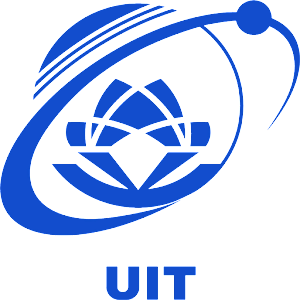
University of Information Technology



Faculty of Computer Network and Communications

FINAL REPORT

Subject: Cryptography Class: NT219.O21.ANTT

Lecturer: Nguyen Ngoc Tu

**Cryptanalysis on ECC-based Algorithms**

Nguyen Hai Phong – 22521088

Ho Trung Kien – 22520704

Ho Vinh Nhat – 22521013

Table of Contents

# Contents

[1. Contents 2](#_Toc168750708)

[2. Overview 3](#_Toc168750709)

[2.1 Elliptic curve: 4](#_Toc168750710)

[2.2 Elliptic Curves over Finite Fields 9](#_Toc168750711)

[2.3 The Elliptic Curve Discrete Logarithm Problem (ECDLP) 11](#_Toc168750712)

[2.3.1 The Double-and-Add Algorithm 12](#_Toc168750713)

[2.3.2 How hard is the ECDLP? 13](#_Toc168750714)

[3. Key generation in ECC 14](#_Toc168750715)

[4. Some attack models in ECC 16](#_Toc168750716)

[4.1 Baby-step giant-step 16](#_Toc168750717)

[4.2 Polard’s rho attack 17](#_Toc168750718)

[4.3 Pohlig – Hellman attack 19](#_Toc168750719)

[4.4 Smart attack where #E(Fp) = p 19](#_Toc168750720)

[4.5 Invalid curve attack 20](#_Toc168750721)

[5. Implementation and testing 21](#_Toc168750722)

[5.1 Baby-step giant-step (BSGS) 21](#_Toc168750723)

[5.2 Polard-rho attack 23](#_Toc168750724)

[5.3 Pohlig-Hellman attack 27](#_Toc168750725)

[5.4 Smart attack 28](#_Toc168750726)

[5.5 Invalid curve attack 30](#_Toc168750727)

[6. Deployment: 39](#_Toc168750728)

[7. References 40](#_Toc168750729)

# Overview

* + Elliptic Curve Cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. One of the advantages of ECC is that ECC allows smaller keys to provide equivalent security, compared to cryptosystems based on modular exponentiations in Galois fields, such as RSA, ELGamal, which is particularly beneficial in systems with limited computational resources. ECC is used in a many of fields and applications:
    - Internet of Things (IoT): ECC’s efficiency makes it ideal for IoT devices, which often have limited resources. It provides strong security without overburdening the device’s resources, also with lesser key-size, ECC is faster when compared to other cryptosystems like RSA
    - Smart grid network: ECC’s lightweight authentication, less key-size and equivalent security compared to other asymmetric cryptosystems make it an excellent choice for this network.
    - Key agreement applications: One of ECC applications is key agreement between server and clients. Using Diffie–Hellman key exchange with ECC, ECDHE (Elliptic-Curve Diffie-Hellman Ephemeral) is widely used in popular shopping platforms such as Amazon, Shopee, … for establish session keys.
    - Blockchain applications: Another ECC application is digital signature, which is used in creating signature of transactions in blockchain for security. Using DSA (Digital Signature Algorithm) with ECC. ECDSA (Elliptic-Curve Digital Signature Algorithm) can be trusted to sign a transaction.
    - Secure Web Browsing: ECC is one of algorithms which is used in security protocols like Transport Layer Security (TLS) and Secure Sockets Layer (SSL) to secure web traffic.
    - Cloud Computing: Cloud service providers use ECC to ensure the privacy and security of their users’ data.
    - Mobile Applications: Many mobile apps use ECC to secure communications, protecting user data and privacy.
  + In short, ECC have the same benefits of the other cryptosystems such as confidentiality, integrity, authentication and non-reputation but what make it widely used in many areas of technology is it has shorter key lengths, which helps the encryption, decryption and signature verification process speed up. Also it will save lots of storage when storing keys and decrease the amount of bandwidth when exchanging key using ECDH. In this project, we will validate the strength and security of their ECC implementation to prevent potential breaches.

## Elliptic curve:

* An elliptic curve is the set of solutions to an equation of the form:

𝑌2 = 𝑋3 + 𝐴𝑋 + 𝐵

* Equations of this type are called Weierstrass equations after the mathematician who studied them extensively during the nineteenth century. Two examples of elliptic curves (are illustrated in Figure 1).

𝐸1: 𝑌2 = 𝑋3 − 𝑋 𝑎𝑛𝑑 𝐸2: 𝑌2 = 𝑋3 − 𝑋 + 1

A red circle and a black background

Description automatically generated with medium confidence

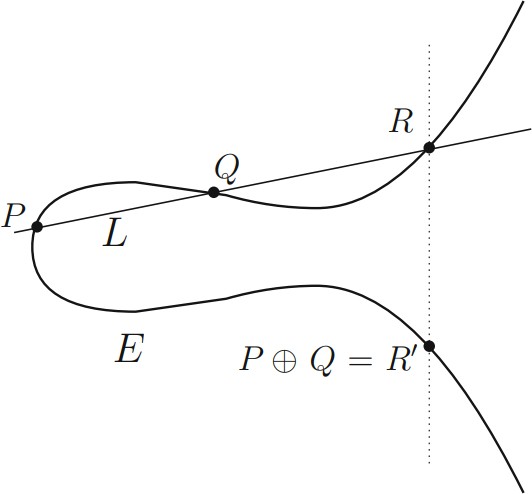
*Figure 1*

* There are some other forms of Elliptic curve, such as:
* Montgomery form: 𝐵𝑦2 = 𝑥3 + 𝐴𝑥2 + 𝑥
* Edwards form: 𝑥2 + 𝑦2 = 1 + 𝑑𝑥2𝑦2
* Twisted Edwards form: 𝑎𝑥2 + 𝑦2 = 1 + 𝑑𝑥2𝑦2
* Doubling-oriented Doche–Icart–Kohel form: 𝑦2 = 𝑥3 + 𝑎𝑥2 + 16ax
* Tripling-oriented Doche–Icart–Kohel form: 𝑦2 = 𝑥3 + 3a(𝑥+1)2

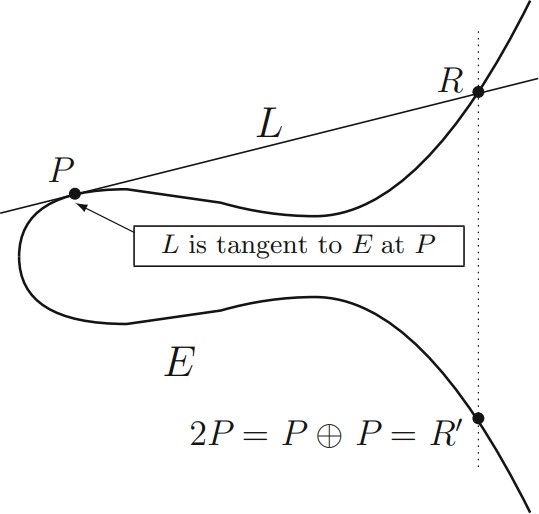
**\*\*\* Point addition**

* One of features of elliptic curves is that there is a natural way to take two points on an elliptic curve and “add” them to produce a third point. To visualize the addition of two points on the curve, a geometric representation is presented here.
* Let P and Q be two points on an elliptic curve E, as illustrated in Figure 2. We draw the line L through P and Q. This line intersects at three points: P, Q and one other point R. We take that point R and reflect it across the x-axis to get a new point R’. The point R’ is called the “sum of P and Q,” although as you can see, this process is nothing like ordinary addition. For now, we denote this strange addition law by the symbol ⊕. Thus, we write:

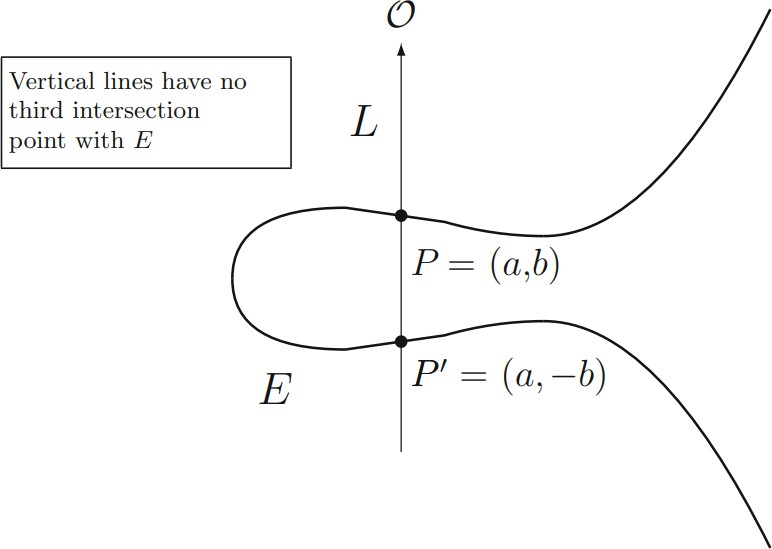
P ⊕ Q = R



* *Figure 2: The addition law on elliptic curve*
* There are a few subtleties to elliptic curve addition that need to be addressed. First, what happens if we want to add a point P to itself? Imagine what happens to the line L connecting P and Q if the point Q slides along the curve and gets closer and closer to P. In the limit, as Q approaches P, the line L becomes the tangent line to E at P. Thus in order to add P to itself, we simply take L to be the tangent line to E at P, as illustrated in Fig. 6.3. Then L intersects E at P and at one other point R, so we can proceed as before. In this case, L still intersects E at three point, but P is counted two times.



*Figure 3: Adding a point P to itself*

* A second potential problem with our “addition law” arises if we try to add a point P = (a, b) to its reflection about the X-axis P’ = (a, −b). The line L through P and P’ is the vertical line x = a, and this line intersects E in only the two points P and P’. (See Figure 4). There is no third point of intersection, so it appears that we are stuck! But there is a way out. The solution is to create an extra point O that lives “at infinity.” More precisely, the point O does not exist in the XY -plane, but we pretend that it lies on every vertical line. We then set:

𝑃 + 𝑃′ = 𝑂

*Figure 4: The vertical line L through P = (a, b) and P = (a, −b)*

* To understand more about point addition in elliptic curve, we recommend following <https://curves.xargs.org/> to have an intuitive view.
* We also need to figure out how to add O to an ordinary point P = (a, b) on E. The line L connecting P to O is the vertical line through P, since O lies on vertical lines, and that vertical line intersects E at the points P, O, and P’ = (a, −b). To add P to O, we reflect P’ across the X-axis, which gets us back to P. In other words, P ⊕ O = P, so O acts like zero for elliptic curve addition.

