Analisis Kompleksitas Algoritma RSA dan ECC pada Aplikasi Pengirim Pesan

Leonardus Brain Fatolosja - 13524146
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: Leonardusbrain0@gmail.com , 13524146@std.stei.itb.ac.id

***Abstract*—Di era digital saat ini, keamanan privasi menjadi sesuatu yang krusial. Banyaknya kasus penyadapan dan kebocoran data kerap membuat banyak pihak menjadi waspada. Sebagai bentuk perlindungan, kriptografi berperan penting dalam menjaga informasi dan kerahasiaan data. Kriptografi asimetris (*public key cryptography)* adalah salah satu bentuk kriptografi yang digunakan secara masif. RSA dan ECC adalah salah satu bentuk kriptografi asimetris yang berperan penting dalam proses enkripsi dan dekripsi sebuah informasi.**

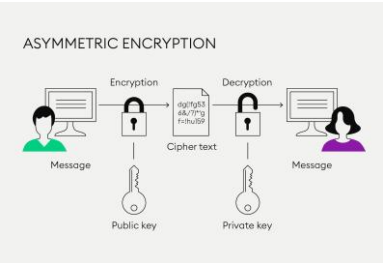
***Kata kunci*—Kriptografi, RSA, ECC, Kompleksitas Algoritma.**

I. PENDAHULUAN

Seiring berkembangnya zaman dan teknologi, proses berbagi pesan ke seluruh dunia menjadi sebuah kegiatan yang mudah dilakukan. Kini, pengguna bisa saling mengirim pesan hanya melalui gadget melalui internet. Tidak hanya itu, sekarang banyak sekali aplikasi atau *platform* yang menyediakan fasilitas untuk pengguna bisa saling mengirim pesan.

Namun, sejalan dengan berkembangnya teknologi berbagi pesan, muncul pula kekhawatiran akan privasi pesan yang dikirim. Banyak pihak yang mengkhawatirkan pesan yang dikirimnya dapat dibaca oleh aplikasi atau pun pihak lain yang tidak bertanggung jawab. Oleh karena itu, aplikasi-aplikasi pengirim pesan sekarang menggunakan enkripsi untuk pesan-pesan yang dikirim melalui aplikasi mereka.

Terdapat berbagai metode enkripsi di dunia, terutama untuk kriptografi pesan. Dalam makalah ini, metode yang akan dibahas secara lebih spesifik adalah kriptografi asimetris.



Gambar 1.1. Ilustrasi Kriptografi Asimetris
(sumber: https://www.bitpanda.com/academy/en/lessons/what-is-asymmetric-encryption/)

Kriptografi asimetris adalah jenis kriptografi yang memanfaatkan dua kunci yang berbeda antara enkripsi dan dekripsi. Pada jenis ini, kunci untuk enkripsi disebarkan kepada umum atau disebut kunci publik (*public key*), sedangkan kunci dekripsi tidak disebarkan kepada umum atau disebut kunci pribadi (*private key*). Oleh karena itu, metode ini sering disebut dengan kriptografi kunci publik (*public key cryptography*).

Contoh dari kriptografi asimetris yang akan dibahas lebih jauh dalam makalah ini adalah RSA dan ECC. RSA (Rivest-Shamir-Adleman) adalah kriptografi asimetris yang menggunakan metode faktorisasi bilangan prima besar, sedangkan ECC (*Elliptic Curve Cryptography*) adalah kriptografi asimetris yang menggunakan prinsip matematika kurva eliptik.

Kedua metode enkripsi tersebut mempunyai karakteristik yang berbeda, mulai dari efisiensi, keamanan, mau pun ukuran kunci. Dengan perbedaan itu, tentu ada algoritma enkripsi yang lebih efisien dan lebih aman. Untuk menilai efisiensi dari algoritma enkripsi RSA dan ECC, dapat digunakan pendekatan melalui analisis kompleksitas algoritma untuk menilai kecepatan dari kedua algoritma tersebut. Dengan menilai aspek kecepatan algoritma dan tingkat keamanan antara RSA dengan ECC, dapat ditentukan algoritma enkripsi yang lebih efisien, aman, dan relevan untuk kebutuhan kriptografi di masa sekarang. Tidak seperti RSA, ECC tidak secara langsung digunakan untuk enkripsi dan dekripsi karena lebih umum dimanfaatkan dalam mekanisme *key aggrement* dan tanda tangan digital. Aplikasi ECC yang akan dibahas pada makalah ini dan dibandingkan dengan RSA adalah ElGamal ECC, yaitu ElGamal yang menggunakan prinsip ECC dalam proses enkripsi dan dekripsi.

II. LANDASAN TEORI

A. *Kritografi Asimetris*

Kriptografi adalah ilmu yang mempelajari praktik dan cara untuk melindungi sebuah informasi melalui algoritma-algoritma enkripsi. Enkripsi akan mengubah informasi menjadi bentuk yang sudah terenkripsi agar tidak dapat dibaca oleh pihak lain. Sehingga selain melakukan enkripsi untuk melindungi pesna yang dikirim, proses dekripsi juga menjadi bagian dari

kriptografi untuk memecahkan informasi yang terenkripsi agar dapat dibaca oleh penerima.

Kriptografi terbagi ke dalam dua jenis, yaitu kriptografi simetris dan asimetris. Kriptografi simetris adalah kriptografi yang memakai satu kunci untuk enkripsi dan dekripsi, sedangkan kriptografi asimetris adalah kriptografi yang menggunakan kunci yang berbeda untuk enkripsi dan dekripsi. Kriptografi asimetris memiliki banyak jenis, seperti RSA dan ECC.

Kriptografi asimetris memakai dua kunci yang berbeda untuk enkripsi dan dekripsi sehingga tidak memungkinkan untuk melakukan proses dekripsi dengan hanya mengetahui kunci enkripsi. Berbeda dengan proses enkripsi yang konvensional, enkripsi dengan kriptografi asimetris (*public-key encryption)* memungkinkan pengirim pesan dan penerima pesan mempunyai algoritma yang berbeda dan kunci yang berbeda pula. *Public key cryptography* sebetulnya terbagi ke dalam tiga kategori, yaitu enkripsi-dekripsi, tanda tangan digital, dan *key exchange*. Berawal dari prinsip kriptografi asimetris, banyak bermunculan pula algoritma-algoritma yang menggunakan prinsip tersebut. Beberapa di antaranya bisa melakukan ketiga kategori tersebut, tetapi ada juga yang hanya bisa melakukan satu atau pun dua. Pada kasus di makalah ini, ECC dan RSA bisa diperuntukkan untuk ketiga kategori tersebut.

