Galois Field with Power of Prime Order in Cryptography

Dewita Sonya Tarabunga
13515021
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
dewitast20@gmail.com

***Abstract*— *Elliptic Curve Cryptography* was claimed to be more power-efficient system to use rather than usual integer system in message encryption and decryption and needed less key length to provide similar security. In this paper, the writer will try to implement another field system to use i.e Galois field and compare it with *Elliptic Curve Cryptography*. Galois field will surely needed less key than ECC because we can choose much smaller prime. Basically, the idea is to use usual operation (multiplication) but on higher dimensions. This paper will implement Galois Field in ElGamal algorithm.**

***Keywords*— Cryptography, Elliptic Curve Cryptography, Galois Field, El-Gamal**

I. Introduction

The notion of cryptography has developed throughout history and currently has a lot of meanings. But, the simplest and first meaning of cryptography is a system that provide secure communication in the presence of malicious third parties, known as adversaries [1]. That is, when two or more people want to share messages between each other, and they do not want another person outside their group to know anything about the messages, they will use cryptography to ensure the security of the messages.

Encryption and decryption is two main processes of cryptography. Encryption is a process to turn messages to some meaningless sentence. And on the other side, decryption is a process to turn that meaningless sentence back to original messages. By using the right key and the right system, only the right people will have access to the original message.

Throughout history, there are two widely used cryptosystem, namely symmetric key cryptography and public key cryptography. Both have their own advantages and disadvantages. There are numbers of encryption and decryption algorithm that are developed on both systems. RSA and El-Gamal [2] is examples of public key cryptography. On this paper, we will only focus on ElGamal algorithm. That is because ElGamal is a cryptographic algorithm that works on finite field.

One of the disadvantages of symmetric key cryptography is the needs to maintain a lot of key. This is because if some person *A* wants to communicate to *B* and *C*, and *A* does not want *B* to have access to the messages that *A* send to *C*, then he has to have two separate different keys. The more people *A* wants to communicate to, the more keys he has to manage. With public key cryptography, this would not happen because the key used to communicate is public, thus every one just have to maintain one private key for theirselves and the rest is public.

Elliptic Curve is an algebraic structure that forms a finite field under the geometric operations of tangent line [3]. It also works as an underlying structure for public key cryptography. El-Gamal formerly implemented in $\mathbb{Z}_p$, group of integers modulo p with usual addition and multiplication operation. With *Elliptic Curve* field, due to its complex operation, it is claimed that *Elliptic Curve Cryptography* will provide equivalent security with less key length, thus lessen the cost of encryption and decryption without lessen the security.

The field of integer modulo $p, \mathbb{Z}_p$ is an example of Galois Field that will be discussed more later. The idea on this paper is to use Galois Field, but on much higher dimensions. The addition operation can still be used, but the multiplication will be much more complex because the trivial multiplication operation on vector of integers will not work. The author will implement El-Gamal on both fields, *Elliptic Curve* and Galois Field with higher dimensions and compare some aspect.

II. Fundamental Theories

A. *ElGamal*

ElGamal is one of an example of algorithm for public key cryptography created by Taher Elgamal [4]. It is based on another algoritm on key exchange called Diffie-Hellman. Basically, the underlying problem is called the discrete logarithm problem that stated as follow:

Given any group G and any two elements $a, b \in G$, find some $k \in \mathbb{N}$ such that $b^k = a$ respect to operation in group G.

The security of ElGamal algorithm relies heavily upon the underlying group G and its multiplication operation. In reality, any cyclic group is sufficient to be used as the group on ElGamal algorithm. But, the more complex the group and its operation, the stronger the encryption will be because it will increase the cost of computing the multiplication in the group.

There are three main process in ElGamal: key generation, encryption, and decryption that each will be explained.

1. **Key Generation**
   - Define some cyclic group G, and choose one base element $a \in G$,
   - Choose one $k \in \mathbb{N}, 1 \leq k < |G|$,
   - Compute $b = a^k$

   Finally, $G, a,$ and b are the public keys whereas k is the only private key. So, we have to publish G, a, b and keep the private key, k, as a secret.

