

# University of Information Technology

Faculty of Computer Network and Communications



**UIT**

## FINAL REPORT

Subject: Cryptography

Class: NT219.O21.ANTT

Lecturer: TS. Nguyen Ngoc Tu

## **Cryptanalysis on ECC-based Algorithms**

Nguyen Hai Phong – 22521088

Ho Trung Kien – 22520704

Ho Vinh Nhat – 22521013

## Table of Contents

### 1. Contents

1.Contents .....	2
2.Overview .....	3
2.1    Elliptic curve: .....	4
2.2    Elliptic Curves over Finite Fields .....	10
2.3    The Elliptic Curve Discrete Logarithm Problem (ECDLP).....	12
2.3.1    The Double-and-Add Algorithm .....	13
2.3.2    How hard is the ECDLP?.....	14
3.Key generation in ECC.....	15
4.Some attack models in ECC.....	17
4.1    Baby-step giant-step: .....	17
4.2    Polard's rho attack.....	17
4.3    Pohlig – Hellman attack .....	19
4.4    Invalid curve attack.....	20
4.5    Mov attack.....	21
4.6    Frey ruck attack.....	21
4.7    Smart attack.....	21
5.Implementation and testing.....	22
5.1    Baby-step giant-step (BSGS).....	22
Chall.py: .....	22
Solve.py: .....	24
5.2    Polard-rho attack .....	27
Chall.py:.....	27
Solve.py: .....	29
5.3    Pohlig-Hellman attack .....	30
Chall.py:.....	30
Solve.py: .....	31

<b>5.4 Invalid curve attack .....</b>	<b>33</b>
<b>Chall.py:.....</b>	<b>33</b>
<b>Solve.py: .....</b>	<b>35</b>
<b>5.5 Mov attack .....</b>	<b>37</b>
<b>Chall.py: .....</b>	<b>38</b>
<b>Solve.py: .....</b>	<b>39</b>
<b>5.6 Frey ruck attack .....</b>	<b>40</b>
<b>Chall.py: .....</b>	<b>40</b>
<b>Solve.py: .....</b>	<b>41</b>
<b>5.7 Smart attack .....</b>	<b>44</b>
<b>Chall.py: .....</b>	<b>44</b>
<b>Solve.py: .....</b>	<b>45</b>
<b>6.Deployment:.....</b>	<b>48</b>
<b>7.References .....</b>	<b>49</b>

## 2. 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.

## 2.1 Elliptic curve:

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

$$Y^2 = X^3 + AX + B$$

- 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).

$$E1: Y^2 = X^3 - X \quad \text{and} \quad E2: Y^2 = X^3 - X + 1$$

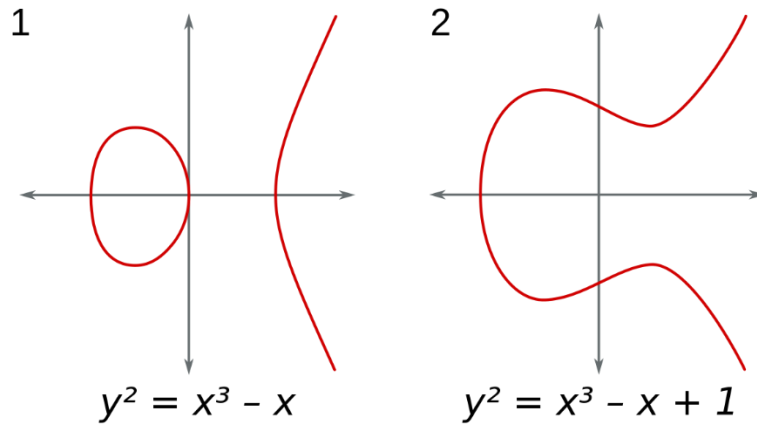


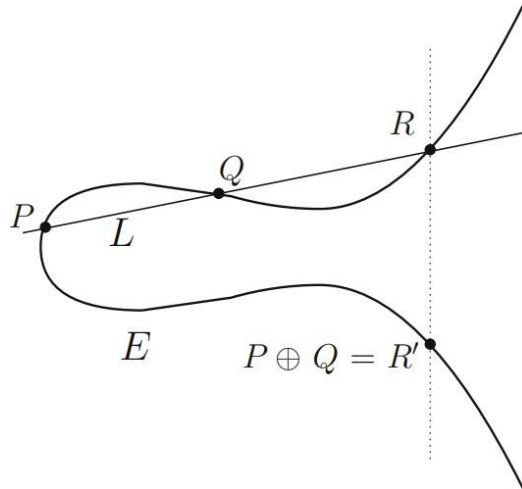
Figure 1

- There are some other forms of Elliptic curve, such as:
  - Montgomery form:  $By^2 = x^3 + Ax^2 + x$
  - Edwards form:  $x^2 + y^2 = 1 + dx^2y^2$
  - Twisted Edwards form:  $ax^2 + y^2 = 1 + dx^2y^2$
  - Doubling-oriented Doche–Icart–Kohel form:  $y^2 = x^3 + ax^2 + 16ax$
  - Tripling-oriented Doche–Icart–Kohel form:  $y^2 = x^3 + 3a(x+1)^2$