B. RSA (Rivest-Shamir-Adleman)

RSA adalah salah satu bentuk kriptografi asimetris yang dibuat oleh Ron Rivest, Adi Shamir, dan Len Adleman. RSA memanfaatkan prinsip faktorisasi prima yang biasanya berupa bilangan besar untuk alasan keamanan dan modulo. Selain itu, RSA memiliki tiga prinsip penting dalam proses enkripsi-dekripsi secara keseluruhan, yaitu pembuatan kunci publik-privat, proses enkripsi, dan proses dekripsi.

1.) Proses pembuatan kunci publik-privat

Sebelum memulai proses enkripsi dan dekripsi, diperlukan adanya kunci publik dan privat. Kunci publik diperbolehkan untuk disebarkan ke publik, sedangkan kunci privat tidak diperbolehkan untuk disebarkan. Langkah pertama adalah menentukan dua bilangan prima yang umumnya berupa bilangan besar untuk alasan keamanan. Semisal bilangan p dan q adalah dua bilangan prima tersebut, maka langkah selanjutnya adalah menentukan n (hasil perkalian kedua bilangan prima), e (kunci untuk enkripsi), dan d (kunci untuk dekripsi).

$$n = p \,.\, q \tag{1}$$

Setelah mendapat nilai n, diperlukan adanya nilai Φ(n), yaitu banyaknya bilangan yang relatif prima dengan n di dalam rentang 1 sampai n-1.

$$\Phi(\mathrm{n}) = \Phi(\mathrm{p}.\mathrm{q}) = \Phi(\mathrm{p}).\Phi(\mathrm{q}) \tag{2}$$

$$\Phi(\mathrm{n}) = (p-1).(q-1) \tag{3}$$

Karena menurut fungsi totient euler, bilangan prima memiliki nilai Φ sebanyak bilangan tersebut dikurang satu. Oleh karena itu, Φ(n) bisa disingkat menjadi (p-1) . (q-1) karena p dan q merupakan dua bilangan prima yang membentuk n.

Untuk melakukan proses enkripsi, diperlukan bilangan e yang digunakan untuk enkripsi dan nantinya diperbolehkan untuk disebarkan ke publik. Pertama, bilangan e berada pada rentang nilai 1 sampai Φ(n). Φ(n) didapat dari persamaan ke-3.

$$1 < e < \Phi(\mathrm{n}) \tag{4}$$

Kemudian, nilai e akan diambil secara acak dengan syarat nilai e harus relatif prima dengan Φ(n). Artinya juga bahwa PBB dari e dan Φ(n) haruslah 1.

Untuk melakukan proses dekripsi, diperlukan bilangan d yang digunakan dalam proses dekripsi. Hal-hal yang diperlukan untuk mendapatkan nilai d adalah nilai e dan Φ(n). Persamaan berikut diperlukan untuk mendapatkan nilai d.

$$d \,.\, e = 1 \; mod \; \Phi(\mathrm{n}) \tag{5}$$

Artinya nilai d didapat dengan menghitung modulo invers dari e mod Φ. Nilai d memang bisa bervariasi, tetapi semuanya merupakan nilai yang benar karena akan mendekripsikan *ciphertext* ke pesan/informasi asli yang sama. Beberapa cara untuk menghitung modulo invers adalah teorema fermat dan *extended euclidean algorithm.*

Setelah mendapatkan nilai e dan d, kunci publik dan privat pun bisa dibuat. Kunci publik adalah (n,e) dan kunci privat adalah (n,d). Nilai n pada keduanya merupakan bilangan yang sama.

2.) Proses Enkripsi

Sebelum memulai proses enkripsi, nilai e harus didapatkan terlebih dahulu dari proses pembuatan kunci. Dengan mempunyai nilai e, proses enkripsi baru bisa dijalankan. Persamaan enkripsi dalam RSA sebenarnya cukup sederhana yang dituangkan dalam persamaan berikut ini.

$$C = M^e mod \; n \tag{1}$$

C adalah *ciphertext*, yaitu pesan atau informasi yang sudah dienkripsi. M adalah pesan asli dan e serta n adalah nilai yang sudah diketahui sebelumnya. RSA hanya mengubah angka dalam persamaan ini, sehingga untuk mengubah data lain, seperti huruf, diperlukan perubahan dari huruf menjadi angka. Pada kasus huruf, huruf harus diubah menjadi angka dengan skema-skema *encoding*, seperti ASCII.

3.) Proses Dekripsi

Proses dekripsi memerlukan *ciphertext*, nilai d, dan n. Dengan mempunyai semua persyaratan tersebut, proses dekripsi bisa dijalankan dengan persamaan berikut ini.

$$M = C^d \ mod \ n \tag{1}$$

Pada persamaan tersebut, C adalah *ciphertext* yang akan diubah ke dalam pesan aslinya, nilai d dan n adalah kunci privat, dan M adalah pesan asli yang didapatkan dengan mengubah nilai C. Sama seperti proses enkripsi, proses dekripsi juga hanya mengubah angka. Oleh karena itu, proses dekripsi huruf juga memerlukan skema *encoding* setelah angka aslinya, yaitu M sudah didapat.

C. *ECC (Eliptic Curve Cryptography)* dan ElGamal

ECC adalah salah satu bentuk kriptografi asimetris yang lebih baru dibandingkan dengan RSA. ECC dikembangkan oleh Neal Koblitz dan Victor S. Miller pada tahun 1985. ECC menawarkan keamanan yang setara dengan kunci yang lebih pendek dibandingkan dengan RSA yang membuat ECC dianggap lebih efisien. ECC juga dianggap lebih aman dibandingkan dengan RSA karena ECC memanfaatkan prinsip logaritma diskrit yang lebih sulit dipecahkan jika dibandingkan dengan faktorisasi bilangan besar yang dipakai dalam RSA. Oleh karena itu, ECC dianggap lebih aman dan efisien dibandingkan dengan RSA.

Berbeda dengan RSA, ECC tidak bisa langsung dipakai dalam proses enkripsi-dekripsi karena penggunannya tidak hanya sebatas pada itu. Selain enkripsi-dekripsi, ECC dipakai dalam tanda tangan digital dan *key agreement*. Oleh karena itu, untuk memanfaatkan prinsip ECC dalam proses enkripsi dan dekripsi, diperlukan sebuah algoritma yang dapat dimodifikasi sehingga memanfaatkan prinsip ECC.

Algoritma enkripsi dan dekripsi yang dapat dipakai untuk mengaplikasikan ECC adalah ElGamal. ElGamal dibuat oleh Taher ElGamal pada tahun 1985. ElGamal memanfaatkan prinsip logaritma diskrit dalam proses enkripsi dan dekripsi. ECC bisa diaplikasikan menggunakan ElGamal karena prinsipnya yang sama. ECC memanfaatkan permasalahan logaritma diskrit pada kurva eliptik atau sering disebut ECDLP (*Elliptic Curve Discrete Logarithm Problem)* sebagai prinsip matematikanya. Karena keduanya sama-sama memanfaatkan logaritma diskrit, sehingga ECC bisa diaplikasikan dengan menggunakan ElGamal. Oleh karena itu, selanjutnya proses enkripsi dan dekripsi yang dijelaskan akan berhubungan dengan pengaplikasian ECC dalam ElGamal atau sering disebut dengan *Eliptic Curve ElGamal*.