2. **Encryption**
   The encryption algoritm is based on the public keys already described above.
   - Choose one $l \in \mathbb{N}, 1 \leq l < |G|$,
   - Calculate $c_1 = a^l$
   - Let $m \in G$ is the message, then compute $c_2 = mb^l$

   Finally, the pair (c_1, c_2) is the encrypted message.

3. **Decryption**
   - Compute $m = c_2 {c_1}^{-k}$

   The algorithm works because

$$\begin{aligned} c_2 {c_1}^{-k} &= mb^l a^{-kl} \\ &= ma^{kl} a^{-kl} \\ &= m \end{aligned}$$

B. *Group*

In mathematics, group is a set of elements with some defined binary operation that satisfies four conditions, namely *closure, associativity, identity, invertibility* [5]. Group is important in this paper because a set of elements that will be used along with ElGamal needs to be a group, otherwise the algorithm will not work. The four condition of group G with operation $a \cdot b = ab$ is:

1. **Closure**
   The operation $\cdot$ of group G has to be closed, that is, for every two elements $a, b \in G$, then $ab \in G$. In other words, ab must also be an element of G.

2. **Associativity**
   The operation $\cdot$ also must be associative, that is, for every three elements $a, b, c \in G$, then $(ab)c = a(bc)$. The order of the operation does not matter and can not change the final result. Because of that, we can write $(ab)c = a(bc) = abc$.

3. **Identity**
   There must be an element $e \in G$ such that for every $a \in G$, then $ae = ea = a$. That element is called the identity element of G. Furthermore, this element is unique.

4. **Inverse**
   For every element $a \in G$, there exist $b \in G$, such that $ab = ba = e$. This element is called the inverse of a and also unique in G. We will denote the inverse of a as a^{-1}. It is trivial to prove that $e^{-1} = e$.

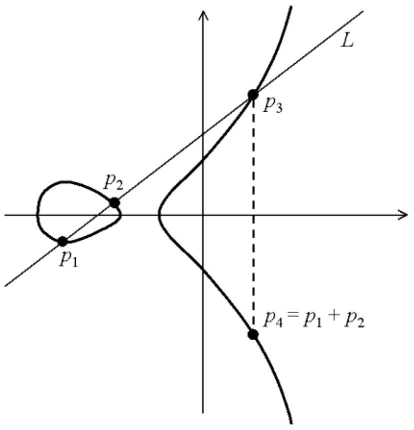
One very important family of group is a cyclic group. A group is cyclic if and only if there exist $a \in G$, such that if $b \in G$, then there exist $k \in \mathbb{N}, 1 \leq k \leq |G|$, such that $b = a^k$. Such an element a is called a generator of G. In fact, if $m = |G|$, then every elements of G that have m as their order is a generator of G.

C. *Elliptic Curve Cryptography*

Elliptic Curve Cryptograhy is a system based on algebraic structure made by *Elliptic Curve* over finite field [3]. It has been proved that the defined discrete *Elliptic Curve* along with its operation that based on tangent line satisfies the four axiom of group. It also has been proved that the group created is a cyclic group. Thus, it can be used as an underlying system for ElGamal algorithm.

There was a claim that, due to the complexity of operation in *Elliptic Curve,* then it can provide equivalent security as usual group with much less key length, thus decreasing the needs of power and the cost of computing. It also has been widely used, especially in machine that can only provide limited power, such as handphone.

Elliptic curve is defined as plane curve over an integer modulo p, which consist of the points satisfying the equation $y^2 = x^3 + ax + b$, along with a *point at infinity* as the identity element.



Gambar 1 *ECC* Addition
Sumber: https://i.imgur.com/EdPk12E.gif

The group operation is showed above. To add two points together, first draw a line connecting both points. The line will intersect the curve in one other point, beside those two points. Reflect that point about the y-axis to get the result. With that operation and some result from geometry, it can be proved that *Elliptic Curve* is closed, associative, has identity element (namely, ∞), and every element has inverse.

