Group security using ECC

Cryptography G21, Himanshu Pal - *S20190010064*, Yash Shukla - *S20190010198*

Abstract

As per the survey conducted by ITU in 2019, 53.6% of global populations are using the internet for their day to day work. That is why data transmission security is a major concern. There are a lot of cryptographic algorithms but encryption using ECC gives the improved security with smaller size.

This paper has presented the group security using ECC. Group security is using the m-gram selection. We want to reduce the time for the encryption and decryption using ECC taking the advantage of common grams. This paper will contrast the difference in processing time of the traditional ECC algorithm and the algorithm that is being implemented using common-gram technique. The fact that is being taken into advantage in this section is that the extended-common gram selection decreases the time for computation.

Index Terms

GFGS, m-gram ECC, EMS, Group security

I. Defined Goals

The main goal of the paper is to reduce the processing time for encryption and decryption using ECC.
The additional process should not be tedious and easy to implement.
We are using the diagram technique to take the two sequential bit information and then encrypt it.

II. Contributions

Himanshu Pal

- Removing the stopwords from the given plain-text
- Designing GFGS algorithm for getting the frequent common gram from the given setof inputs.
- Converting the result obtained from the above process into biword creation.
- ECC decryption

Yash Shukla

- ECC encryption
- ECC signature
- Integration of the ECC algorithm with the GFGS implementation
- Verification of message

III. ECC

ECC, an alternative to RSA, is a strong cryptography method. It creates security between key pairs for public key encryption using elliptic curve mathematics.

RSA uses prime numbers instead of elliptic curves for something similar, but ECC has lately gained prominence due to its reduced key size and ability to maintain security. This trend is likely to continue as the requirement for devices to stay safe grows due to the increasing size of keys, putting a strain on finite mobile resources. This is why it is critical to comprehend elliptic curve cryptography in context.

Unlike RSA, ECC based its approach to public key cryptography systems on how elliptic curves are algebraically organised over finite fields. As a result, ECC generates keys that are mathematically more difficult to crack. As a result, ECC is regarded as the next generation implementation of public key cryptography, as well as being more secure than RSA.

It also makes sense to use ECC in order to maintain high levels of speed and security. This is because ECC is becoming more widely used as websites aim for improved online security in client data while also improving mobile optimization. As more sites use ECC to encrypt data, there will be a larger demand for this type of concise tutorial to elliptic curve cryptography. For contemporary ECC applications, an elliptic curve is a plane curve over a finite field composed of points fitting the equation: $y^2=x^3 + ax + b$.

IV. GFGS

Generalized frequent-common gram selection.
In general, each piece of information is taken into account independently during data transmission and processing. Individual information selection and processing necessitates extra searching and processing time. Instead of processing individual components, a group of often occurring elements known as m-grams is discovered, and all operations are applied to the common grammes. The nominated frequent common grammar acts as a representation for the group of items. We looked at a 2-gram approach that searches for the set of items using a set of two bits termed di-bits. In this case, we used the ECC technique on a grouping of bits rather than individual parts to ensure group security while reducing processing time. It is represented by the UCI repository's wholesalers dataset.

V. Proposed Work

The algorithm will be searching for the suspicious elements in the group of elements based on the frequency of the common characters. For the implementation purpose we are doing it using the biword selection for the suspicious elements.

After getting the elements in the first part, we will do the encryption of these biwords instead of the actual plain text using ECC algorithm.
It has been observed that the time taken for the implementation using the classical ECC algorithm is much more. But as the aim of paper to reduce the time for a group of plain-text keeping the same level of the security.

Biword has greatly able to bring down the time to apply ECC on group elements and also reducing the external memory consumption.

VI. Implementation

Each step is explained with its code.