ElGamal menggunakan persamaan sederhana berikut sebagai dasar dari algoritmanya.

$$y = \ g^x (mod \ p) \tag{1}$$

Pada persamaan ini, X akan susah untuk dicari walaupun nilai Y, g, dan p sudah diketahui. Kunci publik dapat ditentukan dari persamaan tersebut, yaitu (Y,g,p) sebagai kuncinya dan X sebagai kunci privat. Setelah mendapatkan kunci publik, maka diperlukan nilai r sehingga dapat muncul persamaan berikut.

$$a = \ g^r (mod \ p) \tag{2}$$

$$b = Y^r . M \tag{3}$$

Bentuk a dan b adalah bentuk enkripsi/*ciphertext* dari pesan asli M. Untuk proses dekripsi, dapat dilakukan persamaan berikut ini.

$$M = \frac{b}{a^x} = \frac{Y^r . M}{g^{r.x}} = \frac{g^{r.x} . M}{g^{r.x}} = M \ (mod \ p) \tag{4}$$

M adalah bentuk asli dari *chipertext* yang dikirim dalam bentuk a dan b. Algoritma ElGamal memiliki prinsip yang berbeda dibandingkan dengan RSA.

ECC dengan prinsip kurva eliptiknya tentu dapat diaplikasikan dalam ElGamal. Dengan beberapa perubahan dan penyesuaian, ElGamal dapat diubah menjadi ECC. Perubahan yang dimaksud meliputi penggantian operasi perkalian modulo menjadi operasi titik pada kurva eliptik.

Langkah awal yang harus dibuat adalah menentukan kunci privat X, yaitu sebuah nilai acak dan membuat sebuah kunci publik dari kunci privat tersebut. Sebagai contoh, kunci Y bisa didapat dari persamaan berikut.

$$Y = X . G \tag{1}$$

Y adalah kunci publik berupa titik di dalam kurva eliptik yang ditentukan. G adalah sebuah titik pada kurva sebagai titik dasar / *generator point*. G didapat dengan syarat berada pada kurva, sehingga sebelum mendapatkan G, kurva eliptik harus dibuat terlebih dahulu dengan persamaan berikut.

$$Y^2 = X^3 + aX + b \ mod \ p \tag{2}$$

a dan b adalah nilai acak yang dipilih untuk membuat sebuah kurva eliptik acak, sedangkan p adalah bilangan prima yang dipilih secara acak pula. Dengan membuat sebuah kurva eliptik, titik dasar G bisa ditentukan dengan syarat berada pada kurva. Dengan mempunyai nilai-nilai ini, maka kita bisa melakukan proses enkripsi dan dekripsi.

Proses enkripsi dimulai dengan mempunyai pesan asli M. M dapat berupa angka atau pun huruf. Jika berupa huruf, maka harus dilakukan encoding agar menjadi angka dan semuanya seragam. Pesan tersebut kemudian akan dikonversi menjadi titik dalam kurva. M yang berupa titik dapat memiliki banyak macam skema *encoding,* termasuk membuat offset dan menaruh pesan pada koordinat X atau sering disebut *offset encoding*. Artinya jika tidak ada X sehingga dapat membuat titik (X,Y), maka akan ditambahkan offset dan nilainya disimpan untuk keperluan dekripsi. Bentuk enkripsi atau *ciphertext* dapat diambil dengan persamaan berikut ini.

$$C1 = r . G \tag{3}$$

$$C2 = r . Y + M \tag{4}$$

Nilai r adalah acak. C1 dan C2 adalah bentuk enkripsi dari pesan yang dikirim.

Selanjutnya untuk memulai proses dekripsi, C1 dan C2 diperlukan sebagai *ciphertext* dan kunci privat X. Pesan asli dapat diperoleh dengan persamaan berikut.

$$M = C2 - X.C1 \quad (5)$$

Dari kedua titik, maka akan didapat sebuah titik M yang bernilai (X,Y). Dengan skema *offset encoding* yang dipakai dalam enkripsi, koordinat X tersebut merupakan nilai pesan aslinya setelah dikurangi nilai *offset* tentunya. Nilai yang didapat bisa dilakukan *encoding* lanjutan untuk merubahnya ke bentuk yang diinginkan, seperti huruf.

D. Kompleksitas Algoritma

Kompleksitas Algoritma adalah bidang atau cara untuk menghitung performa sebuah algoritma, baik dari waktunya mau pun memorinya. Untuk membandingkan performa kecepatan, kompleksitas waktu dapat digunakan untuk menganalisis sebuah kode.

Pada suatu algortima, pertumbuhan kebutuhan waktu algoritmanya berjalan sejalan dengan pertumbuhan ukuran masukannnya. Kompleksitas waktu, biasa dilambangkan dengan T(n), berperan dalam menyediakan seberapa banyaknya waktu yang diperlukan untuk algoritma tersebut berjalan. Untuk masukan yang besar, notasi yang dipakai oleh kompleksitas waktu dinamakan kompleksitas waktu asimpotik.

Kompleksitas waktu asimptotik ada banyak macamnya, yaitu Big-O, Omega, dan Theta. Notasi O-besar atau Big-O memiliki definisi sebagai berikut.

DEFINISI 1. $T(n) = O(f(n))$ (dibaca "$T(n)$ adalah $O(f(n))$, yang artinya *$T(n)$ berorde paling besar $f(n)$*) bila terdapat konstanta C dan n_0 sedemikian sehingga

$$T(n) \leq C f(n)$$

untuk $n \geq n_0$.

Gambar 2.1 Definisi Big-O
(Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf)

Oleh karena itu, apa pun yang memenuhi persamaan tersebut merupakan Big-O dari algoritma itu. Namun, pada umumnya, Big-O akan dilambangkan dengan yang memiliki kecepatan tercepat. Contohnya adalah sebagai berikut.

$$T(n) = 2n^2 + 6n + 1 = O(n^2) = O(n^3)$$

Keduanya merupakan jawaban yang benar, tapi pada umumnya notasi big-O akan dipilih dari yang tercepat, yaitu $O(n^2)$.

Selain definisi dari big-O, notasi ini juga mempunyai beberapa operasi yang perlu diperhatikan.

TEOREMA 2. Misalkan $T_1(n) = O(f(n))$ dan $T_2(n) = O(g(n))$, maka
(*a*) $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$
(*b*) $T_1(n)T_2(n) = O(f(n))O(g(n)) = O(f(n)g(n))$
(*c*) $O(cf(n)) = O(f(n))$, c adalah konstanta
(*d*) $f(n) = O(f(n))$

Gambar 2.2 Operasi-operasi notasi Big-O
(Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf)

Dengan adanya teorema ini, perhitungan kompleksitas waktu akan menjadi lebih mudah.

