BAB III
METODOLOGI PENELITIAN

3.1 Objek dan Subjek Penelitian

Objek penelitian yang digunakan adalah waktu pembangkitan kunci, waktu enkripsi, waktu dekripsi, waktu pembangitan kunci untuk tanda tangan digital (*digital signature key generation)*, waktu pembuatan tanda tangan digital, serta nilai kompleksitas waktu dari setiap algoritma.

Subjek penelitian yang digunakan adalah aplikasi enkripsi dan dekripsi yang ditulis dengan Bahasa pemrograman C++ menggunakan library NTL dalam perangkat lunak Code::Blocks versi 20.03. Kebutuhan minimum untuk menjalankan software Code::Blocks akan dijelaskan seperti tabel dibawah ini.

Tabel 3. 1 Spesifikasi minimum Code::Blocks

Komponen	Spesifikasi Minimum
OS	Windows 98
Prosesor (CPU)	Intel Dual Core
Memori (RAM)	256 MB
Harddisk	100 MB

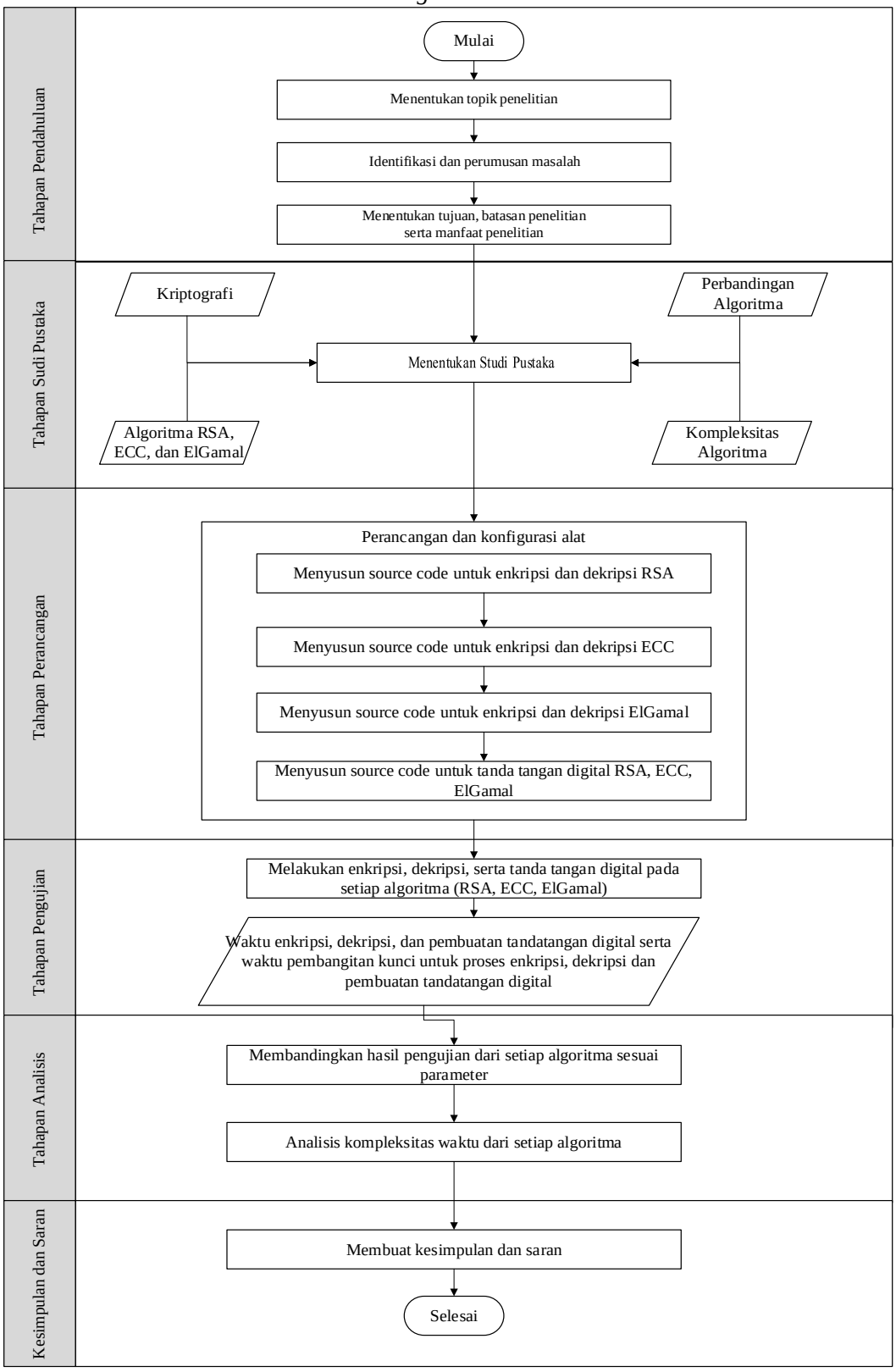
Berdasarkan spesifikasi minimum yang dibutuhkan, maka penulis menggunakan perangkat keras dan perangkat lunak dengan spesifikasi yang digunakan dapat dilihat pada tabel 3.1.