A. Point Class

This Class is defined for representing the point on the elliptic curve and for the ease of better understanding for the user

```python
class Point(object):
    #Construct a point with two given coordindates.
    def __init__(self, x, y):
        self.x, self.y = x, y
        self.inf = False

    #Construct the point at infinity.
    @classmethod
    def atInfinity(cls):
        P = cls(0, 0)
        P.inf = True
        return P

    def __str__(self):
        if self.inf:
            return 'Inf'
        else:
            return '(' + str(self.x) + ',' + str(self.y) + ')'

    def __eq__(self,other):
        if self.inf:
            return other.inf
        elif other.inf:
            return self.inf
        else:
            return self.x == other.x and self.y == other.y

    def is_infinite(self):
        return self.inf
```

B. Curve Class

This Class is Used for designing, displaying and performing curve operations like addition of points, multiplication and many more

```
class Curve(object):
    # Set attributes of a general Weierstrass cubic y^2 = x^3 + ax^2 + bx + c over any field.
    def __init__(self, a, b, c, char, exp):
        self.a, self.b, self.c = a, b, c
        self.char, self.exp = char, exp
        print(self)

    def __str__(self):
        # Cases for 0, 1, -1, and general coefficients in the x^2 term.
        if self.a == 0:
            aTerm = ''
        elif self.a == 1:
            aTerm = ' + x^2'
        elif self.a == -1:
            aTerm = ' - x^2'
        elif self.a < 0:
            aTerm = " - " + str(-self.a) + 'x^2'
        else:
            aTerm = " + " + str(self.a) + 'x^2'
        # Cases for 0, 1, -1, and general coefficients in the x term.
        if self.b == 0:
            bTerm = ''
        elif self.b == 1:
            bTerm = ' + x'
        elif self.b == -1:
            bTerm = ' - x'
        elif self.b < 0:
            bTerm = " - " + str(-self.b) + 'x'
        else:
            bTerm = " + " + str(self.b) + 'x'
        # Cases for 0, 1, -1, and general coefficients in the constant term.
        if self.c == 0:
            cTerm = ''
        elif self.c < 0:
            cTerm = " - " + str(-self.c)
        else:
            cTerm = " + " + str(self.c)
        # Write out the nicely formatted Weierstrass equation.
        self.eq = 'y^2 = x^3' + aTerm + bTerm + cTerm
        # Print prettily.
        if self.char == 0:
            return self.eq + ' over Q'
        elif self.exp == 1:
            return self.eq + ' over ' + 'F_' + str(self.char)
        else:
            return self.eq + ' over ' + 'F_' + str(self.char) + '^' + str(self.exp)

    # Double a point on the curve.
    def double(self, P):
        return self.add(P,P)

    # Add P to itself k times.
    def mult(self, P, k):
        if P.is_infinite():
            return P
        elif k == 0:
            return Point.atInfinity()
        elif k < 0:
            return self.mult(self.invert(P), -k)
        else:
            # Convert k to a bitstring and use peasant multiplication to compute the product quickly.
            b = bin(k)[2:]
            return self.repeat_additions(P, b, 1)

    # Add efficiently by repeatedly doubling the given point, and adding the result to a running
    # total when, after the ith doubling, the ith digit in the bitstring b is a one.
    def repeat_additions(self, P, b, n):
        if b == '0':
            return Point.atInfinity()
        elif b == '1':
            return P
        elif b[-1] == '0':
            return self.repeat_additions(self.double(P), b[:-1], n+1)
        elif b[-1] == '1':
            return self.add(P, self.repeat_additions(self.double(P), b[:-1], n+1))
```