1

***R***

2

3

4

***Q P***

***P***

***P***

***P***

***Q***

***Q***

***P + Q + R = 0***

***P + Q + Q = 0***

***P + Q + 0 = 0***

***P + P + 0 = 0***

After all, we have a final definition for elliptic curve:

***Definition:*** An elliptic curve E is the set of solutions to a Weierstrass equation

𝑌2 = 𝑋3 + 𝐴𝑋 + 𝐵

together with an extra point O, where the constants A and B must satisfy:

4𝐴2 + 27𝐵3 ≠ 0 (\*)

*Remark (\*): What is this extra condition* 4𝐴2 + 27𝐵3 ≠ 0*? The quantity ΔE =* 4𝐴2 + 27𝐵3 *is called the discriminant of E. The condition ΔE* ≠ *0 is equivalent to the condition that the polynomial* 𝑋3 + 𝐴𝑋 + 𝐵 *have no repeatedly roots.*

***Theorem 1:*** *Let E be an elliptic curve. Then the addition law on E has the following properties:*

1. 𝑃 + 𝑂 = 𝑂 + 𝑃 = 𝑃 for all P ∈ E [Identity]
2. 𝑃 + (−𝑃) = 𝑂 for all P ∈ E [Inverse]
3. 𝑃 + (𝑄 + 𝑅) = (𝑃 + 𝑄) + 𝑅 for all P, Q, R ∈ E [Associative]
4. 𝑃 + 𝑄 = 𝑄 + 𝑃 for all P, Q ∈ E [Communicative]

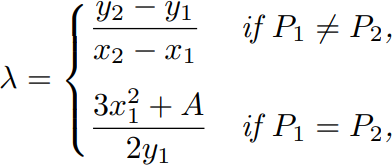
Our next task is to find explicit formulas to enable us to easily add and subtract points on an elliptic curve. The derivation of these formulas uses elementary analytic geometry, a little bit of differential calculus to find a tangent line, and a certain amount of algebraic manipulation. We state the results in the form of an algorithm.

***Theorem 2:*** *(Elliptic Curve Addition Algorithm). Let:*

𝐸: 𝑌2 = 𝑋3 + 𝐴𝑋 + 𝐵

*be an elliptic curve and let P1 and P2 be points on E.*

1. *If P1 = O, then* 𝑃1 + 𝑃2 = 𝑃2
2. *Otherwise, if* 𝑃2 = 0*, then* 𝑃1 + 𝑃2 = 𝑃1
3. *Otherwise, write* 𝑃1 = (𝑥1, 𝑦1), 𝑃2 = (𝑥2, 𝑦2)
4. *If* 𝑥1 = 𝑥2 𝑎𝑛𝑑 𝑦1 = −𝑦2*, then* 𝑃1 + 𝑃2 = 𝑂
5. *Otherwise, define λ by*



*and let*

𝑥3 = 𝜆2 − 𝑥1 − 𝑥2 𝑎𝑛𝑑 𝑦3 = 𝜆(𝑥1 − 𝑥3) − 𝑦1

*Then* 𝑃1 + 𝑃2 = (𝑥3, 𝑦3)*.*

## Elliptic Curves over Finite Fields

In the previous section we developed the theory of elliptic curves geometrically. For example, the sum of two distinct points P and Q on an elliptic curve E is defined by drawing the line L connecting P to Q and then finding the third point where L and E intersect, as illustrated in Fig. 6.2. However, in order to apply the theory of elliptic curves to cryptography, we need to look at elliptic curves whose points have coordinates in a finite field Fp. This is easy to do.

***Definition:*** *Let* 𝑝 ≥ 3 *is a prime. An elliptic curve over* 𝐹𝑝 *is an equation of the form*

𝐸: 𝑌2 = 𝑋3 + 𝐴𝑋 + 𝐵 *with A,B ∈* 𝐹𝑃 *satisfying* 4𝐴3 + 27𝐵2 ≠ 0

The set of points on E with coordinates in 𝐹𝑃 is the set

𝐸(𝐹𝑃) = {(𝑥, 𝑦) ∶ 𝑥, 𝑦 ∈ 𝐹𝑃 𝑠𝑎𝑡𝑖𝑠𝑓𝑦 𝑦2 = 𝑥3 + 𝐴𝑥 + 𝐵} ∪ {𝑂}

*Example: Consider the elliptic curve*

𝐸: 𝑌2 = 𝑋3 + 3𝑋 + 8 *over the field* 𝐹13

*We can find the points of E(*𝐹13*) by substituting in all possible values X = 0, 1, 2,..., 12 and checking for which X values the quantity* 𝑋3 + 3𝑋 + 8 *is a square modulo 13. For example, putting X = 0 gives 8, and 8 is not a square modulo 13. Next we try X = 1, which gives 1+3+8 = 12. It turns out that 12 is a square modulo 13; in fact, it has two square roots,*

*52 ≡ 12 (mod 13) and 82 ≡ 12 (mod 13).*

*This gives two points (1, 5) and (1, 8) in E(*𝐹13*). Continuing in this fashion, we end up with a complete list,*

𝐸(𝐹13) = {𝑂, (1, 5), (1, 8), (2, 3), (2, 10), (9, 6), (9, 7), (12, 2), (12, 11)}.

*Thus,* 𝐸(𝐹13) *contains nine points.*

Suppose now that P and Q are two points in E(𝐹𝑃) and that we want to “add” the points P and Q. One possibility is to develop a theory of geometry using the field Fp instead of R.

Let 𝑃 = (𝑥1, 𝑦1) and 𝑄 = (𝑥2, 𝑦2) be points in E(𝐹𝑃). We define the sum P + Q to be the point (𝑥3, 𝑦3) obtained by applying the elliptic curve addition algorithm (Theorem

2). Notice that in this algorithm, the only operations used are addition, subtraction, multiplication, and division involving the coefficients of E and the coordinates of P and Q. Since those coefficients and coordinates are in the field Fp, we end up with a point (𝑥3, 𝑦3) whose coordinates are in 𝐹𝑃.

***Theorem 3:*** *Let E be an elliptic curve over* 𝐹𝑃 *and let P,Q be points in* 𝐸(𝐹𝑃)*.*

1. *The elliptic curve addition algorithm (Theorem 2) applied to P and Q yields a point in E(Fp). We denote this point by P + Q.*
2. *This addition law on* 𝐸(𝐹𝑃) *satisfies all of the properties listed in Theorem 1. In other words, this addition law makes* 𝐸(𝐹𝑃) *into a finite group.*

*Example: We continue with the elliptic curve*

𝐸: 𝑌2 = 𝑋3 + 3𝑋 + 8 *over the field* 𝐹13

*and we use the addition algorithm (Theorem 2) to add the point P=(9,7) and Q=(1,8) in* 𝐸(𝐹13)*. Step (e) of that algorithm tells us to first compute:*

𝑦2 − 𝑦1 8 − 7 1

𝜆 = = =

1

=  = 8 (𝑚𝑜𝑑 13)

𝑥2 − 𝑥1

1 − 9

−8 5

*where recall that all computations are being performed in the field* 𝐹13*, so -8 = 5 ,*

1 = 5−1 = 8*. Next we compute*

5

𝑣 = 𝑦1 − 𝜆𝑥1 = −65 = 0

*Finally, the addition algorithm tells us to compute*

𝑥3 = 𝜆2 − 𝑥1 − 𝑥2 = 64 − 9 − 1 = 54 = 2*,*

𝑦3 = −(𝜆𝑥3 + 𝑣) = −8 ∗ 2 = −16 = 10*.*

*This completes the computation of*

𝑃 + 𝑄 = (1,8) + (9,7) = (2,10) *in* 𝐸(𝐹13)*.*

## The Elliptic Curve Discrete Logarithm Problem (ECDLP)

In [Diffie-Hellman key exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange), we know about the discrete logarithm (DLP) in the finite field 𝐹𝑃. In order to create a cryptosystem based on the DLP for 𝐹𝑃, Alice publishes two numbers g and h, and her secret is the exponent x that solves the

congruence

ℎ ≡ 𝑔𝑥 (𝑚𝑜𝑑 𝑝)

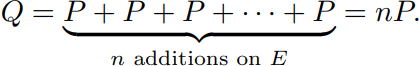
Let’s consider how Alice can do something similar with an elliptic curve E over 𝐹𝑃. If Alice views g and h as being elements of the group 𝐹𝑃, then the discrete logarithm problem requires Alice’s adversary Eve to find an x such that

ℎ ≡ 𝑔 ∗ 𝑔 ∗ 𝑔 … 𝑔 (𝑚𝑜𝑑 𝑝) (x multiplications)

In other words, Eve needs to determine how many times g must be multiplied by itself in order to get to h. With this formulation, it is clear that Alice can do the same thing with the group of points 𝐸(𝐹𝑃) of an elliptic curve E over a finite field Fp. She

chooses and publishes two points P and Q in 𝐸(𝐹𝑃), and her secret is an integer n that

makes



Then Eve needs to find out how many times P must be added to itself in order to get

Q. Keep in mind that although the “addition law” on an elliptic curve is conventionally written with a plus sign, addition on E is actually a very complicated operation, so this elliptic analogue of the discrete logarithm problem may be quite difficult to solve.