Tabel 3. 2 spesifikasi komponen penelitian

Komponen	Spesifikasi	
Komputer/Laptop	Seri	: Lenovo Ideapad 330
	Processor	: Intel® Quad Core i3-8250U
	RAM	: 8 GB
	Harddisk	: 1 TB
	Sistem operasi	: Windows 10

3.2 Diagram Alir Penelitian

Tabel 3. 3 Diagram Alir Penelitian



3.2.1 Tahapan Pendahuluan

Tahapan pendahuluan dimulai dengan menentukan topik penelitian yang akan diteliti. Hal ini bertujuan untuk menentukan pedoman pembahsan yang akan dijabarkan selanjutnya. Setelah menentukan topik penelitian selanjutnya adalah melakukan identifikasi dan perumusan masalah. Perumusan masalah berisi kalimat tanya yang bertujuan untuk membantu peneliti agar pembahasan penelitian dilakukan sesuai dengan alur penelitian. Menentukan tujuan, batasan masalah serta manfaat dari penelitian, hal ini bertujuan untuk memberikan batasan serta memfokuskan pembahasan sesuai dengan topik penelitian dan perumusan masalah yang diangkat.

3.2.2 Tahapan Studi Pustaka

Tahapan studi pustaka dilakukan dengan mencari literatur berupa jurnal penelitian sebelumnya yang berkaitan dengan topik penelitian yang diangkat. Hal ini bertujuan untuk membantu peneliti dalam memperluas wawasan serta memberikan gambaran tentang penelitian yang akan dilakukan. Selain jurnal penelitian sebelumnya, peneliti menggunakan buku yang berkaitan dengan topik penelitian yang selanjutnya digunakan sebagai dasar teori dalam mengembangkan penelitian. Secara garis besar proses pencarian literatur dalam penelitiann ini dibagi menjadi empat bagian yaitu kriptografi, algoritma kriptografi asimetris (RSA, ECC, dan ElGamal), perbandingan algoritma, serta kompleksitas waktu algoritma.

3.2.3 Tahapan Perancangan

Tahapan perancangan dilakukan dengan membuat rancangan sistem yang akan digunakan dalam penelitian. Langkah awal yang dilakukan adalah pembuatan *flowchart* atau bagan alir dari sistem yang akan dibangun. Setelah pembuatan *flowchart* selanjutnya peneliti membuat *pseoducode* yang dikembangkan dari *flowchart*. *Pseudocode*

berisi deskripsi atau rancangan sederhana dari algoritma yang selanjutnya akan digunakan untuk membuat sistem yang akan digunakan dalam penelitian.

Sistem enkripsi dan dekripsi yang digunakan dalam penelitian dibangun berdasarkan rancangan *pseudocode* yang telah dibuat. Selanjutnya peneliti menyusun *source code* sistem enkripsi dan dekripsi dari masing-masing algoritma serta *soure code* untuk melakukan tanda tangan digital dari setiap algoritma.

3.2.4 Tahapan Analisis

Tahapan analisis dilakukan dengan melakukan pengambilan data dari setiap parameter yang telah ditentukan diawal penelitian. Data yang diperoleh berupa waktu pembangkitan kunci public dan kunci privat, waktu enkripsi dan dekripsi, waktu pembangkitan kunci tanda tangan digital serta waktu pembuatan tandatangan digital. Data yang peroleh selanjutnya akan dibandingkan dari masing-masing algoritma. Data waktu yang diperoleh selanjutnya digunakan untuk melakukan perhitungan kompleksitas waktu dari masing-masing algoritma.

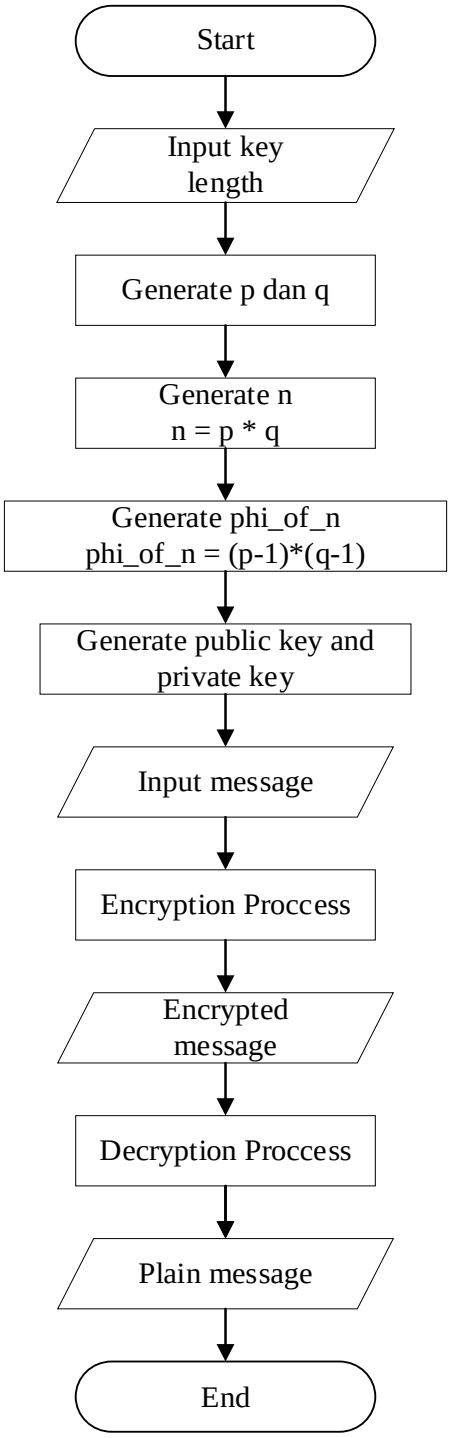
3.2.5 Tahapan Kesimpulan dan Saran

Kesimpulan diambil dari hasil perbandingan data yang diperoleh dari parameter setiap algoritma serta perbandingan hasil perhitungan kompleksitas waktu.