### \*\*\* Point addition

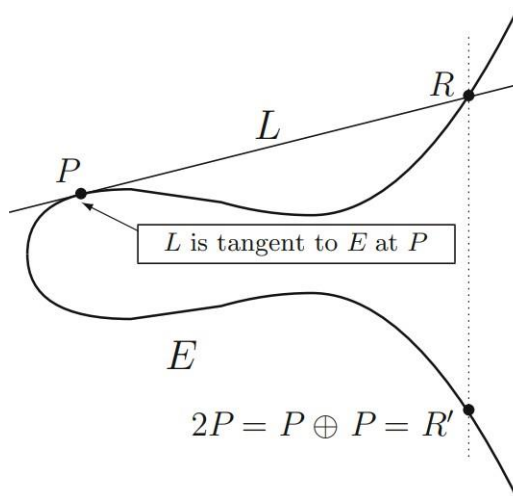
- One of features of elliptic curves is that there is a natural way to take two points on an elliptic curve and “add” them to produce a third point. To visualize the addition of two points on the curve, a geometric representation is presented here.
- Let P and Q be two points on an elliptic curve E, 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  $\oplus$ . Thus, we write:

$$P \oplus Q = R$$



- *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:

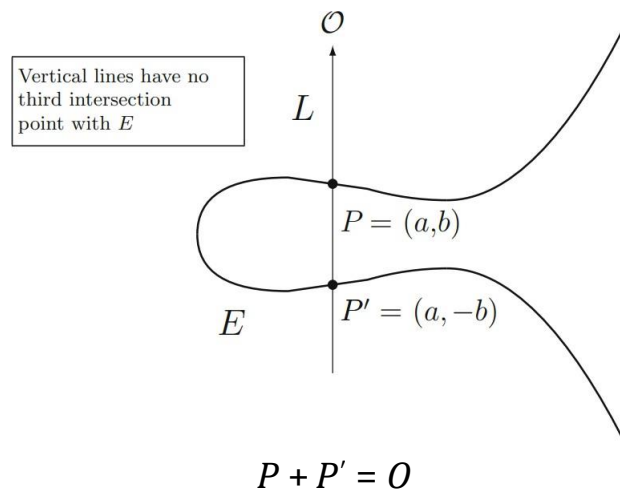
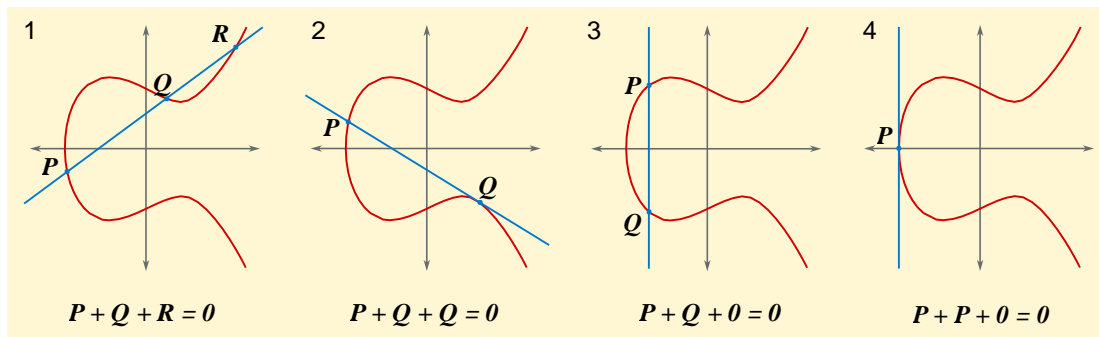


Figure 4: The vertical line  $L$  through  $P = (a, b)$  and  $P' = (a, -b)$

- 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,  $P \oplus O = P$ , so  $O$  acts like zero for elliptic curve addition.



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

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

$$Y^2 = X^3 + AX + B$$

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

$$4A^2 + 27B^3 \neq 0 (*)$$

*Remark (\*):* What is this extra condition  $4A^2 + 27B^3 \neq 0$ ? The quantity  $\Delta E = 4A^2 + 27B^3$  is called the discriminant of  $E$ . The condition  $\Delta E \neq 0$  is equivalent to the condition that the polynomial  $X^3 + AX + B$  have no repeatedly roots.