Kedua, Omega juga merupakan notasi yang umum dipakai atau sering disebut sebagai notasi big-omega (**Ω**). Notasi ini memiliki definisi berikut ini.

Definisi 2. $T(n) = \Omega(g(n))$ (dibaca "$T(n)$ adalah Omega ($g(n)$" yang artinya *$T(n)$ berorde paling kecil $g(n)$*) bila terdapat konstanta C dan n_0 sedemikian sehingga $T(n) \geq C(g(n))$ untuk $n \geq n_0$.

Gambar 2.3 Definisi Big-Omega
(Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf)

Ketiga, Tetha adalah notasi yang umum dipakai juga dan dipakai jika ingin menunjukkan bahwa notasi Big-O dan Big-Omega dari suatu algoritma memiliki nilai yang sama. Oleh karena itu, dapat dikatakan bahwa Θ-besar memiliki definisi sebagai berikut.

Definisi 3. $T(n) = \Theta(h(n))$ (dibaca "$T(n)$ adalah tetha $h(n)$") yang artinya $T(n)$ berorde sama dengan $h(n)$ jika $T(n) = O(h(n))$ dan $T(n) = \Omega(h(n))$.

Gambar 2.4 Definisi Big-Tetha
(Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf)

E. Algoritma Euclidean dan modulo

Algoritma Euclidean adalah algoritma yang dipakai untuk mencari PBB (pembagi bilangan terbesar) antara dua angka. Algoritma ini jauh lebih efisien dibandingkan dengan mencari semua faktor dari masing-masing angka dan membandingkannya. Selain itu, algoritma ini lebih mudah untuk diimplementasikan dalam bentuk kode di komputer.

Algoritma Euclidean memuat modulo dalam tiap langkahnya. Tujuannya adalah untuk mencari sampai modulonya adalah 0. Langkah-langkahnya adalah sebagai berikut.

1. Semisal ada bilangan a dan b, dengan a lebih besar atau sama dengan b.
2. Persamaan $a = b\,.\,c + m$ akan dibuat dari bilangan a dan b dengan m adalah jarak antara perkalian b dan c dengan bilangan a.
3. Ulangi persamaan tersebut sampai m menjadi 0 dengan a diganti dengan b dan b diganti dengan m.
4. Ketika m sudah mencapai 0, maka PBB dari kedua bilangan a dan b di awal adalah nilai b pada akhir.

Dengan mengetahui dasar dari algoritma euclidean, PBB dari dua angka atau lebih dapat dicari dengan lebih cepat. Dengan pengetahuan akan dasarnya, algoritma ini bisa dipakai untuk beberapa hal lain, seperti untuk mencari modulo invers.

Modulo adalah sisa pembagian antara dua bilangan. Modulo bisa disajikan dengan persamaan $a = b\,.\,c + m$, dengan m adalah modulonya. Contohnya adalah sebagai berikut.

$$15\ mod\ 11 = 4, \qquad karena\ 11\,.\,1 + 4 = 15$$

Pada modulo, kekongruenan merupakan sebuah istilah yang sering dipakai. Kekongruenan merujuk pada sebuah nilai yang sama jika berada pada modulo tertentu. Contohnya adalah sebagai berikut.

$$5 \equiv 15\ (mod\ 10)$$

Hal tersebut benar karena 5 mod 10 = 15 mod 10. Oleh karena itu, bisa disimpulkan bahwa 5 kongruen dengan 15 dalam modulus 10. Selain itu, dengan definisi tersebut, $a \equiv b \ (mod\ m)$ akan menghasilkan persamaan lain, yaitu $a = b + k.m$. Dengan pengetahuan akan kekongruenan dan modulus, maka mencari modulo invers akan bisa dilakukan.

Mencari modulo invers tidak semudah mencari invers dari bilangan bulat. Sebuah invers dari modulo tertentu harus memenuhi persamaan berikut.

$$X.a = 1(mod\ m) \tag{1}$$

Jika ada suatu nilai X yang memenuhi persamaan tersebut, maka bisa dikatakan bahwa X adalah modulo invers dari a mod m. Ada banyak cara untuk menentukan modulo invers dengan dasar yang sama dan salah satunya adalah dengan menggunakan algoritma euclidean atau lebih tepatnya dinamakan *extended euclidean algorithm*. Langkah-langkah dan penjelasan mengenai algoritma euclidean untuk mencari modulo invers adalah sebagai berikut.

1. Dapat diketahui bahwa ketika ada $a\ mod\ m$ dan ingin mencari modulo inversnya yang memenuhi persamaan $ax + my = 1$, persamaan berikut merupakan sebuah kombinasi linear dari bilangan a dan m yang merupakan relatif prima dengan persyaratan PBB(a,m) = 1. Artinya juga modulo invers ada jika dan hanya jika a dan m di persamaan tersebut merupakan relatif prima.
2. Dengan memanfaatkan persamaan tersebut, dapat diciptakan sebuah algoritma euclidean untuk mencari modulo invers. Tujuannya adalah untuk mencari X yang mengakibatkan perkaliannya dengan a dan pertambahannya dengan m dan y menghasilkan nilai 1.
3. Hitung PBB dari a dan m dengan algoritma euclidean biasa. Jika bukan satu, artinya tidak ada modulo invers yang tersedia.
4. Lakukan substitusi balik dari angka satu hingga mencapai kombinasi linear $ax + my = 1$.
5. X diambil sebagai modulo invers dari a mod m.

III. Implementasi Algoritma

Untuk menganalisis kompleksitas algoritma RSA dan ECC, diperlukan implementasi nyata dari kedua algoritma tersebut. Dengan membuat sebuah simulasi sederhana, algoritma RSA dan ECC bisa dianalisis secara lebih sistematis dan mendalam. Pembuatan simulasi ini menggunakan bahasa pemrograman C supaya lebih terstruktur.