3.2.6 Teknik Pengumpulan Data

Data dalam penelitian diambil dari sistem enkripsi dan dekripsi yang dibangun dengan menggunakan Bahasa pemrograman C++. Proses enkripsi dan dekripsi dari masing-masing algoritma akan dijelaskan menggunakan *flowchart* serta *pseudocode* dari masing-masing algoritma agar lebih mudah dipahami. Proses enkripsi dan dekripsi akan dijelaskan sebagai berikut.

1. Algoritma RSA
   a. Enkripsi dan Dekripsi



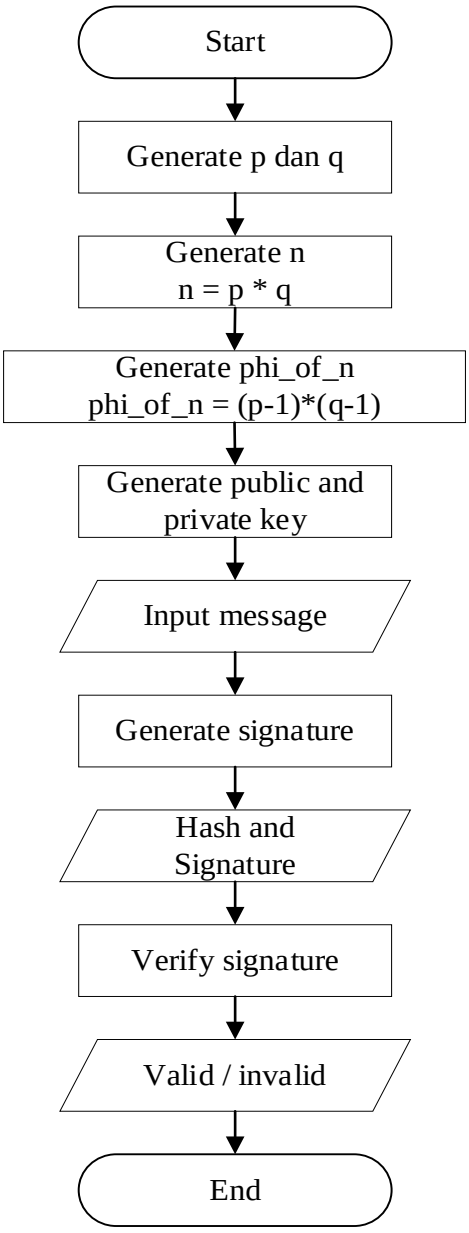
Gambar 3. 1 Flowchart Enkripsi dan Dekripsi Algoritma RSA

Tabel 3. 3 Pseudocode enkripsi dan Dekripsi Algoritma RSA

Pseudocode

```
program RSA_Encryption and Decryption

deklarasi:
var  p,  q,  n,  phi_of_n,  public_key,  private_key,
    message, encrypted_message,decrypted_message : ZZ
    panjang_kunci: integer

Algoritma:
INPUT panjang_kunci
p ← GenPrime_ZZ(panjang_kunci, 80)
q ← GenPrime_ZZ(panjang_kunci, 80)
n ← p * q
phi_of_n ← (p-1)*(q-1)

i: ZZ
bits_of_n: long
begin
FOR ( ; i<=n; i*=10)
   bits_of_n++
END FOR

public_key ← GenPrime_ZZ(bits_of_n/2, 80)
private_key ← InvMod(public_key, phi_of_n)

INPUT message
encrypted_message = PowerMod(message, public_key, n)
decrypted_message = PowerMod(encrypted_message,
   private_key, n);

OUTPUT (p, q, n, phi_of_n, public_key, private_key,
   encrypted_message)
```