Formally, if we want to compute $R(x_R, y_R) = P(x_p, y_p) + Q(x_Q + y_Q)$, then

$$x_R = k^2 - x_P - x_Q$$
$$y_R = k(x_P - x_R) - y_P$$

where

$$k = (y_P - y_Q)(x_P - x_Q)^{-1}$$

There is some difference of formula if we want to add some point to itself, $P + P = 2P$. This is because there is no definition of inverse of 0 when we want to calculate k. So, the formal definition of above problem is as follow.
Let $R(x_R, y_R) = 2P(x_p, y_p)$, then

$$x_R = k^2 - 2x_p$$
$$y_R = k(x_P - x_R) - y_P$$

where

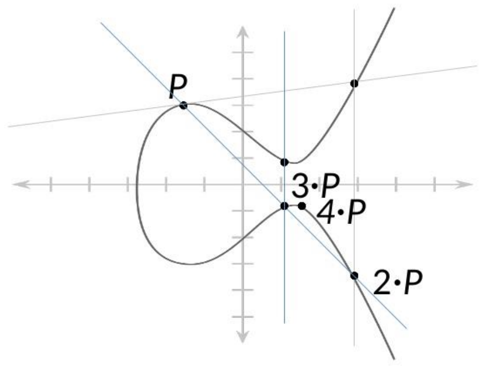
$$k = (3{x_p}^2 + a)(2y_P)^{-1}$$

where a is the coefficient of x^3 of the curve equation.

After we define the addition of point to itself, one most important operation is the multiplication of point to integer, kP, where $k > 2$. Using the definition of $2P$ and $P + Q$, we can easily multiplication of kP using only those two operation. For example, if we want to compute $5P$, then

$$5P = (P + 2(2P))$$

It can be seen that the only operation involved is of two kind, i.e. $P + Q$ and $2P$.



Gambar 2 *ECC* Multiplication
Sumber: www.slideshare.net/MartijnGrooten/

D. Galois Field

Field is an algebraic strucure that is much stronger that group. Basically, field F is a set of element that has two defined operation, namely addition and operation such that F is an additive group (forms a group along with addition operation) [5]. Let 0 be the identity element of F as an additive group. Then, $F - \{0\}$ must form a multiplicative group (forms a group along with multiplication operation). Furthermore, F must satisfy the distributive condition, that is, for any $a, b, c \in F$, then

$$a(b + c) = ab + ac$$

It is known that the order of any field can be finite or infinite. $\mathbb{R}$, the field of rational number along with usual addition and multiplication is an example of field with infinte order. On the other side, $\mathbb{Z}_p$ the field of integer modulo p along with modular addition and multiplication is an example of field with finite order.

A great result due to Galois, show that the order of field with finite order can only be of the form p^k, where $p, k \in \mathbb{N}$ is a natural number and p is a prime number. Furthermore, Galois also show that the number of field with order p^k for every $p, k \in \mathbb{N}$, p prime is only one, up to isomorphisms. That is, given any $p, k \in \mathbb{N}$,if there are any two fields F anf G such that $|F| =$

$|G| = p^k$, then $F \cong G$.

Galois Field is a finite field. Because of the result above, for every possible order of finite field, there is exist exactly one Galois Field of that order. Galois Field with order p^k denoted as $\mathbb{Z}_{p^k}$, defined as vector that consist of k elements of $\mathbb{Z}_p$. Now, we are left with the definition of addition and multiplication of Galois Field.

The addition of Galois Field is pretty trivial, i.e. to sum the vector element-wise. If $a = (a_1, a_2, \dots, a_k), b = (b_1, b_2, \dots, b_k) \in \mathbb{Z}_{p^k}$, then

$$a + b = (a_1 + b_1, \dots, a_k + b_k)$$

with + denote the addition in $\mathbb{Z}_p$.

For the multiplication, maybe the most trivial operation that first comes up is element-wise multiplication. Unfortunately, that multiplication do not satisfy the field conditions. For example $(0,1),(1,0) \in \mathbb{Z}_{2^2} - \{(0,0)\}$ but the multiplication is $(0,1)(1,0) = (0,0) \notin \mathbb{Z}_{2^2} - \{(0,0)\}$

The definition of multiplication that is finally used is polynomial multiplication. Map elements of field to coefficient of some polynomial of degree less than k. Obviously, this mapping is bijective. To ensure the closure, the result of multiplication must be taken modulo some irreducible polynomial of degree k. This operation has been proved to satisfy all the field conditions.