***Definition:*** *Let E be an elliptic curve over the finite field Fp and let P and Q be points in E(Fp). The Elliptic Curve Discrete Logarithm Problem (ECDLP) is the problem of finding an integer n such that Q = nP. By analogy with the discrete logarithm problem for* 𝐹𝑝*, we denote this integer n by*

𝑛 = log𝑃(𝑄)

*and we call n the elliptic discrete logarithm of Q with the respect to P.*

### The Double-and-Add Algorithm

In order for cryptography, we need to compute 𝑛 ∗ 𝑃 from known value n and P efficiently, if n is large, we certainly do not want to compute nP by computing linearly P, 2P, 3P, …

The double-and-add algorithm can solve this problem efficiently. First, we write n in binary form as

𝑛 = 𝑛0 + 𝑛1 ∗ 2 + 𝑛2 ∗ 4 + ⋯ + 𝑛𝑟 ∗ 2𝑟 with 𝑛0, 𝑛1, … , 𝑛𝑟 ∈ {0,1}

(We also assume that 𝑛𝑟 = 1). Next we compute the following quantities:

𝑄0 = 𝑃, 𝑄1 = 2𝑄0, … , 𝑄𝑟 = 2𝑄𝑟−1.

Notice that 𝑄𝑖 is simply twice the previous 𝑄𝑖−1, so:

𝑄𝑖 = 2𝑖𝑃

**Input: Point P ∈ E(Fp) and integer n ≥ 1**

1. Set Q = P and R = O.
2. Loop while n > 0.
3. If n ≡ 1 (mod 2), set R = R + Q.
4. Set Q = 2Q and 𝑛 = [𝑛/2]
5. If n > 0, continue with loop at Step 2.
6. Return the point R, which is equal nP.

*Table 1: The double-and-add algorithm for elliptic curves*

These points are referred to as 2-power multiples of P, and computing them requires r doublings. Finally, we compute nP using at most r additional additions,

𝑛𝑃 = 𝑛0𝑃0 + 𝑛1𝑄1 + ⋯ + 𝑛𝑟𝑄𝑟

We’ll refer to the addition of two points in E(Fp) as a point operation. Thus the total time to compute nP is at most 2r point operations in E(Fp). Notice that n ≥ 2r, so it takes no more than 2 log2 𝑛 point operations to compute nP. This makes it feasible to compute nP even for very large values of n. We have summarized the double-and-add algorithm in Table 1.

### How hard is the ECDLP?

The Elliptic Curve Discrete Logarithm Problem (ECDLP) is considered to be a challenging problem in cryptography. Unlike the finite field Discrete Logarithm Problem (DLP), there are no general-purpose subexponential algorithms to solve the ECDLP. The principal reason that elliptic curves are used in cryptography is the fact that there are no index calculus algorithms known for the ECDLP, and indeed, there are no general algorithms known that solve the ECDLP in fewer than 𝑂(√𝑝) steps. In

other words, despite the highly structured nature of the group 𝐸(𝐹𝑃), the fastest

known algorithms to solve the ECDLP are no better than the generic algorithm that works equally well to solve the discrete logarithm problem in any group. This fact is sufficiently important that it bears highlighting.

**The fastest known algorithm to solve ECDLP in** 𝑬(𝑭𝑷) **takes approximately** √𝒑 **steps**

# Key generation in ECC

* + Key generation in elliptic curve cryptography (ECC) is an important process that involves in creating pairs of cryptographic keys—a private key and a public key which is based on private key. The key generation in ECC cryptography is as simple as securely generating a random integer in certain range of chosen curve’s field size. Here is step-by-step of key generation in ECC:
    - Choosing parameters: The first step in key generation is to select the parameters for the elliptic curve. These parameters include:
* **p**: A prime number specifying the size of the finite field **Fp**
* **a and b**: Coefficients of the elliptic curve equation
* **G**: A base point (generator) on the elliptic curve
* **n**: The order of the base point **G**, which is the smallest positive number such that **nG = O** (the point at infinity)
* **h**: The cofactor
  + - These parameters must be agreed upon by all parties using ECC cryptosystem. The choice of parameters is critical for the security and efficiency of the ECC system, because most common attacks in ECC related to “cryptographic failure”. That means if you choose wrong parameters, that can make some weakness to your curve and attacker can use it to leak some restricted information in your system. We suggest using standard and safe curves which is defined by NIST from <https://safecurves.cr.yp.to/index.html>, which was tested and ensured to be safe.
    - Generating the private key: The private key in ECC is a random integer chosen from the interval **[1, n – 1]**, where **n** is the order of the base point **G**. The size of the private key depends on the desired level of security. The private key should be kept secret and not shared with anyone.
    - Computing the public key: The public key is computed based on the private key and the chosen curve. It is computed by performing scalar multiplication of the chosen base point G by the private key. Scalar multiplication process involves repeatedly adding a point to itself on the elliptic curve
  + Example code for key generation using Curve25519 using Sagemath:

from sage import \*

*#Setting curve parameters*

p = 0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed

K = GF(p)

A = K(0x76d06)

B = K(0x01)

E = EllipticCurve(K, ((3 - A^2)/(3 \* B^2), (2 \* A^3 - 9 \* A)/(27 \* B^3)))

def to\_weierstrass(*A*, *B*, *x*, *y*):

    return (*x*/*B* + *A*/(3\**B*), *y*/*B*)

def to\_montgomery(*A*, *B*, *u*, *v*):

    return (*B* \* (*u* - *A*/(3\**B*)), *B*\**v*)

G = E(\*to\_weierstrass(A, B, K(0x09), K(0x20ae19a1b8a086b4e01edd2c7748d14c923d4d7e6d7c61b229e9c5a27eced3d9)))

E.set\_order(0x1000000000000000000000000000000014def9dea2f79cd65812631a5cf5d3ed \* 0x08)

*#Generating key pair*

import random

private\_key = random.randint(1, E.order()-1)

public\_key = private\_key \* G

*#Output*

print(f"p = {p}")

print(f"a = {A}")

print(f"b = {B}")

print(f'G = {G}')

print(f"public key: {public\_key}")

print(f"private key: {private\_key}")

* + Sample output:

A screen shot of a video game

Description automatically generated

# Some attack models in ECC

## Baby-step giant-step

* + - Baby-step Giant-step is the algorithm used to calculate DLP and presents several standard variants of it. The giant step small step algorithm uses space- time trade-offs to solve the discrete logarithm problem in arbitrary groups.
    - Consider a prime p , an elliptic curve E over the finite field Fp a base point P (with order q). Also Q is another point on the curve. It is required to find k such that Q=kP. The following steps are taken to find out the value of k.
      * Let m = [√𝑝]

Using Euclid theorem, we write: k = im+ j for i, j ∈ {0,1,2,...,m−1}. Now, Q = kP = (im+ j)P = [im]P+[ j]P,

Therefore,

[ j]P = Q−[im]P

Here, the values of i and j are not known. If the values of i and j can be figured out, then k can be found easily since k = im+ j.

* + - * We compute [ j]P for all j = 0,1,2,...,m−1 and store the results in table.
      * We start computing

Q−[im]P for each j = 0,1,...,m−1

For each computation of Giant step: Q − [im]P, it is checked if the obtained value exists in the previous table. If the obtained result exists in table already, it can be said that there is a matching we stop computing giant steps.

* + - * If a match is found in previous step, the values of i and j satisfies [j]P = Q−[im]P as well. We compute k = im+ j. Hence, k has been found.

## Polard’s rho attack

* + - In 1978, Pollard came up with a “Monte-Carlo” method for solving the discrete logarithm problem. Since then the method has been modified to solve the elliptic curve analog of the discrete logarithm problem. As the Pollard-Rho algorithm is currently the quickest algorithm to solve the Elliptic Curve Discrete Logarithm, so the security of the elliptic curve cryptosystem depends on the efficiency of this algorithm. Theoretically, if the Pollard-Rho algorithm is able to solve the ECDLP efficiently and in a relatively short time, then the system will be rendered insecure.
    - Pollard’s rho is another algorithm for computing discrete logarithms. It has the

same asymptotic time complexity 𝑂(√𝑛) of the BSGS algorithm, but its space complexity is just 𝑂(1). If baby-step giant-step can’t solve discrete logarithms because of the huge memory requirements, will Pollard’s rho make it? Let’s see…

* + - Let 𝐺 = 𝐸(𝐹𝑃), such that |𝐺| = 𝑛, and P and Q such that 𝑄 = 𝑥 ∗ 𝑃 in G, our aim is to caculate x. With Pollard’s rho, we will solve a sightly different problem: given P and Q, find the integers a, b, A and B such that

𝑎𝑃 + 𝑏𝑄 = 𝐴𝑃 + 𝐵𝑄. Once four integers are found, we can compute x:

𝑎 − 𝐴

𝑥 =

#### Algorithm: Polard’s Rho algorithm

𝐵 − 𝑏

(𝑚𝑜𝑑 𝑛)

1. *Using hash function, we partition G into 3 sets,* 𝑆1, 𝑆2, 𝑆3 *of roughly the same size, but* 𝑂 ∉ 𝑆2
2. *Define an iterating function f of a random walk:*

𝑄 + 𝑅𝑖, 𝑅𝑖 ∈ 𝑆1

𝑅𝑖+1 = 𝑓(𝑅𝑖) = {

1. *Let* 𝑅𝑖 = 𝑎𝑖𝑃 + 𝑏𝑖𝑄*, and therefore*

2𝑅𝑖, 𝑅𝑖 ∈ 𝑆2

𝑃 + 𝑅𝑖, 𝑅𝑖 ∈ 𝑆3

*(1)*

*and*

𝑎𝑖+1 = {

𝑎𝑖, 𝑅𝑖 ∈ 𝑆1 2𝑎𝑖, 𝑅𝑖 ∈ 𝑆2

𝑎𝑖 + 1, 𝑅𝑖 ∈ 𝑆3

𝑏𝑖 + 1, 𝑅𝑖 ∈ 𝑆1

*(2)*

𝑏𝑖+1 = {

2𝑏𝑖, 𝑅𝑖 ∈ 𝑆2

𝑏𝑖, 𝑅𝑖 ∈ 𝑆3

*(3)*

1. *Start with* 𝑅0 = 𝑃, 𝑎0 = 1, 𝑏0 = 0 *and generate pairs* (𝑅𝑖, 𝑅2𝑖) *until a match is found, in example,*

𝑅𝑚 = 𝑅2𝑚 *for some m Once we found a match, we have*

𝑅𝑚 = 𝑎𝑚𝑃 + 𝑏𝑚𝑄

𝑅2𝑚 = 𝑎2𝑚𝑃 + 𝑏2𝑚𝑄

*Hence we compute x to be:*

𝑥 = 𝑎2𝑚−𝑎𝑚

𝑏𝑚−𝑏2𝑚

(𝑚𝑜𝑑 𝑛) *(4)*

* Assuming that the random walk (4.1) defined in the algorithm produces random terms, the algorithm solves the elliptic curve discrete logarithm problem in

𝑂(√𝑛) operations.