b. Tanda tanga digital



Gambar 3. 2 Flowchart Tanda Tangan Digital Algoritma RSA

Tabel 3. 4 Pseudocode Tanda Tangan Digital Algoritma RSA

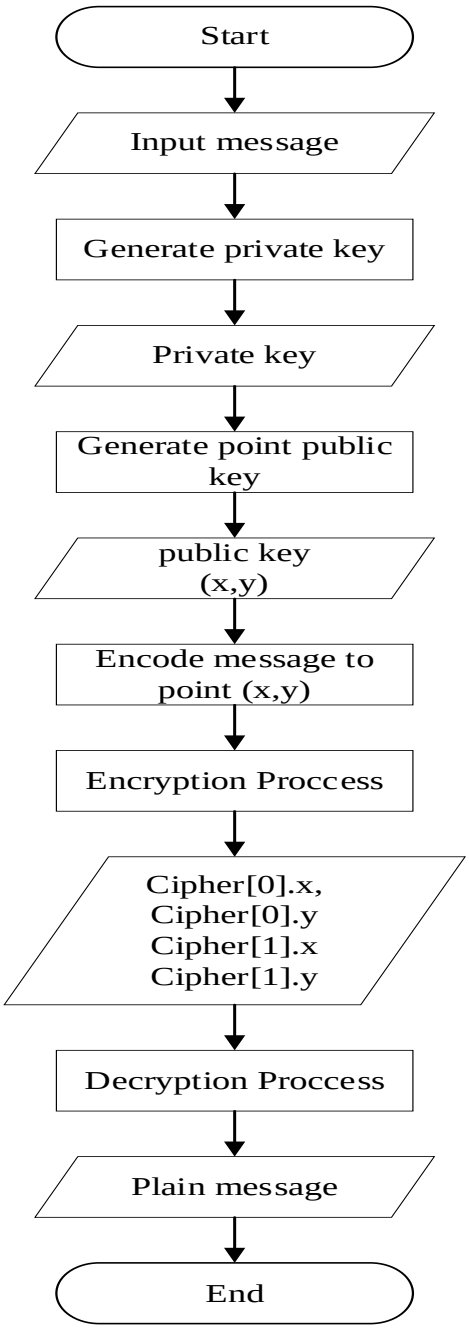
Pseudocode
Program RSA_Digital_signature **Deklarasi** Var p, q, n, phi_of_n, public_key, private_key: ZZ message: std::string panjang_kunci: integer SHA1: include file **Algoritma**

```
convert_hexa_decimal_string_to_ZZ (num:string)
   ZZ result ← ZZ(0);
   FOR (int i = 0; i < num.length(); i++)
      result ← result*16 + decimal_value(num[i])
   END FOR
   return result

convert_decimal_to_ZZ(num:string)
   ZZ result ← ZZ(0)
   FOR (int i = 0; i < num.length(); i++)
      result ← result*10 + num[i] - '0'
   END FOR
   return result

generate_hash(input:string)
   SHA1 checksum
   checksum.update(input)
   const string hash ← checksum.final()
   ZZ hash_val ← convert_hexa_decimal_string_to_ZZ(hash)
   return hash_val

sign(message, private_key, n)
   ZZ hash ← generate_hash(message)
   OUTPUT (hash)
   ZZ signature ← PowerMod(hash, private_key, n)
   OUTPUT (signature)
   return signature

verify_signature(message, public_key, signature, n)
   ZZ hash ← generate_hash(message)
   OUTPUT (hash)
   ZZ hash_dash ← PowerMod(signature, public_key, n)
   OUTPUT (hash_dash)
   IF (hash == hash_dash)
       OUTPUT "Signature is valid"
   ELSE
       OUTPUT "signature is not valid"
    ENDIF

main()
    INPUT panjang_kunci
    p ← GenPrime_ZZ(panjang_kunci, 80)
    q ← GenPrime_ZZ(panjang_kunci, 80)
    n ← p * q
    phi_of_n ← (p-1)*(q-1)

    i : ZZ
    bits_of_n : long
    begin
    FOR ( ; i<=n; i*=10)
       bits_of_n++
    END FOR

    public_key ← GenPrime_ZZ(bits_of_n/2, 80)
    private_key ← InvMod(public_key, phi_of_n)
```

```
INPUT message
ZZ signature ← sign(message, private_key, n)
verify_signature(message, public_key, signature, n)
```