It has been proved that the multiplicative group of any Galois field is in fact, a cyclic group. Because of that reason, Galois Field can be applied to the ElGamal algorithm as well as *Elliptic Curve.*

Galois Field has been used in one of the most famous symmetric key algorithm named AES. AES uses the Galois Field Z_{2^8} with $x^8 + x^4 + x^3 + x^2$ as its modulo irreducible polynomial. This field also called Rijndael Field in tribute to the algorithm creator, Rijndael.

III. IMPLEMENTATION

To compare the usage of the two algebraic structure, both algebraic structure, *Elliptic Curve* and *Galois Field* will be implemented in ElGamal algorithm. The selection of the algoritm is due to its properties that depends a lot to its underlying algebraic structure. The implementations will not include the message encoding (the conversion of plaintext to element of algebraic structure). So, the analysis will be based only of the encryption and decryption of the elements of algrebaic strucutre.

A. *Elliptic Curve*

To implement *elliptic curve*, the most important thing is to implement Point because *elliptic curve* works on two dimensional point. Below are the implementation of some important method on Point.

```
  def negate(self):
       self.ordinat = (self.modulo
- self.ordinat) % self.modulo
       return self

  def gradient(self, point):
       dx    =    (self.absis    -
point.absis) % self.modulo
       if dx<0:
              dx += self.modulo
       dy    =    (self.ordinat    -
point.ordinat) % self.modulo
       if dy<0:
              dy += self.modulo
       if dx==0:
              return None
       return   (dy   *   invmod(dx,
self.modulo)) % self.modulo

  def equal(self, point):
       return    self.absis    ==
point.absis  and  self.ordinat  ==
point.ordinat  and  self.modulo  ==
point.modulo

  def is_inf(self):
       return self.modulo == -1

  def print(self):
       res = "(" + str(self.absis)
+ "," + str(self.ordinat) + ")"
       return res
```

Only the very basic operations are covered on above implementatoion. More crucial methods will be covered on another class implementation, the *elliptic curve*. Below is the *elliptic curve* implementation along with its method

```
       def contains(self, point):
              lhs = (point.ordinat
** 2) % self.modulo
              rhs = (point.absis **
3) % self.modulo
              rhs %= self.modulo
              rhs += (point.absis *
self.coef) % self.modulo
              rhs %= self.modulo
              rhs += self.const
              rhs %= self.modulo
              return (rhs==lhs)

       def gradient(self, point):
              dx = (3 * point.absis
*  point.absis  +  self.coef)  %
self.modulo
              dy = (point.ordinat +
point.ordinat) % self.modulo
            if dy==0:
                  return None
            dy    =    invmod(dy,
self.modulo)
            return  (dx  *  dy)  %
self.modulo

      def square(self, point):
            if point.is_inf():
                  return point
            grad                  =
self.gradient(point)
            if grad == None:
                  return INF
            absis = (grad * grad
-  point.absis  -  point.absis)  %
self.modulo
            if absis<0:
                  absis       +=
self.modulo
            ordinat   =   (grad   *
(point.absis     -     absis)     -
point.ordinat) % self.modulo
            if ordinat<0:
                  ordinat     +=
self.modulo
            return   Point(absis,
ordinat, self.modulo)

      def    add(self,    point1,
point2):
            if
point1.equal(point2):
                  return
self.square(point1)
            if point1.is_inf():
                  return point2
            if point2.is_inf():
                  return point1
            grad                  =
point1.gradient(point2)
            if grad == None:
                  return INF
            absis = (grad * grad
- point1.absis - point2.absis)  %
self.modulo
            if absis<0:
                  absis       +=
self.modulo
            ordinat   =   (grad   *
(point1.absis    -    absis)     -
point1.ordinat) % self.modulo
            if ordinat<0:
                  ordinat     +=
self.modulo
            return   Point(absis,
ordinat, self.modulo)

      def    multiply(self,    k,
point):
            ans = INF
            rest = point
            while k>0:
                  if k&1:
                        ans    =
self.add(ans, rest)
                  rest         =
self.square(rest)
                  k >>= 1
            return ans

      def solve(self, x):
            rhs                =
(x*x*x+x*self.coef+self.const)   %
self.modulo
            if self.modulo==2:
                  return rhs
            if       modpow(rhs,
(self.modulo    -    1)    //    2,
self.modulo) != 1:
                  return None
            ans = 0
            while  (ans  *  ans)  %
self.modulo != rhs:
                  ans += 1
            return ans

      def print(self):
            res = "y^2 = x^3 + "
+  str(self.coef)  +  "x  +  "  +
str(self.const)  +  "  mod  "  +
str(self.modulo)
            return res
```