A. Algoritma RSA

Sebelum membuat algoritmanya, diperlukan adanya *library* bawaan dari C, yaitu stdio.h, stdlib.h, dan math.h. Tujuan ditambahkannya *library* tersebut adalah untuk memudahkan penulisan kode.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

Gambar 3.1 Dokumentasi Code – Library

Setelah itu, fungsi pertama yang dibuat adalah untuk mencari PBB. Kode di bawah merupakan implementasi algoritma euclidean untuk mencari pembagi bilangan terbesar antara bilangan a dan b.

```c
/*Mencari PBB*/
int pbb(int a, int b){
    int mod;
    while (b != 0){
        mod = a % b;
        a = b;
        b = mod;
    }
    return a;
}
```

Gambar 3.2 Dokumentasi Code – PBB

Fungsi kedua yang dibuat adalah fungsi untuk mencari modulo invers dari e mod phi.

```c
/*Menghitung mod inverse dari e*/
int inverse(int e, int phi){
    int a = 0, na = 1;
    int b = phi, nb = e;
    int q,temp;
    while (nb != 0) {
        q = b / nb;
        temp = na;
        na = a - q * na;
        a = temp;

        temp = nb;
        nb = b - q * nb;
        b = temp;
    }
    if (a < 0) a += phi;
    return a;
}
```

Gambar 3.3 Dokumentasi Code – Modulo Invers

Fungsi terakhir yang dibuat adalah fungsi untuk melakukan perhitungan yang ada pada proses enkripsi dan dekripsi, yaitu sebuah base (pesan atau *ciphertext*) yang dipangkatkan suatu nilai (e atau d), dan dikalikan dengan modulo n (dalam kode bernilai mod).

```c
int calculation(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) result = (result * base) % mod;
        base = (base * base) % mod;
        exp = exp / 2;
    }
    return result;
}
```

Gambar 3.4 Dokumentasi Code – Perhitungan untuk enkripsi dan dekripsi

Pada gambar di bawah ini, terdapat program utama sebagai simulasi sederhana sebuah aplikasi berbagi pesan, yaitu ada masukan yang meminta sebuah kalimat. Selanjutnya, kalimat tersebut akan dienkripsi menggunakan RSA dan hasil enkripsinya akan ditampilkan. Terakhir, hasil enkripsi akan dilakukan proses dekripsi dan hasil dekripsinya ditampilkan dalam bentuk *character*.

```c
int main(){
    int p,q,phi,e,d,n;
    /*p dan q dipilih dari awal karena hanya simulasi.
*/
    p = 61;
    q = 47;
    n = p * q;
    phi = (p - 1) * (q - 1);
    /*Menentukan e, yaitu kunci publik*/
    e = 13; //Relatif prima dengan Φ(n)

    /*Menentukan d, yaitu kunci privat*/
    d = inverse(e, phi);

    char input[256];
    printf("Masukkan pesan: ");
    fgets(input, sizeof(input), stdin);

    int ascii[256];
    //Merubah huruf ke ascii
    for (int i=0;i<strlen(input);i++){
        ascii[i] = (int)input[i];
    }
    /*Enkripsi*/
    int encrypted[256];
    for (int i=0;i<strlen(input);i++){
        encrypted[i] = calculation(ascii[i],e,n);
        printf("Encrypted %d : %d",i,encrypted[i]);
        printf("\n");
    }

    /*Dekripsi*/
    int decrypted[256];
    for (int i=0;i<strlen(input);i++){
        decrypted[i] = calculation(encrypted[i],d,n);
    }

    printf("Pesan baru: ");
    for (int i=0;i<strlen(input);i++){
        printf("%c",(char)decrypted[i]);
    }
}
```

Gambar 3.5 Dokumentasi Code – Program utama RSA

Hal pertama yang dilakukan kode adalah melakukan pembuatan kunci privat dan publik. Selanjutnya, pengguna akan diminta memasukkan pesan yang ingin dikirim. Pesan tersebut diubah menjadi bentuk ASCII dan dienkripsi menggunakan RSA dalam baris ke-26. Hasil enkripsi langsung ditampilkan dan disimpan pada *array* encrypted. *Array* tersebut didekripsikan pada baris ke-34 dan ditampilkan kembali dalam bentuk *character*. Hasil program di atas tertera pada gambar di bawah ini.

```
Masukkan pesan: Makalah Matematika Diskrit!!!
Encrypted 0 : 2496
Encrypted 1 : 553
Encrypted 2 : 1997
Encrypted 3 : 553
Encrypted 4 : 1572
Encrypted 5 : 553
Encrypted 6 : 1213
Encrypted 7 : 520
Encrypted 8 : 2496
Encrypted 9 : 553
Encrypted 10 : 2381
Encrypted 11 : 589
Encrypted 12 : 2488
Encrypted 13 : 553
Encrypted 14 : 2381
Encrypted 15 : 1429
Encrypted 16 : 1997
Encrypted 17 : 553
Encrypted 18 : 520
Encrypted 19 : 665
Encrypted 20 : 1429
Encrypted 21 : 2263
Encrypted 22 : 1997
Encrypted 23 : 2647
Encrypted 24 : 1429
Encrypted 25 : 2381
Encrypted 26 : 2188
Encrypted 27 : 2188
Encrypted 28 : 2188
Encrypted 29 : 1495
Pesan baru: Makalah Matematika Diskrit!!!
```

Gambar 3.6 Dokumentasi Code – Hasil Simulasi Sederhana Aplikasi Berbagi Pesan

B. Algoritma ElGamal ECC (Eliptic Curve ElGamal)

Simulasi kedua yang dilakukan adalah menggunakan algoritma enkripsi dan dekripsi ECC ElGamal dalam simulasi sederhana aplikasi berbagi pesan.

Pada gambar 3.7, ditambahkan beberapa *library* untuk membantu penulisan kode dan algoritma. Selain *library,* ada pembuatan variabel global, yaitu PRIME untuk nilai p, AA untuk nilai A pada kurva eliptik, dan BB untuk nilai B pada kurva eliptik. Terakhir, ada pembuatan ADT Point untuk menyimpan nilai titik.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PRIME 751
#define AA 2
#define BB 3

typedef struct {
    int x;
    int y;
    int is_infinity;
} Point;
```

Gambar 3.7 Dokumentasi Code – Library, Variabel Global, dan ADT Point

Pada gambar 3.8, fungsi untuk mencari modulo invers seperti pada RSA ditambahkan pula dengan algoritma yang sama.

```c
/*Mod Inverse untuk ECC*/
int inverse(int e, int phi){
    int t = 0, newt = 1;
    int r = phi, newr = e;
    while (newr != 0) {
        int quotient = r / newr;
        int temp = newt;
        newt = t - quotient * newt;
        t = temp;

        temp = newr;
        newr = r - quotient * newr;
        r = temp;
    }
    if (t < 0)
        t += phi;
    return t;
}
```

Gambar 3.8 Dokumentasi Code – Modulo Invers

Pada gambar 3.9, fungsi yang dibuat bertujuan untuk mengecek posisi titik pt pada kurva yang sudah dibuat sebelumnya dengan AA dan BB.

```c
/*Mengecek apakah titik pt ada di kurva*/
int isOnCurve(Point pt) {
    if (pt.is_infinity) return 1;
    int lhs = (pt.y * pt.y) % PRIME;
    int rhs = (pt.x * pt.x * pt.x + AA * pt.x + BB) % PRIME;
    return lhs == rhs;
}
```

Gambar 3.9 Dokumentasi Code – Pengecekan titik di kurva

Pada gambar 3.10, fungsi ketiga yang dibuat adalah fungsi penambahan dua titik. Tujuannya adalah menghasilkan titik hasil penambahan titik A dan B.