2. Algoritma ECC

   a. Enkripsi dan Dekripsi



Gambar 3. 3 Flowchart Enkripsi dan Dekripsi algoritma ECC

Tabel 3. 5 pseudocode Enkripsi dan Dekripsi algoritma ECC

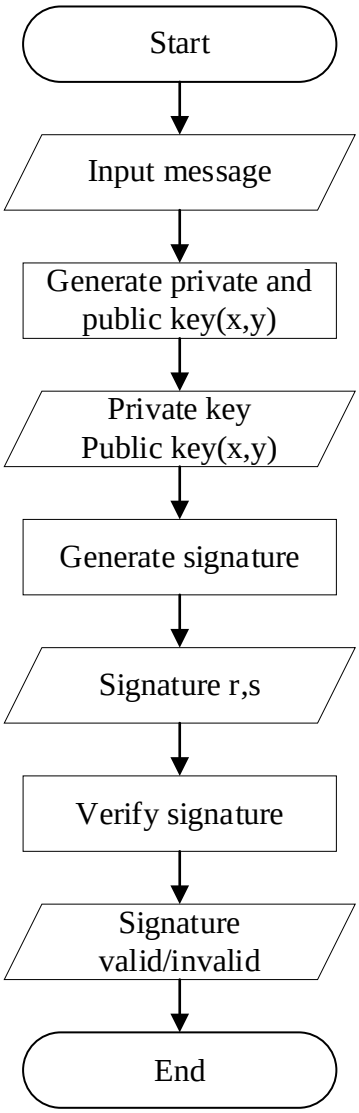
Pseudocode

```
Program ECC_Encryption_and_Decryption

Deskripsi:
typedef struct point{
    ZZ x
    ZZ y
}Point

// NIST parameters for 192bit
ZZ  p  ← power(ZZ(2), long(192)) - power(ZZ(2),
  long(64)) - 1   //feild parameter
ZZ a ← ZZ(-3)  // elliptic cuve parameter
ZZ                          b                    ←
  conv<ZZ>("2455155546008943817740293915197451784769
  1080581611912380 65") // elliptic curve parameter
ZZ                          n                    ←
  conv<ZZ>("6277101735386680763835789423176059013767
  19477318284228 4081") // order of elliptic curve
ZZ                          Px                   ←
  conv<ZZ>("6020462823756886567582134805875261119166
  6989766368846848 18") // x cordinate of base point
ZZ                          Py                   ←
  conv<ZZ>("1740503322936220314048575522802194103640
  234889273866506 41") // y cordinate of base point

Algoritma:
point_doubling(Point P){
    ZZ x1 ← P.x, y1 ← P.y;
    IF(y1 == 0) THEN
        OUTPUT "Point doubling error"
    ELSE
   OUTPUT {ZZ(0), ZZ(0)}

    ZZ m ← ((3*x1*x1 + a) * InvMod((2*y1) % p, p)) %
  p
    ZZ x3 ← (m*m - 2*x1) % p
    ZZ y3 ← (m*(x1 - x3) - y1) % p
    return {x3, y3};
}

point_addition(Point P, Point Q){
    ZZ x1 ← P.x, y1 ← P.y, x2 ← Q.x, y2 ← Q.y;

    IF (y1 == y2 or y1 == -y2) THEN
        OUTPUT "Point Addition invalid operation"
    ELSE
   OUTPUT {ZZ(0), ZZ(0)}

    ZZ m ← (((y2 - y1) % p) * InvMod((x2 - x1) % p,
  p)) % p
    ZZ x3 ← ((m*m) % p - (x1 + x2) % p) % p
    ZZ y3 ← ((m*(x1 - x3)) % p - y1) % p
```

```
    return {x3, y3}

scalar_multiply(ZZ k, Point P)
    Point P1 ← P, P2
    bool p2_initialized ← false

    WHILE(k != ZZ(0))
        IF(operator&(k, ZZ(1)) > ZZ(0)) THEN
            IF(!p2_initialized) THEN
                p2_initialized ← true
                P2 ← P1
            ELSE
                P2 ← point_addition(P1, P2)

        P1 ← point_doubling(P1)
        k ← RightShift(k, long(1))
    ENDWHILE
    return P2


ZZ ecc_y_val(ZZ x)
    return (power(x, long(3)) + a*x + b) % p

encode_message_to_point(ZZ message)
    ZZ xj ← 100*message
    FOR(long j ← 0; j<100; j++){
        ZZ sj ← ecc_y_val(xj + j);
        IF(PowerMod(sj, (p-1)/2, p) == ZZ(1)) THEN
            ZZ yj ← SqrRootMod(sj, p)
            return {xj + j, yj}
      ENDIF
    ENDFOR
    OUTPUT"Message encoding failed"
    return {ZZ(0), ZZ(0)}

choose_random_integer(ZZ num)
    ZZ private_key ← RandomBnd(num)
    WHILE(private_key == ZZ(0))
        private_key ← RandomBnd(num)
    return private_key
    ENDWHILE

generate_public_key(ZZ private_key)
    return scalar_multiply(private_key, {Px, Py})

encrypt_message(Point message, Point public_key)
    ZZ k ← choose_random_integer(n)
    Point C1 ← scalar_multiply(k, {Px, Py})
    Point   k_mul_public_key   ←   scalar_multiply(k,
  public_key)
    Point      C2      ←      point_addition(message,
  k_mul_public_key)
    Vec<Point>cipher
    cipher.append(C1)
    cipher.append(C2)
```

```
    return cipher

decrypt_message(Vec<Point> cipher, ZZ private_key)
    Point c1 ← cipher[0]
    Point c2 ← cipher[1]
    Point            private_key_mul_c1            ←
   scalar_multiply(private_key, c1)
    Point       M       ←       point_addition(c2,
   {private_key_mul_c1.x, -private_key_mul_c1.y})
    return M.x/100
}

main()
    ZZ message
    INPUT message

    private_key ← choose_random_integer(n)
    OUTPUT (private_key)

    public_key = generate_public_key(private_key)
    OUTPUT (public_key)

    M ← encode_message_to_point(message)
    OUTPUT (M)

    cipher ← encrypt_message(M, public_key)
    OUTPUT (cipher[0].x, cipher[0].y)
    OUTPUT (cipher[1].x, cipher[1].y)

    decrypted_message ← decrypt_message(cipher,
    private_key)
    OUTPUT(decrypted_message)
```

b. Tanda tanga digital



Gambar 3. 4 flowchart tanda tangan digital algoritma ECC

Tabel 3. 6 tanda tangan digital algoritma ECC

Pseudocode

```
Program ECC_Digital_Signature

Deskripsi:
typedef struct point{
    ZZ x
    ZZ y
}Point

// NIST parameters for 192bit
```

```
ZZ  p  ← power(ZZ(2), long(192)) - power(ZZ(2),
  long(64)) - 1   //feild parameter
ZZ a ← ZZ(-3)  // elliptic cuve parameter
ZZ                          b                          ←
  conv<ZZ>("24551555460089438177402939151974517847 6
  9108058161191238065") // elliptic curve parameter
ZZ                          n                          ←
  conv<ZZ>("62771017353866807638357894231760590137 6
  7194773182842284081") // order of elliptic curve
ZZ                          Px                         ←
  conv<ZZ>("60204628237568865675821348058752611191 6
  6989766368846848 18") // x cordinate of base point
ZZ                          Py                         ←
  conv<ZZ>("17405033229362203140485755228021941036 4
  023488927386650641") // y cordinate of base point
```

Algoritma:

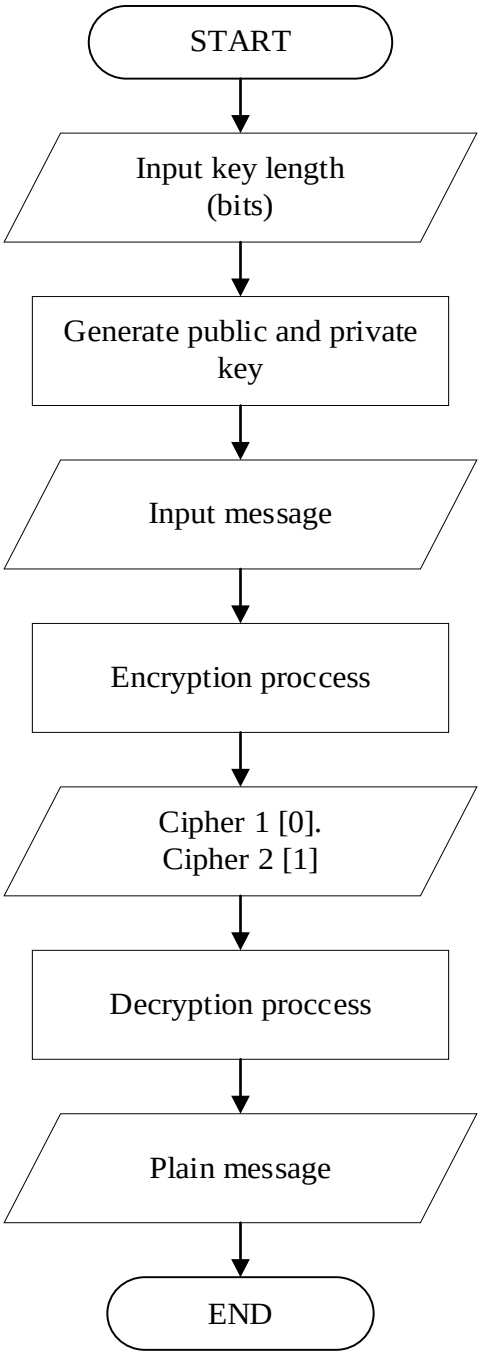
```
point_doubling(P){
    x1 ← P.x, y1 = P.y;
    IF(y1 == 0) THEN
        OUTPUT ("Point doubling error")
    ELSE
       OUTPUT( {ZZ(0), ZZ(0)} )

    m ← ((3*x1*x1 + a) * InvMod((2*y1) % p, p))%p
    x3 ← (m*m - 2*x1) % p
    y3 ← (m*(x1 - x3) - y1) % p
    return {x3, y3}

point_addition(P, Q)
    x1 ← P.x, y1 ← P.y, x2 ← Q.x, y2 ← Q.y;
    IF (y1 == y2 or y1 == -y2) THEN
       OUTPUT ("Point Addition invalid operation")
    ELSE
       OUTPUT( {ZZ(0), ZZ(0)} )
    ENDIF
    m ←(((y2 - y1)%p)*InvMod((x2 - x1) %p, p)) %p
    x3 ← ((m*m) % p - (x1 + x2) % p) % p
    y3 ← ((m*(x1 - x3)) % p - y1) % p
    return {x3, y3}

scalar_multiply(ZZ k, Point P)
    P1 ← P, P2
    bool p2_initialized ← false
       WHILE(k != ZZ(0))
        IF(operator&(k, ZZ(1)) > ZZ(0)) THEN
            IF(!p2_initialized) THEN
                p2_initialized ← true
                P2 ← P1
            ELSE
                P2 ← point_addition(P1, P2)
            ENDIF
        P1 ← point_doubling(P1)
         k ← RightShift(k, long(1))
         ENDIF
```

```
    ENDWHILE
    return P2

ecc_y_val(x)
    return (power(x, long(3)) + a*x + b) % p

choose_random_integer(ZZ num)
    ZZ private_key ← RandomBnd(num)
    WHILE(private_key == ZZ(0))
        private_key ← RandomBnd(num)
    return private_key
    ENDWHILE
generate_public_key(ZZ private_key)
    return scalar_multiply(private_key, {Px, Py})
decimal_value(ch:char){
    SWITCH(ch){
        case '0' : return 0;
        case '1' : return 1;
        case '2' : return 2;
        case '3' : return 3;
        case '4' : return 4;
        case '5' : return 5;
        case '6' : return 6;
        case '7' : return 7;
        case '8' : return 8;
        case '9' : return 9;
        case 'a' : return 10;
        case 'b' : return 11;
        case 'c' : return 12;
        case 'd' : return 13;
        case 'e' : return 14;
        case 'f' : return 15;
    ELSE: return -1;
convert_hexa_decimal_string_to_ZZ(num:string)
    ZZ result ← ZZ(0)
    FOR(int i = 0; i < num.length(); i++)
        result ← result*16 + decimal_value(num[i])
    ENDFOR
    return result

generate_hash(input:string)
  SHA1 checksum
  checksum.update(input)
  const string hash ← checksum.final()
  hash_val ← convert_hexa_decimal_string_to_ZZ(hash)
  return hash_val

sign(string message, ZZ private_key){
    r, s, k;
    hash ← generate_hash(message)
    DO
        r ← ZZ(0)
        DO
            k ← choose_random_integer(n)
            Point kP <-- scalar_multiply(k, {Px, Py})
            r ← kP.x % n
```

```
        WHILE(r == ZZ(0))

        s <-- (InvMod(k, n)*((hash + (private_key*r)
  % n) % n)) % n
    WHILE (s == ZZ(0))
    OUTPUT (signature "r")
    OUTPUT (signature "s")

    Vec<ZZ> signature
    signature.append(r)
    signature.append(s)
    return signature

verify_signature(signature, message, public_key)
    r ← signature[0], s ← signature[1]
    IF(r < 1 or r > n-1 or s < 1 or s > n-1)THEN
        OUTPUT("Signature is invalid"
        return
    ENDIF
    hash ← generate_hash(message)
    w ← InvMod(s, n)
    u1 ← (hash*w) % n
    u2 ← (r*w) % n
    u1_mul_P = scalar_multiply(u1, {Px, Py})
    u2_mul_Public_key     ←     scalar_multiply(u2,
  public_key)
    X ← point_addition(u1_mul_P, u2_mul_Public_key)
    v ← X.x % n
    IF(v == r)
        OUTPUT("Signature is valid")
    ELSE
        OUTPUT("Signature is invalid")

main()
    INPUT message

    private_key ← choose_random_integer(n)
    OUTPUT (private_key)

    public_key = generate_public_key(private_key)
    OUTPUT (public_key)

    M ← encode_message_to_point(message)
    OUTPUT (M)

    cipher ← encrypt_message(M, public_key)
    OUTPUT (cipher[0].x, cipher[0].y)
    OUTPUT (cipher[1].x, cipher[1].y)

    decrypted_message   ←   decrypt_message(cipher,
  private_key)
    OUTPUT(decrypted_message)
```

