebut dengan urutan sebagai berikut, SubBytes, ShiftRows, MixColumns, lalu AddRoundKey. Pada tahap akhir, operasi yang digunakan hanya 3 yaitu SubBytes, ShiftRows, dan AddRoundKey. Untuk proses dekripsi algoritma AES, pada dasarnya hanya membalikkan proses enkripsi sehingga kompleksitas waktu proses dekripsi dan kompleksitas waktu proses enkripsi harusnya sama. Berikut adalah kompleksitas waktu masing-masing operasi:

1. SubBytes: operasi ini menukar nilai setiap blok dengan nilai yang ada pada *S-box* menggunakan address matriks dari nilai sebelum ditukar, kompleksitas waktu operasi ini adalah O(1),
2. ShiftRows: operasi ini melakukan pergeseran baris ke kiri secara sirkular dengan jumlah pergeseran sesuai dengan indeks baris dari 0 sampai 3 karena blok merupakan matris 4x4, kompleksitas waktu operasi ini adalah O(1),
3. MixColumns: operasi ini mengalikan setiap kolom pada blok data dengan polinom a(x) mod (x^4 +1), kompleksitas operasi ini juga O(1),
4. AddRoundKey: Operasi ini melakukan XOR antara blok data dengan *round key* yang diubah seiring putaran terjadi, kompleksitas waktu operasi ini adalah O(1).

Kompleksitas waktu total algoritma AES adalah O(R * 1) dengan R merupakan jumlah ronde atau putaran algoritma AES. Tetapi, ronde algoritma AES sudah ditentukan dari awal sehingga dianggap sebagai konstanta yang menyebabkan kompleksitas algoritma AES per bloknya adalah O(1). Jika ukuran data yang dienkripsi adalah n maka kompleksitas waktu proses enkripsi atau proses dekripsi algoritma AES adalah O(n/1) atau bisa ditulis O(n) (linear terhadap ukuran data).

A.3. Kompleksitas Algoritma Kode Huffman

Dalam algoritma kode huffman, alur kerja algoritama kode huffman dapat dibagi menjadi tiga tahap. Ketiga tahap tersebut adalah tahapa pembuatan pohon, tahap proses enkripsi, dan tahap proses dekripsi. Pada tahap pembuatan pohon, dilakukan beberapa langkah berikut:

1. Menghitung frekuensi kemunculan setiap karakter, kompleksitas waktu langkah ini adalah O(n) dikarenakan perlu memindai setiap karakter pada suatu data,
2. Membuat prioritas queue, kompleksitas waktu langkah ini adalah (k * log k) dengan k adalah jumlah karakter unik pada data yang di enkripsi,
3. Membuat pohon dengan mengambil dua simpul frekuensi terendah, menggabungkannya, dan memasukkan simpul baru sampai hanya tersisa satu simpul(akar), kompleksitas waktu langkah ini adalah O(k * log k),
4. Membangun tabel kode untuk setiap karakter unik, kompleksitas waktu langkah ini adalah O(k^2) pada

kondisi terburuk yaitu pohon terdistorsi dan O(k * log k) pada kondisi terbaik yaitu pohon seimbang.

Secara keseluruhan pada tahap pembangunan pohon memiliki kompleksitas waktu O(n + k log k) dengan k adalah jumlah karakter unik. Nilai k ini biasanya merupakan konstanta contoh $k \leq 256$ untuk teks ASCII, sehingga kompleksitas waktu dapat diubah menjadi O(n).

Selanjutnya adalah proses enkripsi, dalam proses enkripsi kompleksitas waktu yang dibutuhkan untuk menuliskan setiap karakter adalah O(1). Hal ini dilakukan untuk setiap karakter pada data yang di enkripsi sehingga kompleksitas waktunya menjadi O(n). Hal yang sama juga terjadi pada proses dekripsi. Kompleksitas waktu untuk melakukan proses dekripsi pada sebuah karakter adalah O(1) dengan melihat tabel kode yang sudah dibuat. Dengan melakukan proses dekripsi ini untuk setiap karakter yang di enkripsi, kompleksitas waktunya berubah menjadi O(n).