```c
/*Penambahan titik A dan titik B*/
Point addPoint(Point A, Point B) {
    Point R;
    if (A.is_infinity) return B;
    if (B.is_infinity) return A;
    if (A.x == B.x && (A.y + B.y) % PRIME == 0) {
        R.is_infinity = 1;
        return R;
    }
    int m;
    if (A.x == B.x && A.y == B.y) {
        m = (3 * A.x * A.x + AA) * inverse(2 * A.y, PRIME) % PRIME;
    } else {
        m = (B.y - A.y) * inverse(B.x - A.x, PRIME) % PRIME;
    }
    if (m < 0) m += PRIME;
    R.x = (m * m - A.x - B.x) % PRIME;
    if (R.x < 0) R.x += PRIME;
    R.y = (m * (A.x - R.x) - A.y) % PRIME;
    if (R.y < 0) R.y += PRIME;
    R.is_infinity = 0;
    return R;
}
```

Gambar 3.10 Dokumentasi Code – Penambahan dua titik

Pada gambar 3.11, fungsi yang dibuat adalah fungsi perkalian skalar pada sebuah titik. Keluaran dari fungsi ini adalah sebuah titik hasil perkalian skalar dengan titik pt sebagai masukan dari fungsi tersebut.

```c
Point kaliSkalar(int k, Point pt) {
    Point R = {0, 0, 1};
    while (k > 0) {
        if (k % 2 == 1) R = addPoint(R, pt);
        pt = addPoint(pt, pt);
        k = k / 2;
    }
    return R;
}
```

Gambar 3.11 Dokumentasi Code – Perkalian skalar pada sebuah titik

Pada gambar 3.12, fungsi yang bernama encoding bertujuan untuk merubah sebuah angka, yang pada program utama berupa bentuk ASCII dari kalimat atau pesan yang dikirim. Fungsi ini merubahnya ke dalam bentuk titik pada kurva. Selain itu, ada penghitungan dan penyimpanan nilai offset yang berguna untuk proses dekripsi.

```c
Point encoding(int num, int *offset_out) {
    for (int offset = 0; offset < 50; offset++) {
        int try_x = (num + offset) % PRIME;
        for (int y = 0; y < PRIME; y++) {
            Point p = { try_x, y, 0 };
            if (isOnCurve(p)) {
                if (offset_out) *offset_out = offset;
                return p;
            }
        }
    }
    Point inf = {0, 0, 1};
    return inf;
}
```

Gambar 3.12 Dokumentasi Code – Encoding (Pengubahan M menjadi titik)

Pada gambar 3.13, terdapat program utama yang berisi algoritma ECC ElGamal secara keseluruhan dengan bantuan fungsi-fungsi sebelumnya. Program ini menggambarkan simulasi sederhana untuk aplikasi berbagi pesan, yaitu melakukan enkripsi pada pesan yang dikirim dan melakukan dekrispi pada pesan yang diterima sehingga penerima dapat membacanya dengan benar. Proses ini memperlihatkan bahwa selama proses pengiriman, pesan dikirim dalam bentuk *ciphertext* sehingga tidak bisa dibaca oleh pihak lain, sedangkan pesan baru didekripsikan pada saat ingin diterima oleh penerima.

Awalnya, diberikan variabel titik G, d sebagai kunci privat, dan titik Q sebagai kunci publik. Pada tahap pertama, dilakukan simulasi pengiriman pesan sederhana, yaitu pengguna memasukkan pesan yang ingin dikirim. Karena ini hanya simulasi, maka kalimat dibatasi pada 256 bytes atau 256 *character*. Setelah pesan dimasukkan, dilakukan sebuah proses enkripsi pada tiap *characternya* pada baris ke-18 sampai 32. C1 dan C2 menyimpan *ciphertext*, sedangkan variabel lainnya, yaitu kQ, k, dan offset berfungsi untuk mendukung proses enkripsi. Rincian variabel sebelumnya memiliki masing-masing perannya, yaitu kQ untuk mempermudah operasi pada C2, k adalah sebuah nilai acak yang diambil, dan offset untuk menyimpan nilai offset tiap *character*nya dari fungsi encoding. Setelah itu, ditampilkan tiap *ciphertext*nya secara langsung.

Tahap terakhir adalah dekripsi *ciphertext*, hasil enkripsi dari tahap sebelumnya yang disimpan pada *array* C1 dan C2. Hasil dekripsi kemudian disimpan pada *array* recovered. Dengan melakukan algoritma yang sama seperti pada landasan teori, kini *array* recovered berisi ASCII yang sesuai dengan kondisi awal. Langkah terakhir adalah menampilkan tiap nilai *array* recovered dalam bentuk *character.*

```c
int main() {
    /*Generator, d, dan Q*/
    Point G = {3, 6, 0};
    int d = 5;
    Point Q = kaliSkalar(d, G);

    /*Input string*/
    char input[256];
    int ascii[256];
    printf("Enter the word : ");
    fgets(input, sizeof(input), stdin);
    int len = strlen(input);
    for (int i=0; i < len; i++) {
        ascii[i] = (int)input[i];
    }

    // Encryption
    Point C1[256],C2[256],kQ[256];
    int k = 7;
  /*Kasus nyata : K nya akan digenerate secara acak.*/
    int offset[256];
    for (int i=0;i<len;i++) {
        offset[i] = 0;
        Point M = encoding(input[i], &offset[i]);
        if (M.is_infinity) {
            printf("Failed to encode number to curve.\n
");
            return 1;
        }
        C1[i] = kaliSkalar(k, G);
        kQ[i] = kaliSkalar(k, Q);
        C2[i] = addPoint(M, kQ[i]);
        printf("Encrypted C(%d): (%d, %d), C2: (%d, %d)
\n", i, C1[i].x, C1[i].y, C2[i].x, C2[i].y);
    }

    // Decryption
    int recovered[256];
    for (int i=0;i<len;i++) {
        Point dkC1 = kaliSkalar(d, C1[i]);
        dkC1.y = PRIME - dkC1.y;
        Point decrypted = addPoint(C2[i], dkC1);
        recovered[i] = (decrypted.x - offset[i] + PRIME
) % PRIME;

    }

    for (int i=0;i<len;i++) {
        printf("%c", (char)recovered[i]);
    }
    return 0;
}
```

Gambar 3.13 Dokumentasi Code – Program Utama ECC ElGamal

Keseluruhan program ini tidak menggambarkan keamanan yang sebenarnya, tetapi mewakilkan simulasi sederhana mengenai cara kerja enkripsi dan dekripsi pesan pada aplikasi berbasis pesan. Pada gambar 3.14, ditampilkan hasil eksekusi dari program di atas.