3. Algoritma ElGamal

   a. Enkripsi dan dekripsi



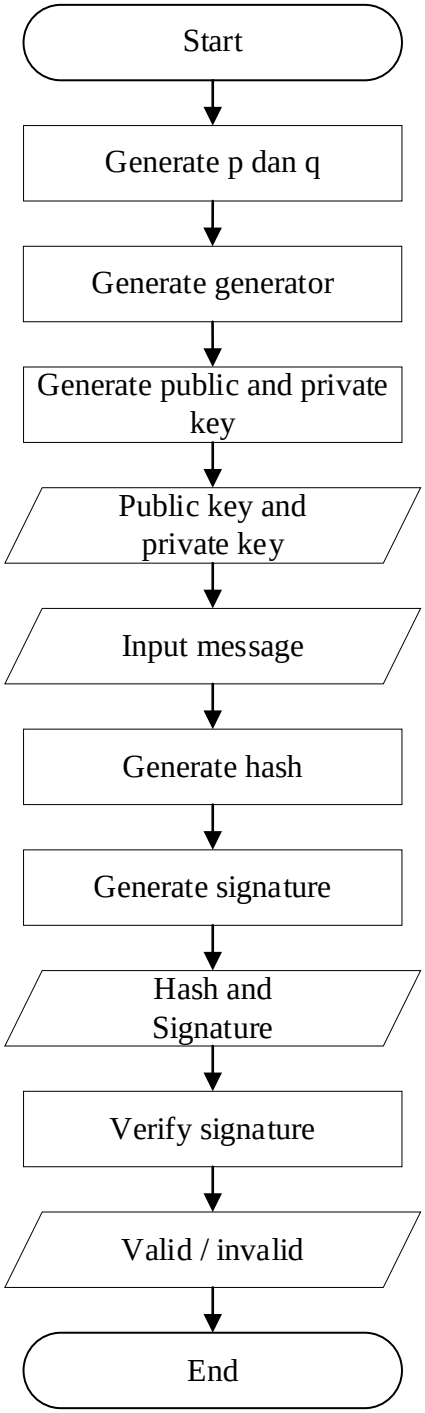
Gambar 3. 5 flowchart enkripsi dan dekripsi algoritma ElGamal

Tabel 3. 7 pseudocode enkripsi dan dekripsi algoritma ElGamal

Pseudocode

```
Program ElGamal_Encryption_and_Decryption

deskripsi:
var p, q, generator, private_key, public_key : ZZ

Algoritma:
generate_keys(panjang_kunci)
    q ← GenGermainPrime_ZZ(panjang_kunci-1, 80)
    p ← 2*q + 1
    generator ← ZZ(1)
    WHILE (generator == ZZ(1))
        ZZ h ← RandomBnd(p)
        WHILE(h == ZZ(0))
            h ← RandomBnd(p)
         ENDWHILE
        generator ← PowerMod(h, (p-1)/q, p)
    ENDWHILE
    private_key ← RandomBnd(q)
    WHILE (private_key == ZZ(0))
       private_key ← RandomBnd(q)
       public_key ← PowerMod(generator, private_key, p)
    ENDWHILE
    OUTPUT (p, q, generator, private_key, public_key)

encrypt(message)
    k ← RandomBnd(q)
    WHILE(k == ZZ(0))
        k ← RandomBnd(q)
    ENDWHILE
    c1 ← PowerMod(generator, k, p)
    c2 ← (message * PowerMod(public_key, k, p)) % p
    cipher
    cipher.append(c1)
    cipher.append(c2)
    return cipher
decrypt(cipher)
    c1 ← cipher[0]
    c2 ← cipher[1]
    inverse_of_c1 ← InvMod(c1, p)
    decrypted_message ← (c2 * PowerMod(inverse_of_c1,
    private_key, p)) % p
    return decrypted_message

main()
    INPUT(panjang_kunci)
    generate_keys(panjang_kunci)
    INPUT(message)
    OUTPUT(message)
    cipher ← encrypt(message)
    OUTPUT(cipher[0], cipher[1])
    decrypted_message ← decrypt(cipher)
    OUTPUT(decrypted_message)
```