C. Curve Over a Finite Field Class

This class Inherits the Curve class for the creation of the curve and defines the curve over the Fp. Input type C = CurveOverFp(0, 1, 1, 2833). This class contains important methods like encrypting, decrypting, inverting and many other functions

```
class CurveOverFp(Curve):
    def __init__(self, a, b, c, p):
        Curve.__init__(self, a, b, c, p, 1)

    def contains(self, P):
        if P.is_infinite():
            return True
        else:
            return (P.y*P.y) % self.char == (P.x*P.x*P.x + self.a*P.x*P.x + self.b*P.x + self.c) % self.
    char

    def add(self, P_1, P_2):
        # Adding points over Fp and can be done in exactly the same way as adding over Q,
        # but with of the all arithmetic now happening in Fp.
        y_diff = (P_2.y - P_1.y) % self.char
        x_diff = (P_2.x - P_1.x) % self.char
        if P_1.is_infinite():
            return P_2
        elif P_2.is_infinite():
            return P_1
        elif x_diff == 0 and y_diff != 0:
            return Point.atInfinity()
        elif x_diff == 0 and y_diff == 0:
            if P_1.y == 0:
                return Point.atInfinity()
            else:
                ld = ((3*P_1.x*P_1.x + 2*self.a*P_1.x + self.b) * mult_inv(2*P_1.y, self.char)) % self.char
        else:
            ld = (y_diff * mult_inv(x_diff, self.char)) % self.char
        nu = (P_1.y - ld*P_1.x) % self.char
        x = (ld*ld - self.a - P_1.x - P_2.x) % self.char
        y = (-ld*x - nu) % self.char
        return Point(x,y)

    def invert(self, P_1):
        if P.is_infinite():
            return P
        else:
            return Point(P.x, -P.y % self.char)

    def generate_points(self):
        points = []
        finite_field = [i for i in range(self.char)]
        x = 0
        while(x < self.char):
            y = ( pow(x, 3) + self.b*x + self.c ) % self.char
            if mod_sqrt(y, self.char) != 0 and y in finite_field:
                points.append(Point(x, mod_sqrt(y, self.char)))
                points.append(Point(x, -mod_sqrt(y, self.char) % self.char))
            x = x + 1
        return points

    def base_point(self, curve_points):
        if curve_points[0].y > curve_points[1].y:
            return Point(curve_points[1].x,curve_points[1].y)
        else:
            return Point(curve_points[0].x,curve_points[0].y)

    def primitive_point(self,curve_points):
        point = random.choice(curve_points)
        return Point(point.x,point.y)

    def random_generator(self):
        return random.randint(0, self.char - 1)

    def scalar_mult(self,k, P):
        return Curve.mult(self,P,k)

    def ecc_encrypt(self,G, Pm, k, text, Pb):
        encrypted_text = []
        print_encrypted_text = []
```

```python
        Pml_text = []
        for char in text:
            Pml = self.scalar_mult(ord(char), Pm)
            Pml_text.append((Pml.x,Pml.y))
            kPb = self.scalar_mult(k, Pb)
            Pml_kPb = []
            Pml_kPb = self.add(Pml, kPb)
            kG = self.scalar_mult(k, G)
            encrypted_text.append([Point(kG.x,kG.y), Point(Pml_kPb.x,Pml_kPb.y)])
            print_encrypted_text.append([(kG.x,kG.y), (Pml_kPb.x,Pml_kPb.y)])

    def ecc_decrypt(self, encrypted_text, nb):
        decrypted_text = []
        for enc_msg in encrypted_text:
            kG = enc_msg[0]
            Pml_kPb = enc_msg[1]
            nbkG = self.scalar_mult(nb, kG)
            Pml = self.add(Pml_kPb, self.invert(nbkG))
            decrypted_text.append((Pml.x,Pml.y))
```

D. *Check Stop Word*

This method is used to check whether the words is having stop words or not

```python
def checkStopword(sentence, stop_words):
    words = sentence.split(' ')
    result = [w for w in words if not w in stop_words]
    result = [w for w in result if len(w) > 1]
    sentence = ""
    for word in result:
        sentence = sentence + word + " "
    sentence = sentence.strip()
    return sentence
```

E. *Remove Stop Word*

This method is used to remove the stop word from the list of words

```python
def stopwordRemove(textList):
    #stop_words = set(stopwords.words('english'))
    stop_words = ["a", "all", "an", "and", "any", "are", "as", "be", "been", "but", "by", "few", "for", "
    have", "he", "her", "here", "him", "his", "how", "i", "in", "is", "it", "its", "many", "me", "my", "
    none", "of", "on", "or", "our", "she", "some", "the", "their", "them", "there", "they", "that", "this",
     "us", "was", "what", "when", "where", "which", "who", "why", "will", "with", "you", "your"]
    text = []
    for i in range(len(textList)):
        text.append(checkStopword(textList[i], stop_words))
    return text
```