B. Galois Field

In the other hand, because Galois Field multiplication can best be seen as polynomial multiplication, thus it is the most important aspect to be implemented. Below is the implementation of the most important method of Polynomial.

```
  def truncate(self):
      while  len(self.coef)>1  and
self.coef[-1] == 0:
            del self.coef[-1]
      self.degree                =
len(self.coef) - 1

  def decrement_degree(self):
      self.degree -= 1
      del self.coef[-1]

  def shift(self):
      if self.coef[-1] > 0:
            self.degree += 1
            self.coef.insert(0,
0)

  def add(self, pol):
      degree   =  max(self.degree,
pol.degree)
      self.coef  +=  [0  for  _  in
range(degree - self.degree)]
      pol.coef  +=  [0  for  _  in
range(degree - pol.degree)]
      self.coef  =  [(self.coef[i]
+ pol.coef[i]) % self.modulo for i
in range(degree + 1)]
        self.degree = degree
        self.truncate()
        pol.truncate()

    def power(self, k):
        if k == 1:
            return
        tmp = self.copy()
        self.coef = [0]
        self.degree = 0
        while k > 0:
            if (k&1) == 1:
                self.add(tmp)
            tmp.add(tmp)
            k >>= 1

    def subtract(self, pol):
        degree = max(self.degree,
pol.degree)
        self.coef += [0 for _ in
range(degree - self.degree)]
        pol.coef += [0 for _ in
range(degree - pol.degree)]
        self.coef = [(self.coef[i]
- pol.coef[i] + self.modulo) %
self.modulo for i in range(degree
+ 1)]
        self.degree = degree
        self.truncate()
        pol.truncate()

    def multiply(self, pol):
        degree = self.degree +
pol.degree
        coef = [0 for _ in
range(degree + 1)]
        for i in range(self.degree
+ 1):
            for j in
range(pol.degree + 1):
                coef[i + j] +=
(self.coef[i] * pol.coef[j]) %
self.modulo
                coef[i + j] %=
self.modulo
        self.coef = coef[:]
        self.degree = degree

    def mod(self, pol):
        if self.degree <
pol.degree:
            return
        res =
Polynomial(self.modulo,
[self.coef[i] for i in
range(pol.degree)])
        base =
Polynomial(self.modulo)
        cpol = pol.copy()
        cpol.power(invmod(cpol.coef
[-1], self.modulo))
        cpol.decrement_degree()
        base.subtract(cpol)
        if self.coef[pol.degree] >
0:

        base.power(self.coef[pol.de
gree])
            res.add(base)

        base.power(invmod(self.coef
[pol.degree], self.modulo))
        md =
Polynomial(self.modulo, base.coef)
        for i in range(pol.degree +
1, self.degree + 1):
            md.shift()
            if md.degree ==
pol.degree:
                k = md.coef[-
1]

        md.decrement_degree()
                base.power(k)
                md.add(base)

        base.power(invmod(k,
self.modulo))
            if self.coef[i] > 0:

        md.power(self.coef[i])
                res.add(md)

        md.power(invmod(self.coef[i
], self.modulo))
        self.coef = res.coef[:]
        self.degree = res.degree

    def print(self):
        string = ''
        for i in range(self.degree
+ 1):
            tmp =
str(self.coef[i])
            if i>0:
                tmp += 'x^' +
str(i)
            if i < self.degree:
                tmp += ' + '
            string += tmp
        print(string)
```