Melihat semua tahap pada algoritma kode huffman memiliki kompleksitas waktu yang sama, dapat dipastikan bahwa kompleksitas waktu algoritma kode huffman adalah O(n).

B. Kompleksitas Algoritma Setelah Optimasi Kode Huffman

Saat melakukan optimasi algoritma enkripsi menggunakan algoritma kompresi, sebaiknya menggunakan algoritma kompresi terlebih dahulu sebelum melakukan proses enkripsi. Hal ini terjadi karena cara kerja algoritma kompresi menggunakan pola statistik dalam data yang akan di kompresi. Pola statistik yang ada pada data akan hilang setelah dilakukan proses enkripsi terhadap data tersebut, sehingga pola statistik data tersebut berubah menjadi pola acak dan tidak beraturan dan membuat algoritma kompresi tidak dapat bekerja dengan maksimal.

Untuk mengetahui kompleksitas algoritma setelah dilakukan optimasi, perlu dicari kompleksitas algoritma pada setiap langkahnya. Dari awal kompresi data hingga proses enkripsi dan proses dekripsi hingga akhirnya proses dekompresi. Alur kerja untuk algoritma enkripsi RSA dengan optimasi algoritma kompresi kode huffman adalah sebagai berikut:

1. Kompresi data menggunakan algoritma huffman, kompleksitas waktu langkah ini adalah $O(N_{original})$ dengan $N_{original}$ adalah ukuran data orginal,
2. Enkripsi data terkompresi dengan menggunakan algoritma RSA, kompleksitas waktu langkah ini adalah $O(N_{compressed} * n^2)$ dengan $N_{compressed}$ adalah ukuran data setelah dikompresi,
3. Dekripsi data terkompresi dengan menggunakan algoritma RSA, kompleksitas waktu langkah ini adalah $O(N_{compressed} * n^2)$ dengan $N_{compressed}$ adalah ukuran data setelah dikompresi,
4. Dekompresi data menggunakan algoritma huffman, kompleksitas waktu langkah ini adalah $O(N_{original})$, dengan $N_{original}$ adalah ukuran data original,

Kompleksitas waktu total pada algoritma RSA yang sudah dioptimasi menunjukkan perubahan dari $O(n^3)$ menjadi $O(N_{compressed} * n^2)$ pada proses enkripsi dan proses dekripsi. Tetapi, jika menggunakan data yang sangat besar maka kompleksitas waktunya akan tetap didominasi $O(n^3)$. Selain itu perlu ditambahkan $O(N_{original})$ untuk proses kompresi dan dekompresi data.

Selanjutnya adalah optimasi algoritma enkripsi AES dengan algoritma kompresi kode huffman. Alur kerja algoritma AES yang telah dioptimasi adalah sebagai berikut:

1. Kompresi data menggunakan algoritma kode huffman, kompleksitas waktu langkah ini adalah $O(N_{original})$ dengan $N_{original}$ adalah ukuran data original,
2. Enkripsi data terkompresi menggunakan algoritma AES, kompleksitas waktu langkah ini adalah $O(N_{compressed})$,
3. Dekripsi data terkompresi menggunakan algoritma AES, kompleksitas waktu langkah ini adalah $O(N_{compressed})$,
4. Dekompresi data menggunakan algoritma kode huffman, kompleksitas waktu langkah ini adalah $O(N_{original})$ dengan $N_{original}$ adalah ukuran data original.

Kompleksitas waktu total pada algoritma AES yang telah dioptimasi menunjukkan perubahan dari O(n) menjadi $O(N_{compressed})$ pada proses enkripsi dan proses dekripsi. Tetapi, proses kompresi dan proses dekompresi menambahkan langkah baru yang memiliki kompleksitas waktu $O(N_{original})$ sehingga perlu ditambahkan juga.

IV. KESIMPULAN DAN SARAN

Penggabungan algoritma kompresi data kode huffman dengan algoritma enkripsi RSA dan AES tidak mengubah kompleksitas algoritma waktu secara signifikan. Hal yang berubah setelah dilakukan kompresi data hanyalah jumlah data yang perlu diproses saat melakukan proses enkripsi dan proses dekripsi.