```
Enter the word : amin
Encrypted C(0): (32, 210), C2: (552, 0)
Encrypted C(1): (32, 210), C2: (53, 605)
Encrypted C(2): (32, 210), C2: (53, 605)
Encrypted C(3): (32, 210), C2: (53, 605)
Encrypted C(4): (32, 210), C2: (368, 662)
amin
```

Gambar 3.14 Dokumentasi Code – Hasil Simulasi Sederhana ECC ElGamal

IV. ANALISIS ALGORITMA KRIPTOGRAFI ASIMETRIS

Setelah membuat simulasi sederhana mengenai RSA dan ECC ElGamal, analisis terhadap kedua algoritma pun akhirnya bisa dilakukan. Dengan mengamati cara kerja dan kode keduanya, analisis keamanan dan kompleksitas waktu pun bisa dilakukan. Hal yang akan dibahas meliputi cara kerja, keamanan, kecepatan, dan kompleksitas waktunya.

A. Analisis RSA

RSA memanfaatkan prinsip faktorisasi prima yang umumnya berupa bilangan besar sebagai matematikanya. Pada implementasi algoritma RSA, dapat dilihat bahwa ada tiga fungsi tambahan berupa algoritma euclidean untuk mencari PBB, *extended euclidean algorithm* untuk mencari modulo invers, dan perhitungan yang ada pada enkripsi dan dekripsi termasuk perpangkatan. Terakhir, pada program utama, hanya dilakukan tiga tahap RSA dan satu tahap menerima masukan serta *encoding* masukan ke dalam bentuk angka supaya bisa dienkripsi dan didekripsi RSA. Tiga tahap RSA meliputi pembuatan kunci publik dan privat, proses enkripsi, dan proses dekripsi.

RSA dikenal dengan keamanannya karena sulitnya untuk melakukan faktorisasi bilangan yang sangat besar. Peretas atau pihak yang ingin mencoba memecahkan RSA membutuhkan waktu yang sangat panjang. Jika ingin memecahkan RSA, maka hal yang bisa dilakukan adalah mencari p dan q dari n, serta menghitung d (kunci privat). Pada skala kecil, tindakan ini bisa dilakukan dengan mudah dan cepat, tetapi pada bilangan yang lebih besar, tindakan ini lebih sulit dan hampir tidak bisa diretas dengan komputer sekarang.

Untuk memecahkan RSA, diperlukan faktorisasi n menjadi p dan q yang mempunyai kompleksitas waktu O(log (n)). Jika algoritma RSA tersebut memakai 2048-bit dan algoritmya yang dipakai untuk mencari faktornya adalah GNFS yang merupakan algoritma faktorisasi tercepat, maka waktu yang dibutuhkan adalah sebagai berikut.

$$T(n) = exp\,(1.923\,.\,11.2\,.\,4.2) = \exp(90.5) = 1{,}4\,.\,10^{39}$$

Jika peretas menggunakan komputer tercepat, yaitu El Capitan yang bisa mengerjakan operasi sebanyak 10^{21}operasi per detiknya, maka waktu yang dibutuhkan sekitar 10^{18}detik atau setara dengan milyaran tahun dan hal ini sangat tidak mungkin. Oleh karena itu, RSA memiliki keamanan yang cukup kuat untuk masa sekarang.

Selanjutnya, untuk melihat kompleksitas waktu dari RSA, implementasi program RSA sebelumnya harus diperhatikan terlebih dahulu. Pada gambar 3.5, dengan melihat pada baris ke-25 sampai 28 dan baris ke-34, dapat diketahui bahwa proses enkripsi dan dekripsi hanya berada pada baris tersebut. Artinya jika kita hanya menilai enkripsi dan dekripsi pada RSA, maka hasil yang didapatkan adalah sebagai berikut.

$$Fungsi\ Calculation = O(\log n),$$

Dengan n adalah masukan yang dapat berupa dua pilihan, yaitu nilai e dan d, Sedangkan proses enkripsi dan dekripsi pada program utama mempunyai O(n). Oleh karena itu, total kompleksitas waktu yang dipunyai algoritma enkripsi dan dekripsi adalah berturut-turut O(n log e) dan O(n log d).

Pada urutan kecepatan kompleksitas waktu, kecepatan ini dapat dikatakan tidak terlalu cepat dan tidak lambat pula. Tetapi dapat kita pastikan bahwa kompleksitas waktu adalah fleksibel, artinya bisa berubah bergantung pada algoritma yang dibuat pada program. Contohnya adalah perbedaan penggunaan algoritma untuk mencari PBB dapat mempengaruhi hasil dan seterusnya. Oleh karena itu, keamanan berlaku sama pada tiap versi penulisan kodenya sedangkan kecepatan bergantung kepada penulisan kode.

B. Analisis ECC ElGamal

ElGamal memanfaatkan prinsip logaritma diskrit dan ECC memanfaatkan prinsip kurva eklipsik. Dengan mengubah ElGamal menjadi ECC yang menggunakan prinsip kurva eklipsik, dapat dihasilkan sebuah algoritma enkripsi dan dekripsi yang lebih aman dibandingkan dengan RSA.

Pada implementasi program yang dibuat, dapat diketahui bahwa program tersebut mempunyai lima fungsi tambahan, yaitu *extended euclidean algorithm* untuk mencari modulo invers, pengecekan titik di kurva, penambahan titik, perkalian skalar sebuah titik, dan *offset encoding*, yaitu proses mengubah sebuah angka menjadi titik pada kurva. Lalu, pada gambar 3.13, dapat dilihat sebuah program utuh mengenai cara kerja ECC ElGamal secara keseluruhan.

Pada tahap awal, dibuat beberapa variabel, seperti titik dasar (G), d (kunci privat), dan Q (kunci publik). Tahapan selanjutnya adalah memasukkan pesan atau informasi. Kemudian, pesan tersebut diubah dalam bentuk titik menggunakan fungsi *encoding*. Setelah proses *encoding*, proses enkripsi dan dekripsi berjalan menggunakan titik-titk yang ada. Berbeda dengan RSA, *ciphertext* dari ECC terdiri dari dua, yaitu C1 dan C2. C1 dan C2 inilah yang nantinya akan dikirim ke pengguna atau tempat lain. *Ciphertext* itu kemudian akan melalui proses dekripsi sebelum sampai pada perangkat penerima. Pada program, proses dekripsi melibatkan C1 dan C2 yang melalui persamaan berikut ini.

$$M = C2 - X.C1$$

Koordinat X dari M merupakan pesan asli dalam bentuk ASCII. Kemudian, ASCII tersebut akan dikonversi menjadi *character* dan ditampilkan ke penerima.