It can be seen that some of the most important operations are already covered on polynomial implementation. That is because Galois Field multiplication is basically just polynomial multiplication. That makes the implementation of Galois Field much more easier because it can directly uses the implementation above. Below is the implementation of Galois Field.

```
    def copy(self):
        return
GaloisField(self.prime,
self.power, self.poly.coef,
self.modulo.coef)
```

```
  def add(self, other):
       self.poly.add(other.poly)

  def multiply(self, other):
       self.poly.multiply(other.po
ly)
       self.poly.mod(self.modulo)

  def powerr(self, k):
       k %= self.order
       if k == 1:
              return
       tmp = self.copy()
       self.poly.coef = [1]
       self.poly.degree = 0
       while k > 0:
              if (k&1) == 1:

       self.multiply(tmp)
              tmp.multiply(tmp)
              k >>= 1

  def print(self):
       self.poly.print()
```

The implementation is much shorter because it has a polynomial as one of its attribute, thus it can call needed polynomial method on that polynomial atrribute.

IV. Analysis

Analysis will be done on every stage, key generation, encryption and decryption.

A. *Key Generation*

First thing to do before encryption and decryption is to define one group *G* to work with. In *elliptic curve,* this means defining some *elliptic curve* equation in the form

$$y^2 = x^3 + ax + b$$

along with its modulo.

And for Galois Field, this can be done only by defining the order of the group, since then we can uniquely determine the prime and its power. One notable thing is determining Galois Field is much easier than determining *elliptic curve*. Determine the prime modulo for large numbers is equivalent to finding large prime numbers which is very hard compared to finding a rather small prime but with large power. The hardest part of defining the Galois Field is to determine the modular polynomial because the polynomial must be irreducible. But, there is already some algorithm or criterion to produce irreducible polynomial for any degree. The most famous one is Eisenstein Criterion.

For the analysis, the group for *elliptic curve* is

$$y^2 = x^3 + 2x + 4 \; mod \; 257$$

and for Galois Field is the order is $256 = 2^8$, and the modular irreducible polynomial is

$$x^8 + x^4 + x^3 + x + 1$$

That polynomial also used on Rijndael Field in AES algorithm. Both orders are chosen because the two numbers only differ by one, although the elements in *elliptic curve* would not be exactly 257.

The next step is to choose one base point. Base point must be one of the element of the group. In this case, Galois Field win the competition again because choosing one element of Galois Field is more simple than of *elliptic curve.* For Galois Field, we only need to randomize integer from 0 to $p-1$, k times, where p^k is the field order. Whereas for *elliptic curve,* we have to choose one point that satisfy the equation, which is not that trivial. To continue, the base point for Galois Field is

$$x^6 + x^3 + x + 1$$

which is generated and for *elliptic curve* is $(2,4)$.

The next step is to choose random number as the private key. It can be any number, but for this paper, we will choose $k = 15$ as private key (for no reason, it's just a random number). Private keys usually are smaller than the group order. But, in *elliptic curve* case, the order of the group created by some equation and its modulo is not really trivial, which is one of the disadvantages of *elliptic curve*.

Last step is to multiply the base point k times to get an element of group as the public key. Both group will be compared based on its complexity in Big-Oh notation. For *elliptic curve,* it uses mathematics to multiply two points, thus the complexity is $O(1)$, and the overall complexity to multiply base point k times to itself is $O(\log k)$ by using divide and conquer. For Galois Field, the complexity of the above implementation on polynomial multiplication is $O(n^2)$, but the best known algorithm for polynomial multiplication is $O(n \log n)$, by using Fast Fourier Transform. So, the overall complexity is $O(n \log n \log k)$, where n is the polynomial degree. It can be seen that the complexity is much worse than that of *elliptic curve.*

The result of multiplication for Galois Field is

$$x^7 + x^5 + x^3 + x + 1$$

and for the *elliptic curve* is

$$(35, 162)$$

which both equivalent to $30 \; mod \; 257$.

By using timeit library from Python, the time to do multiplication for *elliptic curve* and Galois Field is 1.36736s and 17.91904s respectively.

B. *Encryption*

The first step to encrypt a message is to choose one random integer less than the group order. In this paper, we are going to use $l = 21$ (again, for no reason, just another random number). Just like private key, it can actually be any number without any restriction other than less than group order. In fact, any number larger than group order can be used, but is worthless.