Algoritma/ Kombinasi	Tahap/ Proses	Kompleksitas Waktu	Keterangan
RSA	Pembuatan Kunci	$O(n^3)$	n: panjang bit kunci RSA
	Enkripsi / Dekripsi	$O(n^3)$	Dominan untuk operasi pada bilangan besar
AES	Pembuatan Kunci	O(1)	Konstan
	Enkripsi / Dekripsi	O(n)	n: panjang data dalam bit, sangat efisien
Huffman	Pembuatan Pohon	O(n + k log k)	n: ukuran data, k: jumlah karakter unik
	Kompresi / Dekompresi	O(n)	
RSA + Huffman	Total Sistem Hibrida	$O(N_{compressed} * n^2)$	Sangat tidak praktis untuk data besar, didominasi oleh $O(N_{compressed} * n^2)$
AES + Huffman	Total Sistem	$O(N_{original})$	Sangat efisien, kompresi mengurangi data AES

Tabel 1. Kompleksitas Waktu Algoritma RSA, AES, Kode Huffman, dan Penggabungannya

Dapat dilihat pada tabel 1, kompleksitas waktu $O(N_{original})$ dari proses kompresi dan dekompresi tidak terlihat pada gabungan RSA dengan kode Huffman. Hal ini terjadi karena $O(N_{original})$ didominasi oleh $O(N_{compressed} * n^2)$ yang merupakan kompleksitas waktu proses enkripsi dan dekripsinya. Proses kompresi/dekompresi yang didominasi proses enkripsi/dekripsi menunjukkan kurang cocoknya penggunaan algoritma kompresi data kode huffman dengan algoritma enkripsi RSA. Algoritma RSA biasa digunakan hanya untuk data-data berukuran kecil dikarenakan memiliki kompleksitas waktu yang sangat tinggi berupa eksponensial sehingga kegunaan fungsi kompresi data seperti kode Huffman kurang efektif digunakan. Walaupun kode Huffman membantu mengurangi data yang perlu diproses, tetapi kurang cocok untuk digabungkan dengan algoritma RSA.

Untuk kompleksitas waktu gabungan algoritma AES dengan kode huffman, menurut tabel 1 kompleksitas waktu proses kompresi/dekompresi ($O(N_{original})$) mendominasi kompleksitas waktu proses enkripsi/dekripsi ($O(N_{compressed})$) dikarenakan $N_{original} \geq N_{compressed}$. Hal ini menunjukkan algoritma kompresi data seperti algoritma kode huffman sangat cocok untuk algoritma enkripsi AES. Walaupun proses kompresi dan dekompresi menambah langkah baru yang harus dijalankan, tetapi total waktu yang dipakai akan berkurang secara drastis sesuai dengan persentase data yang berkurang. Oleh karena itu, algoritma AES yang telah dioptimasi kode huffman memiliki durasi waktu yang jauh lebih kecil daripada algoritma AES yang belum digabungkan dengan algoritma kode huffman.

REFERENCES

[1] Ophie, Edmund. "Optimasi Enkripsi Teks Menggunakan AES dengan Algoritma Kompresi Huffman", https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2014-2015/Makalah2-2015/Makalah2_Kripto_IF4020_2015_024.pdf, diakses 20 Juni 2025.

[2] Simbolon, Jeremy S.O.N. "On Miller-Rabin Primality Test and Its Worst Witnesses", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/Makalah2021/Makalah-Matdis-2021%20%2847%29.pdf, diakses 20 Juni 2025.

[3] Munir, Rinaldi. 2024. "Kompleksitas algoritma (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf, diakses 20 Juni 2025.

[4] Munir, Rinaldi. 2024. "Kompleksitas algoritma (Bagian 2)" https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf, diakses 20 Juni 2025.

[5] Muhamad Zulfikar, Tapha Imanuddin, Riyan Anugrah Alhad, Nova Eko Prastyo, and Satriya Adi Firmansyah, "Analisis Tingkat Kompleksitas Waktu dan Tingkat Keamanan Enkripsi Pada Algoritma Kriptografi RSA, DES, AES", *LPPM. ITEBA*, vol. 2, no. 2, Jan. 2024.

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, 1 Juni 2025

Jonathan Harijadi, 13524144