b. Tanda tanga digital



Gambar 3. 6 flowchart tanda tangan digital algoritma ElGamal

Tabel 3. 8 pseudocode tanda tangan digital algoritma ElGamal

Pseudocode

```
Program ElGamal_Digital_Signatur

deskripsi:
var p, q, generator, private_key, public_key : ZZ

Algoritma:
generate_keys(panjang_kunci)
    q ← GenGermainPrime_ZZ(panjang_kunci-1, 80)
    p ← 2*q + 1
    generator ← ZZ(1)
    WHILE (generator == ZZ(1))
        ZZ h ← RandomBnd(p)
        WHILE(h == ZZ(0))
            h ← RandomBnd(p)
         ENDWHILE
        generator ← PowerMod(h, (p-1)/q, p)
    ENDWHILE
    private_key ← RandomBnd(q)
    WHILE (private_key == ZZ(0))
      private_key ← RandomBnd(q)
      public_key ← PowerMod(generator, private_key, p)
    ENDWHILE
    OUTPUT (p, q, generator, private_key, public_key)

decimal_value(ch)
    SWITCH(ch)
        case '0': return 0
        case '1': return 1
        case '2': return 2
        case '3': return 3
        case '4': return 4
        case '5': return 5
        case '6': return 6
        case '7': return 7
        case '8': return 8
        case '9': return 9
        case 'a': return 10
        case 'b': return 11
        case 'c': return 12
        case 'd': return 13
        case 'e': return 14
        case 'f': return 15
    ELSE:return-1
    END SWITCH
convert_hexa_decimal_string_to_ZZ (num)
   result ← ZZ(0)
     FOR (int i = 0; i < num.length(); i++)
       result ← result*16 + decimal_value(num[i])
     END FOR
     return result
generate_hash(input:string)
     SHA1 checksum
```

```
      checksum.update(input)
   const string hash ← checksum.final()
    hash_val ← convert_hexa_decimal_string_to_ZZ(hash)
   return hash_val
sign(message){
    s ← ZZ(0), r, hash ← generate_hash(message)
    OUTPUT(message)
    OUTPUT(hash)
    WHILE(s == ZZ(0))
         k ← RandomBnd(p-1)
        WHILE(k == ZZ(0) or k == ZZ(1) or GCD(k, p-1) !=
   ZZ(1))
            k ← RandomBnd(p-1)
        ENDWHILE
        r ← PowerMod(generator, k, p)
    ENDWHILE
    s ← (((hash - (private_key*r) % (p-1)) % (p-1)) *
   InvMod(k, p-1)) % (p-1)
    OUTPUT(r, s)
    Vec<ZZ> signature
    signature.append(r)
    signature.append(s)
    return signature

verify_signature(Vec<ZZ> signature, string message)
    ZZ r = signature[0], s = signature[1]
    OUTPUT(r, s)
    IF(r <= ZZ(0) or r >= p or s <= ZZ(0) or s >= p-1)
   THEN
        OUTPUT("Signature is not valid")
        return;
    ENDIF
    g^H(m) ← (y^r).(r^s) (mod p)
    ZZ hash ← generate_hash(message)
    ZZ lhs ← PowerMod(generator, hash, p)
    ZZ rhs ← (PowerMod(public_key, r, p) * PowerMod(r, s,
   p)) % p

    OUTPUT(hash)
    OUTPUT(lhs)
    OUTPUT(rhs)

    IF(lhs == rhs)
        OUTPUT("Signature is valid")
    ELSE
        OUTPUT("Signature is invalid")
    ENDIF

main()
    INPUT(panjang_kunci)
    generate_keys(panjang_kunci)

    INPUT(message)
    signature = sign(message)
    verify_signature(signature, message)
```

3.3 Analisis Data

Analisis data dilakukan dalam tiga tahapan, yang meliputi:

1. Pengumpulan data (*Data collecting)*

   Pengumpulan data dilakukan dengan menggunakan sistem enkripsi dan dekripsi yang telah dibangun. Data masukan adalah data angka dengan beberapa ukuran data yang berbeda. Nilai keluaran yang didapatkan berupa waktu pembangkitan kunci untuk enkripsi dan dekripsi, waktu enkripsi, waktu dekripsi, waktu pembangkitan kunci tanda tangan digital, serta waktu pembuatan tanda tangan digital ditampilkan dalam satuan waktu *(s)*. Nilai keluaran inilah yang selanjutnya akan digunakan sebagai data penelitian.

2. Penyajian data (*Data Display)*

   Penyajian data dilakukan dengan menampilkan data dalam bentuk tabel secara keseluruhan. Penyajian data untuk setiap parameter yang diuji dalam penelitian ini akan menggunakan grafik untuk menyajikan data secara lebih rinci. Data setiap parameter dari algoritma yang diuji ditampilkan secara terpisah. Penyajian data dalam penelitian ini berfungsi untuk memudahkan peneliti untuk memahami data serta mempermudah peneliti untuk melakukan analisis serta perbandingan terhadap setiap data.

3. Penarikan kesimpulan (*conclusion drawing)*

   Kesimpulan merupakan jawaban rumusan masalah yang dirumuskan di awal penelitian. Kesimpulan dalam penelitian ini berupa nilai perbandingan dari setiap algoritma yang diteliti yaitu RSA, ECC, dan ElGamal sesuai dengan parameter yang telah ditetapkan. Kesimpulan menunjukan perbandingan nilai performa setiap algoritma berdasarkan parameter yang diuji.