Sebagai algoritma yang lebih baru, ECC ElGamal menawarkan keamanan yang kuat. Sampai masa kini, ECC masih secara masif digunakan sebagai algoritma-algoritma kriptografi di dunia digital. Untuk memecahkan ECC, hal yang bisa dilakukan adalah menyelesaikan ECDLP (*Eliptic Curve Discrete Logarithm Problem)*, yaitu mencari k sehingga Q = kP. Langkah terbaik untuk menyelesaikan permasalahan ini adalah dengan Pollard's Rho for Elliptic Curves dengan waktu selama $O\sqrt{n}$. Jika algoritma menggunakan ECC 256-bit dan menggunakan Pollard's Rho for Elliptic Curves sebagai algoritma penyelesaiannya, maka waktu yang dibutuhkan adalah sebagai berikut.

$$T(n) = \sqrt{n} = \sqrt{2^{256}} = 2^{128}$$

Jika menggunakan komputer El Capitan yang bisa mengerjakan operasi sebanyak 10^{21} operasi per detik, maka waktu yang dibutuhkan adalah sebesar 10^{17} detik atau setara dengan milyar tahun. Dengan komputer sekarang, operasi semacam ini mustahil untuk dilakukan. Oleh karena itu, ECC mempunyai keamanan yang kuat untuk masa sekarang.

Terakhir, untuk menilai kompleksitas waktu dari ECC, diperlukan pengamatan terhadap implementasi ECC dalam algoritma ElGamal yang sudah dibuat sebelumnya. Pada gambar 3.13, diperlihatkan program utama algoritma ECC yang memuat proses memasukkan masukan, *encoding*, dekripsi, dan enkripsi. Untuk menilai ECC, diperlukan pengamatan terhadap hanya proses enkripsi dan dekripsi. Proses enkripsi tanpa encoding menghasilkan waktu sebesar O(n . log k) dan dekripsi membutuhkan waktu sebesar O(n . log d).

Pada tahapan ini, proses yang paling memberatkan adalah encoding karena melakukan *brute force* untuk y pada kurva. Lalu, kecepatan algoritma ini cenderung tidak terlalu cepat dan tidak terlalu lambat. Sama seperti yang lain, kompleksitas waktu pada algoritma adalah fleksibel, artinya dapat berubah-ubah bergantung pada penulisan kode. Oleh karena itu, kecepatan algoritma bergantung pada penulisan kode, tetapi keamanan algoritma berlaku sama pada tiap kodenya.

C. Perbandingan Kedua Algoritma

Setelah menganalisis RSA dan ECC, dapat diketahui bahwa masing-masing memang mempunyai prinsip enkripsi dan dekripsi yang berbeda. Perbedaan antara keduanya dimuat pada tabel berikut ini.

Aspek	**Perbedaan Antara**	
	RSA	**ECC ElGamal**
Keamanan	RSA cukup aman dengan setidaknya 2048 bit.	ECC menawarkan keamanan yang lebih tinggi pada bit yang sama dengan RSA. Sehingga ECC 256 bit bisa menyamai keamanan RSA 2048 bit
Ukuran kunci	Kunci RSA cenderung lebih panjang dengan keamanan yang sama.	Kunci ECC lebih pendek dengan keamanan yang sama.
Algoritma / Prinsip	Berdasarkan faktorisasi prima bilangan bulat besar	Berdasarkan logaritma diskrit pada kurva eliptik

Tabel 4.1 Perbedaan antara RSA dan ECC ElGamal

V. KESIMPULAN

Berdasarkan implementasi dan analisis program RSA dan ECC, dapat disimpulkan bahwa ECC merupakan algoritma yang lebih aman dengan kunci yang lebih pendek dibandingkan dengan RSA. Hal ini membuat ECC merupakan algoritma yang lebih efisien.

Selain itu, kompleksitas waktu antara ECC dan RSA tidak jauh berbeda dengan satu catatan penting, yaitu kompleksitas waktu bergantung kepada implementasi algoritma pada sebuah program.

VI. LAMPIRAN

Source code dapat diakses pada tautan berikut ini.
https://github.com/leains/RSA-and-ECC-ElGamal.git

VII. UCAPAN TERIMA KASIH

Atas terselesainya makalah ini, penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas rahmatNya untuk menyelesaikan makalah tanpa adanya hambatan. Terima kasih pula kepada Bapak dosen pengampu mata kuliah Matematika Diskrit yang telah memberikan pengetahuan dan wawasan sehingga penulis bisa menyelesaikan makalah dengan baik. Terakhir, terima kasih terhadap pihak-pihak yang tidak bisa disebutkan penulis atas bantuannya untuk menyelesaikan makalah.

TAUTAN VIDEO DI YOUTUBE

https://youtu.be/Esym8Dp5ktc

REFERENSI

[1] William Stallings , *CRYPTOGRAPHY AND NETWORK SECURITY PRINCIPLES AND PRACTICE Fifth Edition*, Hal. 269-320. 2011.

[2] R. Munir, "Elliptic Curve Cryptography (ECC)". 2023. Diakses pada tanggal 19 Juni 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2022-2023/12-ECC-2023.pdf

[3] R. Munir, "Algoritma ElGamal". 2023. Diakses pada tanggal 19 Juni 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2022-2023/20-Algoritma-Elgamal-2023.pdf.

[4] R. Munir, "Kompleksitas Algoritma". 2023. Diakses pada tanggal 19 Juni 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf.

[5] Prof.Bill Buchanan, "ElGamal Encryption with Elliptic Curves". 2023. Diakses pada tanggal 20 Juni 2025 dari https://billatnapier.medium.com/elgamal-encryption-with-elliptic-curves-2147f990fbd.

[6] Dhimaz Purnama Adjhi, Mohamad Rizal Hanafi, Rastra Wardana Nanditama. "RSA Cryptography Algorithm". 2023. Diakses pada tanggal 19 Juni 2025 dari https://tekkom.upi.edu/2023/02/rsa-cryptography-algorithm/.

[7] Aryasaktiwirasena, "Fungsi Totient Euler (Euler's Totient Function)". 2021. Diakses pada tanggal 20 Juni 2025 dari https://aryasaktiwirasena.web.ugm.ac.id/2021/01/10/fungsi-totient-euler-eulers-totient-function/.

[8] "Apa itu Kriptografi?". Diakses pada tanggal 18 Juni 2025 dari https://aws.amazon.com/id/what-is/cryptography/.

[9] "Euclidean algorithms (Basic and Extended)". 2025. Diakses pada tanggal 20 Juni 2025 dari https://www.geeksforgeeks.org/dsa/euclidean-algorithms-basic-and-extended/.

[10] "RSA Algorithm in Cryptography". 2025. Diakses pada tanggal 20 Juni 2025 dari https://www.geeksforgeeks.org/computer-networks/rsa-algorithm-cryptography/.

[11] "Euler's Totient Function". 2025. Diakses pada tanggal 20 Juni 2025 dari http://geeksforgeeks.org/dsa/eulers-totient-function/.

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, 20 Juni 2025

Leonardus Brain Fatolosja, 13524146