* To be able to calculate x, the denominator in (4) has to be invertible in ℤ𝑛, where n is the group order of 𝐸(𝐹𝑃). If the gcd(𝑏𝑚 − 𝑏2𝑚, 𝑛) > 1, the inverse

of (𝑏𝑚 − 𝑏2𝑚) doesn’t exist. So the Pollard - Rho algorithm does not work in some ECDLP instances.

* In commercial implementations, the curve E, the underlying finite field, 𝐹𝑝 and the point P are choosen such that #𝐸(𝐹𝑝) = 𝑛 is a prime. That means that there is a high probability of success in solving the ECDLP using the Pollard-Rho Algorithm. While choosing n to be prime increases the success of this attack, recall that the Pohlig-Hellman attack works by factoring n. Curves that are susceptible to the Pohlig-Hellman attack are deemed insecure and are unacceptable for use in commercial implementations. Therefore, for a cryptosystem to be protected against the Pohlig-Hellman attack, n should be prime.

## Pohlig – Hellman attack

* + - The Pohlig-Hellman algorithm was presented by Stephan C. Pohlig and Martin

E. Hellman in 1978. This method is a special purpose algorithm uses for solving the DLP for Finite Group whose order can be factored into prime powers of smaller primes. The algorithm reduces the computation of the discrete log in the finite group to the computation of the discrete log in subgroups whose order is a small prime, then use CRT to combine these to a logarithm in the full group.

Let P and Q be points on an elliptic curve. Suppose that we want to solve an integer k such that Q = [k]P. In this attack we know the order N of P and we first compute the prime factorization of N satisfied by:

𝑞

𝑘

𝑁 = ∏

𝑖=1

𝑒𝑖

𝑖

* + - Then starting with q1, we consider 𝑞𝑒1|N and we compute

1

𝑄 = [ 𝑁 ] 𝑄 𝑎𝑛𝑑 𝑃 = [ 𝑁 ] 𝑃

0 𝑞𝑒1 0 𝑞𝑒1

1

1

By using BSGS to solve DLP, we can find an integer k0 such that Q0 = k0 P0 mod(𝑞𝑒1)

1

* + - Now we can find k (mod 𝑞𝑒𝑖) for each i then use the [Chinese Remainder](https://en.wikipedia.org/wiki/Chinese_remainder_theorem) [theorem](https://en.wikipedia.org/wiki/Chinese_remainder_theorem) (CRT) to combine them and then obtain k (mod N).

𝑖

## Smart attack where #E(Fp) = p

* + - As the above section, if the curve’s order is smooth (be a product of small prime factors), the curve will be attacked by Pohlig – Hellman algorithm. So people think a solution to fix that. If #𝐸(𝐹𝑃) = 𝑝 (a prime number), it will be safe with

Pohlig – Hellman, but will be attacked by another algorithm, its name is Smart

attack.

* + - Smart attack describes a linear time method of computing the ECDLP in curves over a field 𝐹𝑃 such that #𝐸(𝐹𝑃) = 𝑝, or in other words

𝑡𝑃 = 𝑝 + 1 − #𝐸(𝐹𝑃) = 1 (the quantity 𝑡𝑃 is called the *trace of Frobenius,* it

relates to some other sides of elliptic curve). Elliptic curves that satisfy that

condition are also called *anomalous curves.*

* + - If a curve E defined over finite field of size p, has a subproup with order of p, then ECDLP problem can be solved in O(1) time.
    - Now, given arbitary curve E over a finite field size p (𝐹𝑃) with #𝐸(𝐹𝑃) = 𝑝, and point 𝑄 = 𝑑 ∗ 𝑃, find d?
      * First step, we try to lift these points to 𝐸(𝑄𝑃) (elliptic curve on p-adic field) using [Hessel’s lift](https://en.wikipedia.org/wiki/Hensel%27s_lemma) to get two new point P’ and Q’. We do this by setting the x component of 𝑃’ equal to the x component of P. We then use Hensel's Lemma to compute y in 𝑄𝑝. We know that 𝑄 = 𝑘𝑃 in 𝐸(𝐹𝑃) so thus in the kernel of that homomorphism.

𝑄′ − 𝑘𝑃′ ∈ 𝐸1(𝑄𝑃)

* + - * Now we rely on the fact that the order of 𝐸(𝐹𝑃) is p, which ensures that multiplying any element in 𝐸(𝑄𝑃) by p maps the elements into 𝐸1(𝑄𝑃) since for any point 𝑅 ∈ 𝐸(𝑄𝑃) the point 𝑝𝑅 will map via Reduction Modulo P to 𝒪 in 𝐸(𝐹𝑝 ). So multiply through by 𝑝 and we get

𝑝𝑄′ − 𝑘(𝑝𝑃′) ∈ 𝐸2(𝑄𝑝)

with 𝑝𝑄′ ∈ 𝐸1(𝑄𝑝) and 𝑝𝑃′ ∈ 𝐸(𝑄𝑃). We can now apply the p-adic elliptic log to get

and thus

ψp(𝑝𝑄′) − kψp(𝑝𝑃′) ∈ pℤp

𝜓𝑝(𝑝𝑄′)

𝑘 =

𝜓

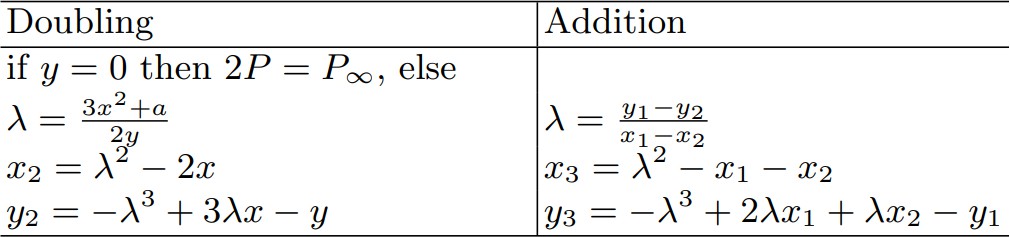
𝑝

(𝑝𝑃′)

and then reduce k modulo p to return 𝐹𝑝 solving ECDLP.

## Invalid curve attack

* + - Invalid Curve Attack relies on the fact that given the Weierstrass equation y2 = x3+ax+b of an elliptic curve over a prime field E(Fp) with base point G, the doubling and addition formulas do not depend on the coefficient b. The following table illustrates this property by giving the formulas for affine coordinates (but it is the case for all representation system):



* + - Thus, if a point is not checked to be on the curve, the attacker can send a point

which lie on the curve E’(Fp) of equation 𝑦2 = 𝑥3 + 𝑎𝑥 + 𝑏1, and now the server will calculate point additions, multiplications on that curve, not the original curve.

* + - This kind of attack doesn’t depend on the weakness of the curve. Any curve can be attacked by an invalid curve attack if the server does not check whether the point is on the curve or not.
    - Using the above property, we can say that different points can be chosen from different curves having the same *a* but different values of *b*:
      * y2 = x3+ax+b mod p
      * y2 = x3+ax+b1 mod p
      * y2 = x3+ax+b2 mod p
      * y2 = x3+ax+b3 mod p
    - Now we have multiple curves from which we can choose our points and share as public keys, and this will not affect the results of Elliptic Curve Arithmetic. We can selectively choose points having small order of the subgroup, generated by scalar multiplication. The remaining steps we can use Pohlig-Helman to solve DLP*(xP=Q)* then use CRT to find x.

# Implementation and testing

In this section, we use python 3.x, sagemath, pycryptodome and some cryptographic libraries to implement and test some algorithms we presented above.

## Baby-step giant-step (BSGS)

* + - We get the following curve

- P = (523283288, 42887324)

- b = 687298370

- a = 196220977

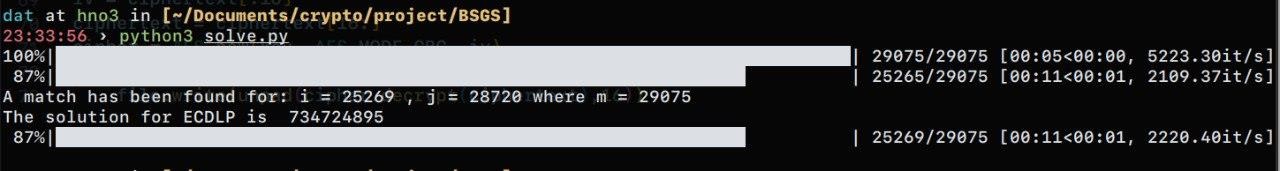
- p = 845337859

- Q = (394977470, 739173023)