F. *Biword Creation*

This method is used to create the biwords i.e the substring of length 2 which is occuring the most number of times as asked for the GFGS Algorithm

```python
def biword_creation(textList):
    text = []
    for i in range(len(textList)):
        max = 0
        words = textList[i].split()
        freq = dict()
        res = ""
        for j in range(len(words)):
            words[j].lower()
            for k in range(len(words[j] ) - 1):
                word = words[j][k: k + 2]
                freq[word] = freq.get(word, 0) + 1
                if(max < freq[word]):
                    max = freq[word]
                    res = word
        if max == 1 and len(words) > 1:
            res = ""
        text.append(res)
    return text
```

G. *Sign msg Creation*

This method is used to sign the message using key (ECCDSA)

```python
# Create a digital signature for the string message using a given curve with a distinguished
# point P which generates a prime order subgroup of size n.
def sign(message, curve, P, n, keypair):
    # Extract the private and public keys, and compute z by hashing the message.
    d, Q = keypair
    z = hash_and_truncate(message, n)
    # Choose a randomly selected secret point kP then compute r and s.
    r, s = 0, 0
    while r == 0 or s == 0:
        sysrand = SystemRandom()
        k = sysrand.randrange(1, n)
        R = curve.mult(P, k)
        r = R.x % n
        s = (mult_inv(k, n) * (z + r*d)) % n
    print('ECDSA sig: (Q, r, s) = (' + str(Q) + ', ' + str(r) + ', ' + str(s) + ')')
    return (Q, r, s)
```

H. *Verifying msg Creation*

This method is used to verify the sign message wether it's is true or not

```python
# Verify the string message is authentic, given an ECDSA signature generated using a curve with
# a distinguished point P that generates a prime order subgroup of size n.
def verify(message, curve, P, n, sig):
    Q, r, s = sig
    # Confirm that Q is on the curve.
    if Q.is_infinite() or not curve.contains(Q):
        return False
    # Confirm that Q has order that divides n.
    if not curve.mult(Q,n).is_infinite():
        return False
    # Confirm that r and s are at least in the acceptable range.
    if r > n or s > n:
        return False
    # Compute z in the same manner used in the signing procedure,
    # and verify the message is authentic.
    z = hash_and_truncate(message, n)
    w = mult_inv(s, n) % n
    u_1, u_2 = z * w % n, r * w % n
    C_1, C_2 = curve.mult(P, u_1), curve.mult(Q, u_2)
    C = curve.add(C_1, C_2)
    return r % n == C.x % n
```

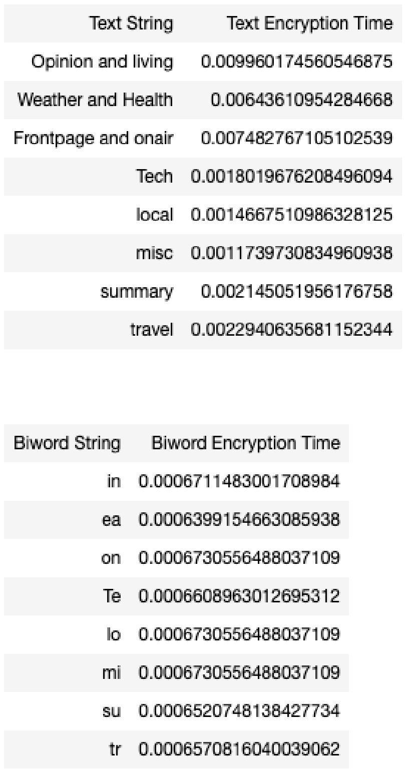
I. *Function for calulation Encryption Time*