The next step is to calculate the multiplication of base point to itself l times as before. Because the complexity and time measurement are already done before, it will not be explained again here. This shows that *elliptic curve* win in this part because its complexity that is much smaller. The result of the multiplication for Galois Field is

$$x^7 + x^5 + x^3$$

and for *elliptic curve* is

$$(255, 136)$$

which both equivalent to $249 \; mod \; 257$. Above results will be one of the ciphertext.

The last step is to compute the second part of the ciphertext (ElGamal produce a pair of ciphertext for a message). This part is similar to the previous part and also to the computation of public key. But before, we have to determine the message in a form of group element. For this purpose, we will determine it in random. Let's say that the message for Galois Field is

$$x^7 + x^6 + x^5 + x^4 + x + 1$$

and for *elliptic curve* is $(77{,}197)$. Both are obtained by random.

Now, it is time to compute the second part of the ciphertext. For Galois Field it's

$$x^5 + x^4 + x^3 + x^2 + 1$$

and for *elliptic curve* it is $(64, 176)$. The analysis and time measurement are similar to the previous one.

C. *Decryption*

This part is to show that the overall algorithm really works, i.e., that the plaintext acquired from the decryption process is the same as the original message. The process that is used is again, multiplication. This makes *elliptic curve* much faster with its smaller multiplication complexity.

The first and only part is to multiply the second part of the ciphertext pair to the k-th power of the inverse of the first part of the ciphertext pair. The result for Galois Fiels is below

$$x^7 + x^6 + x^5 + x^4 + x + 1$$

and for *elliptic curve* is $(77, 197)$. It can be seen from the previous section that the above result is exactly the original message.

D. *Other*

There are already a lot of encode/decode method to convert message to a point in *elliptic curve*. For example, there is Koblitz method that maps one character to one point in *ellipti curve* and there also other better algorithms. To use Galois Field in ElGamal, there must be some encoding algorithm that maps a message to a Galois Field element, and it has to be able to work on any Galois Field. The most trivial way is to divide message to blocks and every character mapped into its ASCII. But, this way is too simple and easier to crack. Therefore, we have to find another good algorithm before we are able to implement Galois Field in real world situation.

Another consideration that made *Elliptic Curve* used widely is the claim that it is too complex and not really well-studied. So, there is no special property that makes the discrete logarithm easier to break. On the other side, Galois Field is a very well-known field in mathematics and cryptography. So, there could be any properties that can be used to break the discrete logarithm problem although there has not been any proof.

VI. Conclusion

After analyzing and comparing some cryptographic aspect between the usage of Galois Field and *Elliptic Curve* in ElGamal algorithm, it can be seen that both algebraic structure have their own advantages and disadvantages. Although both algebraic strucure satisfy the conditions to be used on ElGamal algorithm.

In term of computing cost, *Elliptic Curve* has a much better speed compared to the other algorithm. Whereas Galois Field gives worse complexity and thus is not suitable for devices that has less power source such as mobile phone. But, the key generation and key management process of Galois Field is easier and better than that of *Elliptic Curve,* because key are relatively small for group with similar order.

There is still a lot of improvement that can be made on the implementation of both algebraic structure. For example, we can use Tonelli-Shanks algorithm to find modular square root to be used on *Elliptic Curve*. Other example is to use Fast Fourier Transform for polynomial multiplication, reducing the complexity from $O(n^2)$ to $O(n \log n)$, a much better complexity.

VII. ACKNOWLEDGMENT

I would like to thank God for His grace and kindness, I can finish this paper for Cryptography class with His help. I also want to thank Mr Rinaldi Munir for teaching us in this class for one semester. I would like to thank you my family and friends, which have helped me finish this paper. Without all of them, I would be having a trouble finishing this paper.

REFERENCES

[1] Rinaldi Munir, Slide Pengantar Kriptografi 2018
[2] Rinaldi Munir, Slide Algoritma ElGamal 2018
[3] Rinaldi Munir, Slide *Elliptic Curve Cryptography* 2018
[4] Taher ElGamal, ‘A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithm’
[5] Thomas W. Hungerford, ‘Abstract Algebra: An Introduction’

ORIGINALITY STATEMENT

I hereby declare that this paper is my own writing, not an adaptation, nor translation of other’s paper and nor plagiarism.

Bandung, 14 May 2018

[illegible]

Dewita Sonya Tarabunga
13515021