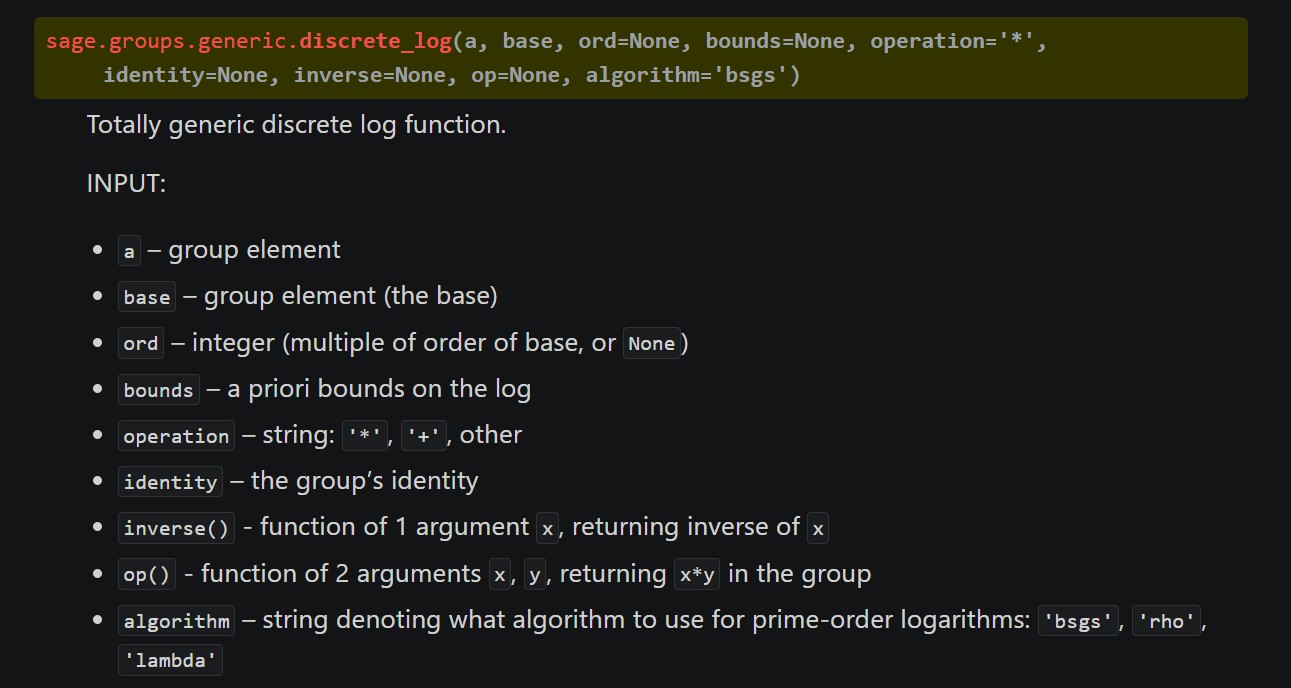
* + - Now we use BSGS to solve Q=dP
    - **def** binary\_search(array, value):

|  |
| --- |
| - n = len(array) |
| - left = 0 |
| - right = n-1 |
| - **while** left <= right: |
| - mid = floor((left + right) / 2) |
| - **if** array[mid] < value: |
| - left = mid + 1 |
| - **elif** array[mid] > value: |
| - right = mid - 1 |
| - **else**: |
| - **return** mid |
| - **return** None |
| - |
| - **def** BSGS\_ECDLP(P, Q, E): |
| - **if** P == Q: |
| - **return** 1 |
| - m = ceil(sqrt(P.order())) |
| - baby\_list = [] |
| - sorted\_list = [] |
| - **for** j **in** trange(m): |
| - PP = j\*P |
| - baby\_list.append(PP) |
| - sorted\_list.append(PP) |
| - sorted\_list.sort() |
| - |
| - **for** i **in** trange(m): |
| - result = Q - (i\*m)\*P |
| - pos = binary\_search(sorted\_list, result) |
| - |
| - **if** pos != None: |
| - idx = baby\_list.index(result) |
| - **print**("A match has been found for: i =",i, ",", |
| - "j =", idx, "where m =", m) |
| - x = (i\*m + idx) % P.order() |
| - **print**("The solution for ECDLP is ", x) |
| - **return** x |
| - **return** False |

* + - Here is output:



* + - Sagemath also has a build-in function discrete\_log that can solve ECDLP using BSGS.



* + - Let’s use it to solve the above problem:

d = discrete\_log(Q,P, operation="+")

* + - Output:



* + - Both get the same result.

## Polard-rho attack

Suppose that we have a pdf file encrypted by a remote server, the server code is below.

Server.py

1. **from** sage.all **import** \*

3. **import** random

5. **from** Crypto.Cipher **import** AES

7. **from** hashlib **import** sha3\_512 # most secure hash I've heard :v

9. **def** check(prime):

11.

**print**("Not a prime!!!")

13.

**if** prime <= (1>>40):

**return** False

12.

**if not** isPrime(prime):

10.

8.

6. **from** Crypto.Util.Padding **import** pad

4. **import** secrets

2. **from** Crypto.Util.number **import** \*

|  |
| --- |
| 14. **print**("Your prime is so weak!!!") |
| 15. **return** False |
| 16. **return** True |
| 17. |
| 18. **def** encrypt(key, mess): |
| 19. key = sha3\_512(str(key).encode()).digest()[:16] |
| 20. iv = secrets.token\_bytes(16) |
| 21. cipher = AES.new(key, AES.MODE\_CBC, iv) |
| 22. ct = cipher.encrypt(pad(mess, AES.block\_size)) |
| 23. **return** iv + ct |
| 24. |
| 25. **def** genPara(p): |
| 26. **while** True: |
| 27. a,b = random.randrange(0,p-1), random.randrange(0,p-1) |
| 28. E = EllipticCurve(GF(p), [a,b]) |
| 29. **if** (4\*a\*\*3 + 27 \* b\*\*2) % p != 0 **and** isPrime(int(E.order())): # make sure it  's not a singular curve |
| 30. **return** a,b |
| 31. |
| 32. **while** True: |
| 33. p = int(input("Enter your prime: ")) |
| 34. **if** check(p): |
| 35. **break** |
| 36. secret = random.randint(0,p-1) |
| 37. |
| 38. F = GF(p) |
| 39. a,b = genPara(p) |
| 40. E = EllipticCurve(F, [a,b]) |
| 41. P = E.gens()[0] |
| 42. Q = P \* secret |
| 43. |
| 44. **print**(f'{a = }') |
| 45. **print**(f'{b = }') |
| 46. **print**(f'{p = }') |
| 47. **print**('P =', P.xy()) |
| 48. **print**('Q =', Q.xy()) |
| 49. with open("input.pdf", 'rb') as file: |
| 50. pt = file.read() |
| 51. |
| 52. ciphertext = encrypt(secret, pt) |
| 53. with open("cipher.enc", "wb") as file: |
| 54. file.write(ciphertext) |
| 55. **print**("Write ciphertext to cipher.enc successfully!!!") |

The following code will encrypt input.pdf using AES, but the key is produced from the private key of ECC. So we have to solve ECDLP to get the private key, then get the key to decrypt AES.

Look at the server’s code, we can see that the modulo is a prime and must be greater than 40 bits length. So if we choose a 40-bit length prime (not such a big prime in ECC), we can use Polard’s rho to attack because the Polard’rho take

𝑂(√𝑛) step ( ≈ 220, which is nearing towards the "long" end of computing times, but is still feasible for our purposes). So let’s use the Polard-rho algorithm to attack this:

1. **def** f(Ri, P, Q):

|  |
| --- |
| 2. y = Ri.xy()[1] |
| 3. **if** 0 < y <= p//3: |
| 4. **return** Q + Ri |
| 5. **elif** p//3 < y < 2\*p//3: |
| 6. **return** 2\*Ri |
| 7. **else**: |
| 8. **return** P+ Ri |
| 9. |
| 10. **def** update\_ab(Ri, ai, bi): |
| 11. y = Ri.xy()[1] |
| 12. **if** 0 < y <= p//3: |
| 13. **return** ai, (bi + 1) % n |
| 14. **elif** p//3 < y < 2\*p//3: |
| 15. **return** 2\*ai % n, 2\*bi % n |
| 16. **else**: |
| 17. **return** (ai +1) % n, bi |
| 18. |
| 19. **def** attack(P, Q, n): |
| 20. a = [] |
| 21. b = [] |
| 22. R = [] |
| 23. R.append(P) |
| 24. a.append(1) |
| 25. b.append(0) |
| 26. i = 1 |
| 27. **while** True: |
| 28. R.append(f(R[i-1], P,Q)) |
| 29. ab = update\_ab(R[i-1], a[i-1], b[i-1]) |
| 30. a.append(ab[0]) |
| 31. b.append(ab[1]) |
| 32. **if** i % 2 == 0 **and** R[i] == R[i//2]: |
| 33. m = i//2 |
| 34. **break** |
| 35. i += 1 |
| 36. # print(R) |
| 37. fr = Fraction(int(a[2\*m] - a[m]), int(b[m] - b[2\*m])) |
| 38. a,b = int(fr.numerator), int(fr.denominator) |
| 39. x = a \* pow(b,-1, n) |
| 40. **return** x |

Connect to server to get secret:

1. **while** True:

|  |
| --- |
| 2. r = process(["python3", "server.py"]) |
| 3. a,b,p,P,Q = getPara() |
| 4. E = EllipticCurve(GF(p), [a,b]) |
| 5. n = E.order() |
| 6. P = E(\*P) |
| 7. Q = E(\*Q) |
| 8. **try**: |
| 9. x = attack(P,Q,n) |
| 10. **assert** int(x)\*P == Q |
| 11. **print**(“Found secret, x =”, x) |
| 12. **break** |
| 13. **except**: |
| 14. **print**("Polard’s rho can't solve this!!!") |

After having secret, recover pdf file becomes so easy…

1. **from** Crypto.Cipher **import** AES

3. ciphertext = open("cipher.enc", 'rb').read()

5. key = sha3\_512(str(x).encode()).digest()[:16]

7. ciphertext = ciphertext[16:]

9. with open("recovered.pdf", 'wb') as write:

11. **print**("Generate recovered.pdf successfully!!!")

write.write(unpad(cipher.decrypt(ciphertext),16))

10.