This method is used for calculating the time taken by encryption

```python
def encrypt():
    encryp_collection = []
    timetaken_enc = []
    for word in text:
        start_time = time.time()
        plain_text = word
        encrypted_text = C.ecc_encrypt(G, Pm, k, plain_text, UA[2])
        encryp_collection.append(encrypted_text)
        end_time = time.time()
        timetaken_enc.append(end_time - start_time)
```

VII. Experiment Results

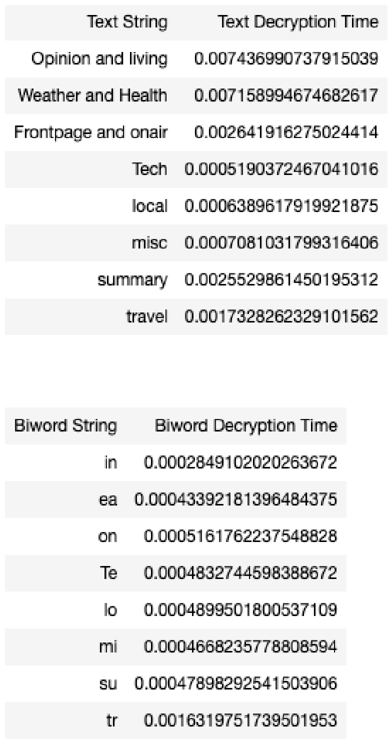
Showing the table of the experimental result Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.



Text String	Text Encryption Time
Opinion and living	0.009960174560546875
Weather and Health	0.00643610954284668
Frontpage and onair	0.007482767105102539
Tech	0.0018019676208496094
local	0.0014667510986328125
misc	0.0011739730834960938
summary	0.002145051956176758
travel	0.0022940635681152344

Biword String	Biword Encryption Time
in	0.0006711483001708984
ea	0.0006399154663085938
on	0.0006730556488037109
Te	0.0006608963012695312
lo	0.0006730556488037109
mi	0.0006730556488037109
su	0.0006520748138427734
tr	0.0006570816040039062

Fig. 1. Encryption Time



Text String	Text Decryption Time
Opinion and living	0.007436990737915039
Weather and Health	0.007158994674682617
Frontpage and onair	0.002641916275024414
Tech	0.0005190372467041016
local	0.0006389617919921875
misc	0.0007081031799316406
summary	0.0025529861450195312
travel	0.0017328262329101562

Biword String	Biword Decryption Time
in	0.0002849102020263672
ea	0.00043392181396484375
on	0.0005161762237548828
Te	0.0004832744598388672
lo	0.0004899501800537109
mi	0.0004668235778808594
su	0.00047898292541503906
tr	0.0016319751739501953

Fig. 2. Decryption Time

The result shows that if we are reducing the length of the message by taking in the most frequent common gram, then total time taken is drastically reduced.

The encryption is also done on a group of elements in the single biword using GFGS.

- The encryption efficiency of this algorithm is about 15% when compared to traditional ECC code.
- The decryption efficiency is 220.14% more as compared to classical ECC
- So, the proposed algorithm provides 87.75% much more efficiency in total

VIII. Conclusion

We are making the conclusion based on the results obtained.

- The proposed algorithm will take a different amount of time on different types of input plain-text received.
- It is better in terms of performance as compared to the existing ECC solution for large sets of data.
- In the worst case, it takes in the whole plain-text as input, if it does not get the most frequent word in the plain text.
- As many elements will be added to the group the common m-gram generated for all the elements of the group will reduce the encryption and decryption time of the ECC to a great extent.

References

[1] Implementation of Group Security using Elliptic Curve Cryptography
— https://drive.google.com/file/d/1mDYcgoQ8Esc7Ujn760j_IWJkaFnbqNtx/view?usp=sharing

[2] Link for the jupyter notebook of the implementation -
https://colab.research.google.com/gist/Yash1256/e00866f977c5c77f5bbf5f981fe6d304/quick-demo.ipynb

[3] Link for the template used for this documentation -
https://www.overleaf.com/latex/templates/one-column-ieee-journal-article/ykctxqgxptrs