8. cipher = AES.new(key, AES.MODE\_CBC, iv)

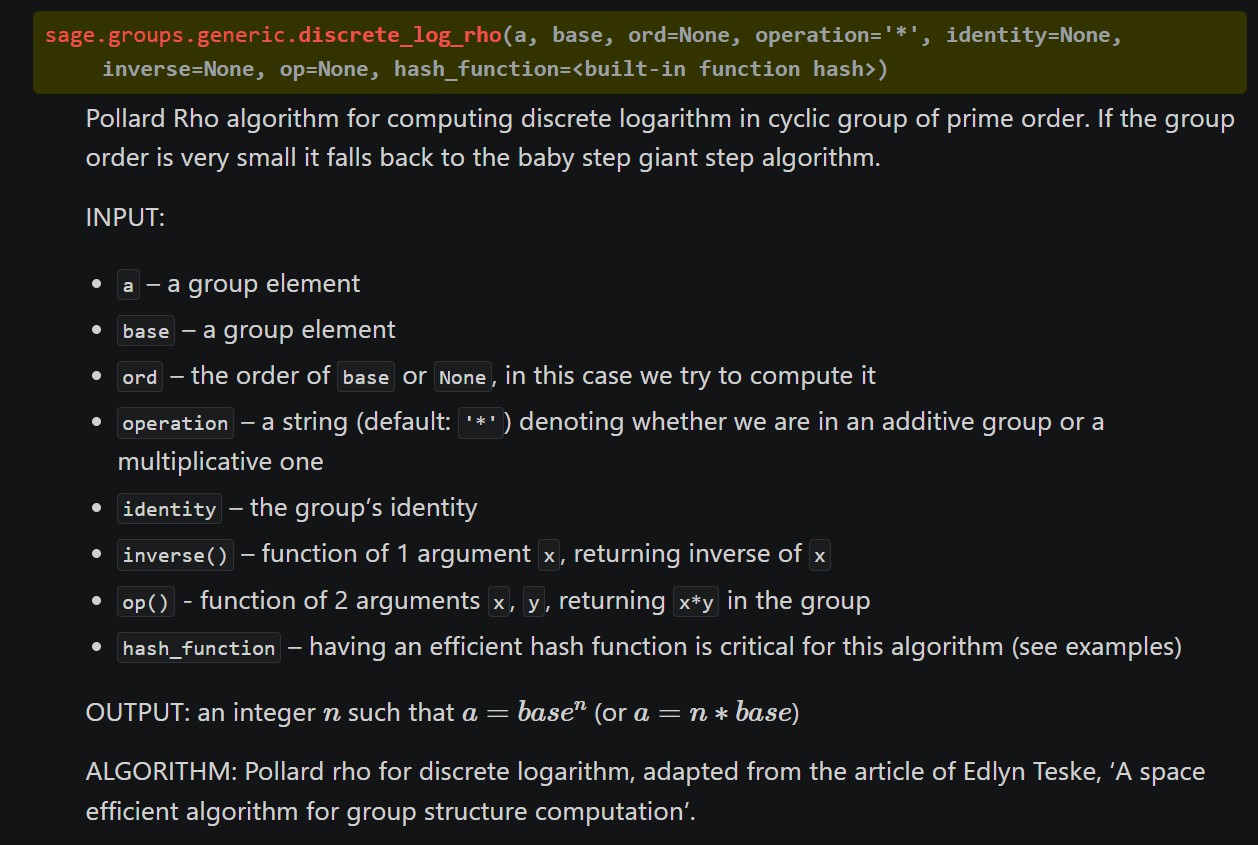
6. iv = ciphertext[:16]

4.

2. **from** hashlib **import** sha3\_512

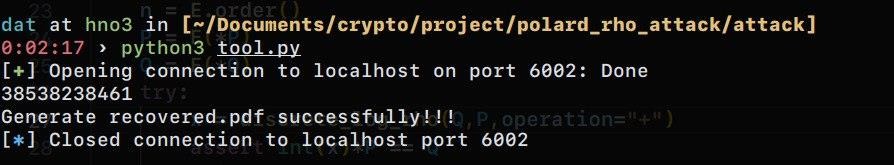


* Sagemath also has a build-in function discrete\_log\_rho that can solve ECDLP using Polard’s rho algorithm.



* Let’s use it to attack

d = discrete\_log\_rho(Q,P, operation="+")



## Pohlig-Hellman attack

* + - Suppose we have a 256-bit curve as follows

- p = 115792089210356248762697446949407573530086143415290314195533631308867097853951

- a = 115792089210356248762697446949407573530086143415290314195533631308867097853948

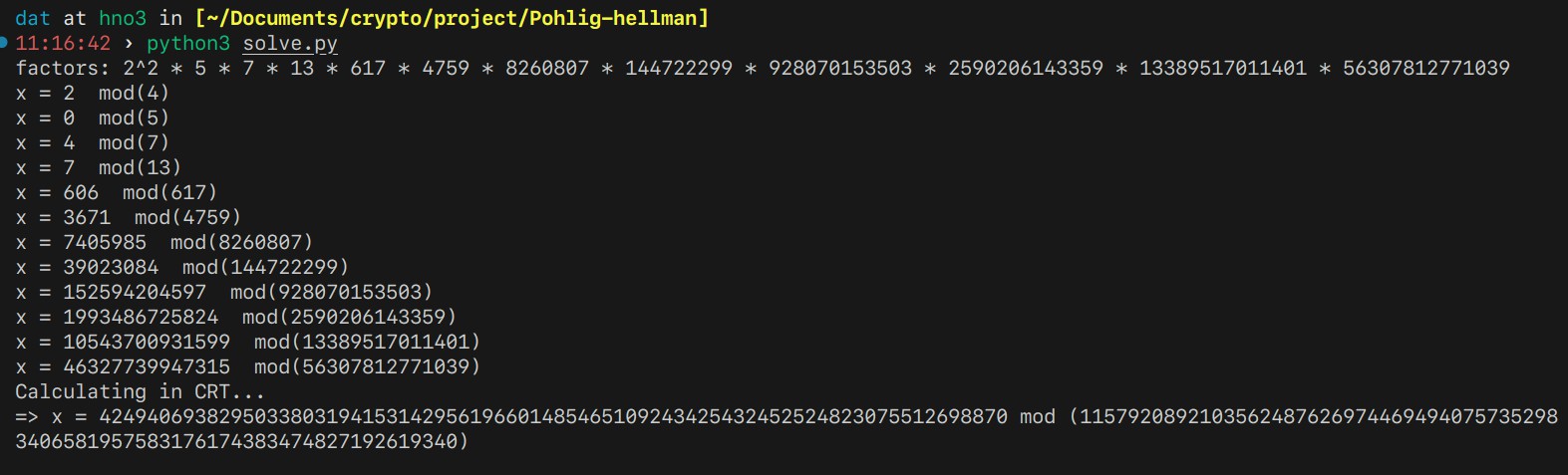
- b = 1235004111951061474045987936620314048836031279781573542995567934901240450608

* + - E = EllipticCurve(GF(p), [a,b])
    - # Generator
    - G = E.gen(0)
    - ​
    - # My secret int, different every time!!
    - n = randint(1, G.order() - 1)
    - # Send this to Bob!
    - public = G \* n
    - Now, according to the Pohlig Hellman algorithm, we will solve this discrete problem as follows:

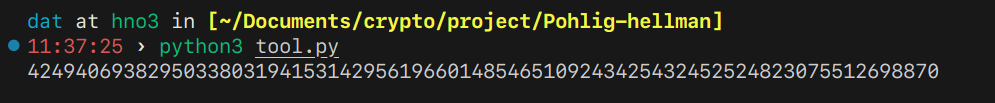
1. p = 115792089210356248762697446949407573530086143415290314195533631308867097853951

|  |
| --- |
| 2. a = 115792089210356248762697446949407573530086143415290314195533631308867097853948 |
| 3. b = 1235004111951061474045987936620314048836031279781573542995567934901240450608 |
| 4. |
| 5. E = EllipticCurve(GF(p), [a,b]) |
| 6. |
| 7. P = E(58739589111611962715835544993606157139975695246024583862682820878769866632269,  86857039837890738158800656283739100419083698574723918755107056633620633897772) |
| 8. Q = E(53389524449399713241908666754198583135391726383277973572010353430393882869587,  96915749683251448875914358893119124077807308307116979366734609648027786948994) |
| 9. with open('cipher.enc', 'r') as f: |
| 10. dataCipher = f.read() |
| 11. iv = bytes.fromhex(dataCipher[0:32]) |
| 12. encrypted\_flag = bytes.fromhex(dataCipher[32:]) |
| 13. |
| 14. n = P.order() |
| 15. fac = factor(n) |
| 16. **print**(f"factors: {fac}") |
| 17. d = [] |
| 18. subgroup = [] |
| 19. **for** prime, exponent **in** fac: |
| 20. P0 = (n // (prime \*\* exponent)) \* P |
| 21. Q0 = (n // (prime \*\* exponent)) \* Q |
| 22. x = discrete\_log(Q0, P0, operation='+') |
| 23. d.append(x) |
| 24. **print**(f"x = {x} mod({prime\*\*exponent})") |
| 25. subgroup.append(prime\*\*exponent) |
| 26. |
| 27. secret = crt(d, subgroup) |
| 28. **print**("Calculating in CRT...") |
| 29. **print**(f"=> x = {secret} mod ({n})") |
| 30. **assert** secret \* P == Q |

* + - As introduced above, we will now solve this discrete problem on subgroups and then use the CRT to find the final result.
    - Here we will generate a generating point P on curve E and have an order of n. Next, analyze n products of prime numbers *(these are also subgroups).* Next, we use the BSGS method introduced above to solve DLP on each small group and finally combine them together and obtain the final result.
    - This is the output of the program:



* + - Sagemath also has a build-in function discrete\_log that can solve ECDLP using Pohlig – Hellman with BSGS.
    - This is the output of code using that function:



And the same result is given.

## Smart attack

Suppose that we have a pdf file encrypted by a following python code:

1. **from** sage.all **import** \*

|  |
| --- |
| 2. **import** random |
| 3. **from** hashlib **import** sha1 |
| 4. **from** Crypto.Cipher **import** AES |
| 5. **from** Crypto.Util.Padding **import** pad |
| 6. # Curve params |
| 7. p = 0xa15c4fb663a578d8b2496d3151a946119ee42695e18e13e90600192b1d0abdbb6f787f90c8d102  ff88e284dd4526f5f6b6c980bf88f1d0490714b67e8a2a2b77 |
| 8. a = 0x5e009506fcc7eff573bc960d88638fe25e76a9b6c7caeea072a27dcd1fa46abb15b7b6210cf90c  aba982893ee2779669bac06e267013486b22ff3e24abae2d42 |
| 9. b = 0x2ce7d1ca4493b0977f088f6d30d9241f8048fdea112cc385b793bce953998caae680864a7d3aa4  37ea3ffd1441ca3fb352b0b710bb3f053e980e503be9a7fece |
| 10. E = EllipticCurve(GF(p), [a,b]) |
| 11. |
| 12. # prime order may protect us from Pohlig-Hellman, right ? :v |

13. **assert** is\_prime(E.order())

15. with open("input.pdf", "rb") as f: 17.

19.

key = sha1(str(key).encode()).digest()[:16]

21.

cipher = AES.new(key, AES.MODE\_CBC, iv)

23.

25. n = random.randint(1,P.order() - 1)

27. **print**(f'Public key: {P\*n}')

29. encrypted = encrypt(n)

31.

cipher.write(encrypted)

30. with open("cipher.enc", "wb") as cipher:

28. # write ciphertext to cipher.enc

26. **print**(f'{P = }')

24. P = E.gen(0)

**return** iv + cipher.encrypt(pad(pt, 16))

22.

iv = random.randbytes(16)

20.

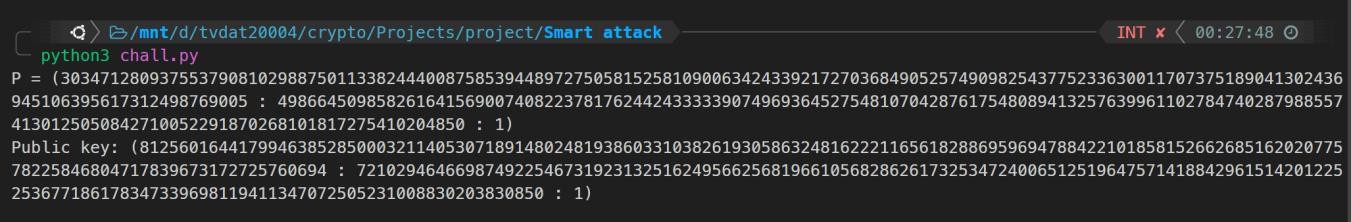
18. **def** encrypt(key):

pt = f.read()

16.

14. # open file to get plaintext

The server give us the output:

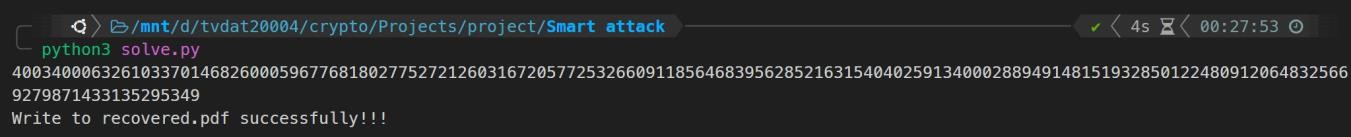


We can see that the curve’s order is a prime to prevent from Pohlig-Hellman attack, but the server have made a fault that the curve’s order is equal to p, which is attacked by Smart attack. With the provided algorithm and some Sagemath function, we can easily decrypt and recover the pdf file.

1. **from** sage.all **import** \*

|  |
| --- |
| 2. **from** hashlib **import** sha1 |
| 3. **from** Crypto.Cipher **import** AES |
| 4. **from** Crypto.Util.Padding **import** unpad |
| 5. p = 0xa15c4fb663a578d8b2496d3151a946119ee42695e18e13e90600192b1d0abdbb6f787f90c8d102  ff88e284dd4526f5f6b6c980bf88f1d0490714b67e8a2a2b77 |
| 6. a = 0x5e009506fcc7eff573bc960d88638fe25e76a9b6c7caeea072a27dcd1fa46abb15b7b6210cf90c  aba982893ee2779669bac06e267013486b22ff3e24abae2d42 |
| 7. b = 0x2ce7d1ca4493b0977f088f6d30d9241f8048fdea112cc385b793bce953998caae680864a7d3aa4  37ea3ffd1441ca3fb352b0b710bb3f053e980e503be9a7fece |
| 8. E = EllipticCurve(GF(p), [a,b]) |
| 9. |
| 10. P = E(303471280937553790810298875011338244400875853944897275058152581090063424339217  2703684905257490982543775233630011707375189041302436945106395617312498769005, 498664  509858261641569007408223781762442433333907496936452754810704287617548089413257639961  1027847402879885574130125050842710052291870268101817275410204850) |
| 11. Q = E(812560164417994638528500032114053071891480248193860331038261930586324816222116  5618288695969478842210185815266268516202077578225846804717839673172725760694, 72102  946466987492254673192313251624956625681966105682862617325347240065125196475714188429  61514201225253677186178347339698119411347072505231008830203830850) |
| 12. enc = open("cipher.enc", "rb").read() |

|  |  |  |
| --- | --- | --- |
| 13.   1. # Lifts a point to the p-adic field. 2. **def** \_lift(E, P, gf): 3. x, y = map(ZZ, P.xy()) 4. **for** point\_ **in** E.lift\_x(x, all=True): 5. \_, y\_ = map(gf, point\_.xy()) 6. **if** y == y\_: 7. **return** point\_ 8. **def** attack(G, P): 9. """ 10. Solves the discrete logarithm problem using Smart's attack. 11. More information: Smart N. P., "The discrete logarithm problem on elliptic curve s of trace one" 12. :param G: the base point 13. :param P: the point multiplication result 14. :return: l such that l \* G == P 15. """ 16. E = G.curve() 17. gf = E.base\_ring() 18. p = gf.order() 19. **assert** E.trace\_of\_frobenius() == 1, f"Curve should have trace of Frobenius = 1." 20. E = EllipticCurve(Qp(p), [int(a) + p \* ZZ.random\_element(1, p) **for** a **in** E.a\_inva riants()]) | | |
| 35. | G = p \* \_lift(E, G, | gf) |
| 36. | P = p \* \_lift(E, P, | gf) |
| 37. | Gx, Gy = G.xy() |  |
| 38. | Px, Py = P.xy() |  |
| 39. | **return** int(gf((Px / | Py) / (Gx / Gy))) |
| 40. |  |  |
| 1. secret = attack(P,Q) 2. **assert** P\*secret == Q 3. **print**(secret) 4. key = sha1(str(secret).encode()).digest()[:16] 5. iv, ct = enc[:16], enc[16:] 6. cipher = AES.new(key, AES.MODE\_CBC, iv) 7. with open("recovered.pdf", "wb") as file: 8. file.write(unpad(cipher.decrypt(ct),16)) 9. **print**("Write to recovered.pdf successfully!!!") | | |



## Invalid curve attack

* + - This is the code snippet for setting up the curve on a server.

- **class** Curve:

-

self.p = p

-

self.b = b

-

**def**  eq (self, other):

-

**return** self.p == other.p **and** self.a == other.a **and** self.b == other.b

-

**return** None

-

**if** isinstance(other, Curve):

-

-

self.a = a

-

**def**  init (self, p, a, b):

-

|  |
| --- |
| - **def**  str (self): |
| - **return** "y^2 = x^3 + %dx + %d over F\_%d" % (self.a, self.b, self.p) |
| - |
| - |
| - **class** Point: |
| - **def**  init (self, curve, x, y): |
| - **if** curve == None: |
| - self.curve = self.x = self.y = None |
| - **return** |
| - self.curve = curve |
| - self.x = x % curve.p |
| - self.y = y % curve.p |
| - |
| - **def**  str (self): |
| - **if** self == INFINITY: |
| - **return** "INF" |
| - **return** "(%d, %d)" % (self.x, self.y) |
| - |
| - **def**  eq (self, other): |
| - **if** isinstance(other, Point): |
| - **return** self.curve == other.curve **and** self.x == other.x **and** self.y == other.y |
| - **return** None |
| - |
| - **def**  add (self, other): |
| - **if not** isinstance(other, Point): |
| - **return** None |
| - **if** other == INFINITY: |
| - **return** self |
| - **if** self == INFINITY: |
| - **return** other |
| - p = self.curve.p |
| - **if** self.x == other.x: |
| - **if** (self.y + other.y) % p == 0: |
| - **return** INFINITY |
| - **else**: |
| - **return** self.double() |
| - p = self.curve.p |
| - l = ((other.y - self.y) \* pow(other.x - self.x, -1, p)) % p |
| - x3 = (l \* l - self.x - other.x) % p |
| - y3 = (l \* (self.x - x3) - self.y) % p |
| - **return** Point(self.curve, x3, y3) |
| - |
| - **def**  neg (self): |
| - **return** Point(self.curve, self.x, self.curve.p - self.y) |
| - |
| - **def**  mul (self, e): |
| - **if** e == 0: |
| - **return** INFINITY |
| - **if** self == INFINITY: |
| - **return** INFINITY |
| - **if** e < 0: |
| - **return** (-self) \* (-e) |
| - ret = self \* (e // 2) |
| - ret = ret.double() |
| - **if** e % 2 == 1: |
| - ret = ret + self |
| - **return** ret |
| - |
| - **def**  rmul (self, other): |
| - **return** self \* other |
| - |

* + - **def** double(self):

|  |
| --- |
| - **if** self == INFINITY: |
| - **return** INFINITY |
| - p = self.curve.p |
| - a = self.curve.a |
| - l = ((3 \* self.x \* self.x + a) \* pow(2 \* self.y, -1, p)) % p |
| - x3 = (l \* l - 2 \* self.x) % p |
| - y3 = (l \* (self.x - x3) - self.y) % p |
| - **return** Point(self.curve, x3, y3) |
| - |
| - |
| - INFINITY = Point(None, None, None) |

* + - And here is the code snippet for using the curve to encrypt data on the server. To align with a real-world server, we have limited the number of requests sent to the server to 3. *Since no server allows continuous requests, we will limit the number of requests to the minimum possible (which is 3)*:
    - with open('input3.pdf', "rb") as f:
    - flag = f.read()
    - f.close()
    - # NIST P-256

- a = -3

- b = 41058363725152142129326129780047268409114441015993725554835256314039467401291

- p = 2\*\*256 - 2\*\*224 + 2\*\*192 + 2\*\*96 - 1

* + - E = Curve(p, a, b)

- n = 115792089210356248762697446949407573529996955224135760342422259061068512044369

- Gx = 48439561293906451759052585252797914202762949526041747995844080717082404635286

- Gy = 36134250956749795798585127919587881956611106672985015071877198253568414405109

* + - G = Point(E, Gx, Gy)
    - # server's secret

- d = 11079208921356230762697446949407573529996920224135760342421115906106851204435

* + - P = G \* d
    - **def** point\_to\_bytes(P):
    - **return** P.x.to\_bytes(32, "big") + P.y.to\_bytes(32, "big")
    - **def** encryptFlag(P, m):
    - key = point\_to\_bytes(P)
    - **return** ARC4.new(key).encrypt(m)
    - **def** encryptPoint(P, m):
    - key = point\_to\_bytes(P)

- m = m.ljust(64, b"\0")

* + - **return** ARC4.new(m).encrypt(key)
    - quotes = [
    - "Konpeko, konpeko, konpeko! Hololive san-kisei no Usada Pekora-peko! domo, domo!",
    - "Bun bun cha! Bun bun cha!",
    - "kitira!",
    - "usopeko deshou",
    - "HA↑HA↑HA↓HA↓HA↓",
    - "HA↑HA↑HA↑HA↑",
    - "it's me pekora!",

|  |
| --- |
| - "ok peko", |
| - ] |
| - |
| - **print**("Konpeko!") |
| - **print**("watashi no public key: %s" % P) |
| - |
| - **for** \_ **in** range(3): |
| - **try**: |
| - **print**("nani wo shitai desuka?") |
| - **print**("1. Start a Diffie-Hellman key exchange") |
| - **print**("2. Get an encrypted flag") |
| - **print**("3. Exit") |
| - option = int(input("> ")) |
| - **if** option == 1: |
| - **print**("Public key wo kudasai!") |
| - x = int(input("x: ")) |
| - y = int(input("y: ")) |
| - S = Point(E, x, y) \* d |
| - **print**(encryptPoint(S,choice(quotes).encode()).hex()) |
| - **elif** option == 2: |
| - r = randbelow(n) |
| - C1 = r \* G |
| - C2 = encryptFlag(r \* P, flag) |
| - **print**(point\_to\_bytes(C1).hex()) |
| - with open("/output/cipher.enc", "wb") as fi: |
| - fi.write(C2) |
| - fi.close() |
| - **elif** option == 3: |
| - **print**("otsupeko!") |
| - **break** |
| - **print**() |
| - **except** Exception as ex: |
| - **print**("kusa peko") |
| - **print**(ex) |
| - **break** |

* + - From the code snippet for setting up the curve for usage, we can observe that a point not lying on the initialized curve can be obtained without encountering any errors because there is no mechanism to check for it.
    - After identifying that the curve setup on the server is not secure, we will now proceed with the attack to find the secret key using the *invalid curve* approach. Since the server only allows a maximum of 3 requests, we will attempt to find 2 curves where the order of the curve is a prime factor greater than or equal to 128 bits and smooth. *The reason for choosing 2 curves like this is because the initial curve of the server is NIST p-256, so the 2 points we send to the server must be at least 128 bits in order to obtain a 256-bit result after performing the final CRT computation, matching the original order size.*
    - Here is the program snippet for us to perform that:

- n = 115792089210356248762697446949407573529996955224135760342422259061068512044369

- F = GF(p)

- p = 2\*\*256 - 2\*\*224 + 2\*\*192 + 2\*\*96 - 1

- a = -3

* + - res = []

|  |
| --- |
| - **while** 1: |
| - b = random.randint(0, p-1) |
| - **print**(f"tmp\_b: {b}") |
| - E = EllipticCurve(F, [a, b]) |
| - G = E.gen(0) |
| - od = G.order() |
| - fac = list(od.factor()) |
| - ar = [] |
| - **for** f, e **in** fac: |
| - **if**(f\*\*e < 2\*\*40): |
| - ar.append(f\*\*e) |
| - **if** prod(ar) >= 2\*\*128: |
| - res.append((b, G.xy(), od, ar)) |
| - **print**(f"final b: {b}") |
| - **print**(f"ar: {ar}") |
| - **if** len(res) == 2: |
| - **break** |
| - |
| - **print**(res) |

* + - Once we obtain the 2 satisfying curves, we will send these points to the server for it to compute the new points P. Then, we will use the Pohlig-Hellman algorithm combined with CRT to solve the discrete logarithm problem with dQi

= Pi.

- params = [(45414823601602260778224209358538085665608006289781851470006386103331059477430, ( 24870727596394817653892652912463435696383977530992674273472974337801990440451, 212702440549

00676788943564170469491390933540024902095537177543727954437071892), 11579208921035624876269

7446949407573529673367188115198453872416350022924704288, [32, 73, 5059, 954977629, 10152551

83, 305568740189, 516269126357]), (44261963214358962003713454237936025295900133434937345271

376756487835813786473, (1218397159775723857701628202775112266105084537096873855607136052081

0338510106, 83603341908040821254892062545236015989108529973669275766118778535863958202332),

115792089210356248762697446949407573530482738211187783374681848292196804100752, [16, 29, 3

907, 518417, 554633, 1039789, 19048433, 26506703, 18553894189])]

- p = 2\*\*256 - 2\*\*224 + 2\*\*192 + 2\*\*96 - 1

* + - r = remote("localhost", 2030)
    - # r = process(["python3", "server.py"])
    - r.recvuntil(b"watashi no public key: ")
    - quotes = [
    - "Konpeko, konpeko, konpeko! Hololive san-kisei no Usada Pekora-peko! domo, domo!",
    - "Bun bun cha! Bun bun cha!",
    - "kitira!",
    - "usopeko deshou",
    - "HA↑HA↑HA↓HA↓HA↓",
    - "HA↑HA↑HA↑HA↑",
    - "it's me pekora!",
    - "ok peko",

- ]

* + - **def** bytes\_to\_point(b : bytes):
    - **return** bytes\_to\_long(b[:32]), bytes\_to\_long(b[32:])
    - **def** point\_to\_bytes(P):
    - **return** P[0].to\_bytes(32, "big") + P[1].to\_bytes(32, "big")

|  |
| --- |
| - **def** get\_dlp(b, x, y): |
| - E = EllipticCurve(GF(p), [-3, b]) |
| - Q = E(x, y) |
| - r.sendlineafter(b"> ", b"1") |
| - r.sendlineafter(b"x: ", str(x).encode()) |
| - r.sendlineafter(b"y: ", str(y).encode()) |
| - ct = bytes.fromhex(r.recvlineS().strip()) |
| - **for** m **in** quotes: |
| - m = m.encode().ljust(64, b"\0") |
| - xy = ARC4.new(m).decrypt(ct) |
| - **try**: |
| - P = E(\*bytes\_to\_point(xy)) |
| - **return** P, Q |
| - **except**: |
| - **pass** |
| - |
| - **def** get\_encrypted\_flag(): |
| - r.sendlineafter(b'> ', b'2') |
| - E = EllipticCurve(GF(p), [-  3, 41058363725152142129326129780047268409114441015993725554835256314039467401291]) |
| - C1 = E(\*bytes\_to\_point(bytes.fromhex(r.recvlineS().strip()))) |
| - with open("output/cipher.enc", 'rb') as file: |
| - C2 = file.read() |
| - **return** C1, C2 |
| - |
| - |
| - # E(Fp), p = f1.f2...fn (factor) |
| - # CRT sol: (p/fn)P = x(p/fn)Q |
| - **def** sol\_dlp(): |
| - sec = [] |
| - mod = [] |
| - **for** b, (x,y), od, subgroups **in** params: |
| - P,Q = get\_dlp(b,x,y) |
| - **for** subgroup **in** subgroups: |
| - **print**(f"solving size {subgroup}") |
| - tmp = od // subgroup |
| - k = discrete\_log(tmp \* P, tmp \* Q, ord=ZZ(subgroup), operation="+") |
| - **print**(f"k: {k} mod {subgroup}") |
| - sec.append(k) |
| - mod.append(subgroup) |
| - **return** crt(sec, mod) |
| - |
| - d = sol\_dlp() |
| - **print**(f"secret = {d}") |
| - |
| - C1, C2 = get\_encrypted\_flag() |
| - |
| - # P = dG, C1 = rG => dC1 = rdG = rP |
| - K = d\*C1 |
| - key = point\_to\_bytes(list(map(int, K.xy()))) |
| - |
| - flag = ARC4.new(key).decrypt(C2) |
| - with open('recover4.pdf', 'wb') as filee: |
| - filee.write(flag) |
| - r.close() |

* + - Here is output

1. solving size 32
2. k: 19 mod 32
3. solving size 73
4. k: 33 mod 73
5. solving size 5059
6. k: 788 mod 5059
7. solving size 954977629
8. k: 635593824 mod 954977629
9. solving size 1015255183
10. k: 82564736 mod 1015255183
11. solving size 305568740189
12. k: 57448838225 mod 305568740189
13. solving size 516269126357
14. k: 143376137800 mod 516269126357
15. solving size 16
16. k: 3 mod 16
17. solving size 29
18. k: 25 mod 29
19. solving size 3907
20. k: 1902 mod 3907
21. solving size 518417
22. k: 233479 mod 518417
23. solving size 554633
24. k: 337839 mod 554633
25. solving size 1039789
26. k: 425311 mod 1039789
27. solving size 19048433
28. k: 14805776 mod 19048433
29. solving size 26506703
30. k: 16506897 mod 26506703
31. solving size 18553894189
32. k: 1065742876 mod 18553894189
33. secret = 11079208921356230762697446949407573529996920224135760342421115906106851204435

*We have included all code of this project in this github* [*link*](https://github.com/tvdat20004/Cryptography-project)*.*

# Deployment:

In this project, we have found some risks that attackers can deploy and attack our system, so to enhance security of ECC-based cryptosystem, we suggest:

* Use recommended curve and strong key: Our curve must be chosen carefully, because most of attacks on ECC are based on the weakness of the curve. …In order to prevent these exploitation happen again, we strong suggest using curve which is verified by NIST or standard curve like Curve25519, M-383, M-511,… Here are some links references to it <https://www.secg.org/sec2-v2.pdf>, <https://neuromancer.sk/std/>
* To avoid invalid curve attack, the server must check all untrusted data from the user, which in this context means we must check the point the user entered is on the curve or not.
* Regularly upgrade, update and periodically test security measures for cryptosystems.

# References

* Silverman, J. H., Pipher, J., & Hoffstein, J. (2008). An introduction to mathematical cryptography (Vol. 1). Springer New York. (<https://link.springer.com/book/10.1007/978-0-387-77993-5>)
* Dubois, R. (2017). Trapping ECC with invalid curve bug attacks. Cryptology ePrint Archive.

(<https://eprint.iacr.org/2017/554.pdf>)

* Wienardo, W., Yuliawan, F., Muchtadi-Alamsyah, I., & Rahardjo, B. (2015, September). Implementation of Pollard Rho attack on elliptic curve cryptography

(<https://sci-hub.se/10.1063/1.4930641>)

* Scholl, T. (2018). Isolated Curves and the MOV Attack. Cryptology ePrint Archive.

(<https://eprint.iacr.org/2018/307.pdf>)

* Breitner, J., & Heninger, N. (2019). Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23* (pp. 3-20). Springer International Publishing.

(<https://eprint.iacr.org/2019/023.pdf>)