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

- |                                 |                         |                 |
|---------------------------------|-------------------------|-----------------|
| (a) $P + O = O + P = P$         | for all $P \in E$       | [Identity]      |
| (b) $P + (-P) = O$              | for all $P \in E$       | [Inverse]       |
| (c) $P + (Q + R) = (P + Q) + R$ | for all $P, Q, R \in E$ | [Associative]   |
| (d) $P + Q = Q + P$             | for all $P, Q \in E$    | [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:*

$$E: Y^2 = X^3 + AX + B$$

*be an elliptic curve and let  $P_1$  and  $P_2$  be points on  $E$ .*

- (a) *If  $P_1 = O$ , then  $P_1 + P_2 = P_2$*
- (b) *Otherwise, if  $P_2 = O$ , then  $P_1 + P_2 = P_1$*
- (c) *Otherwise, write  $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$*
- (d) *If  $x_1 = x_2$  and  $y_1 = -y_2$ , then  $P_1 + P_2 = O$*
- (e) *Otherwise, define  $\lambda$  by*

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2, \\ \frac{3x_1^2 + A}{2y_1} & \text{if } P_1 = P_2, \end{cases}$$

*and let*

$$x_3 = \lambda^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1$$

*Then  $P_1 + P_2 = (x_3, y_3)$ .*

## 2.2 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  $F_p$ . This is easy to do.

**Definition:** Let  $p \geq 3$  is a prime. An elliptic curve over  $F_p$  is an equation of the form

$$E: Y^2 = X^3 + AX + B \quad \text{with } A, B \in F_p \text{ satisfying } 4A^3 + 27B^2 \neq 0$$

The set of points on  $E$  with coordinates in  $F_p$  is the set

$$E(F_p) = \{(x, y) : x, y \in F_p \text{ satisfy } y^2 = x^3 + Ax + B\} \cup \{O\}$$

*Example:* Consider the elliptic curve

$$E: Y^2 = X^3 + 3X + 8 \text{ over the field } F_{13}$$

We can find the points of  $E(F_{13})$  by substituting in all possible values  $X = 0, 1, 2, \dots, 12$  and checking for which  $X$  values the quantity  $X^3 + 3X + 8$  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,

$$5^2 \equiv 12 \pmod{13} \text{ and } 8^2 \equiv 12 \pmod{13}.$$

This gives two points  $(1, 5)$  and  $(1, 8)$  in  $E(F_{13})$ . Continuing in this fashion, we end up with a complete list,

$$E(F_{13}) = \{O, (1, 5), (1, 8), (2, 3), (2, 10), (9, 6), (9, 7), (12, 2), (12, 11)\}.$$

Thus,  $E(F_{13})$  contains nine points.

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

Let  $P = (x_1, y_1)$  and  $Q = (x_2, y_2)$  be points in  $E(F_p)$ . We define the sum  $P + Q$  to be the point  $(x_3, y_3)$  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  $F_p$ , we end up with a point  $(x_3, y_3)$  whose coordinates are in  $F_p$ .

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

*(a) The elliptic curve addition algorithm (Theorem 2) applied to  $P$  and  $Q$  yields a point in  $E(F_p)$ . We denote this point by  $P + Q$ .*

*(b) This addition law on  $E(F_p)$  satisfies all of the properties listed in Theorem 1. In other words, this addition law makes  $E(F_p)$  into a finite group.*

*Example: We continue with the elliptic curve*

$$E: Y^2 = X^3 + 3X + 8 \text{ over the field } F_{13}$$

*and we use the addition algorithm (Theorem 2) to add the point  $P=(9,7)$  and  $Q=(1,8)$  in  $E(F_{13})$ . Step (e) of that algorithm tells us to first compute:*

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{8 - 7}{1 - 9} = \frac{1}{-8} = \frac{1}{5} = 8 \pmod{13}$$

*where recall that all computations are being performed in the field  $F_{13}$ , so  $-8 = 5$ ,  $\frac{1}{5} = 5^{-1} = 8$ . Next we compute*

$$v = y_1 - \lambda x_1 = -65 = 0$$

*Finally, the addition algorithm tells us to compute*

$$x_3 = \lambda^2 - x_1 - x_2 = 64 - 9 - 1 = 54 = 2,$$

$$y_3 = -(\lambda x_3 + v) = -8 * 2 = -16 = 10.$$

*This completes the computation of*

$$P + Q = (1,8) + (9,7) = (2,10) \text{ in } E(F_{13}).$$

## 2.3 The Elliptic Curve Discrete Logarithm Problem (ECDLP)

In [Diffie-Hellman key exchange](#), we know about the discrete logarithm (DLP) in the finite field  $F_p$ . In order to create a cryptosystem based on the DLP for  $F_p$ , Alice publishes two numbers  $g$  and  $h$ , and her secret is the exponent  $x$  that solves the congruence

$$h \equiv g^x \pmod{p}$$

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

$$h \equiv g * g * g \dots g \pmod{p} \quad (x \text{ multiplications})$$

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  $E(F_p)$  of an elliptic curve  $E$  over a finite field  $F_p$ . She chooses and publishes two points  $P$  and  $Q$  in  $E(F_p)$ , and her secret is an integer  $n$  that makes

$$Q = \underbrace{P + P + P + \dots + P}_{n \text{ additions on } E} = nP.$$

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  $F_p$  and let  $P$  and  $Q$  be points in  $E(F_p)$ . 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  $F_p$ , we denote this integer  $n$  by

$$n = \log_P(Q)$$

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

### 2.3.1 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  $P, 2P, 3P, \dots$

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

$$n = n_0 + n_1 * 2 + n_2 * 4 + \dots + n_r * 2^r \text{ with } n_0, n_1, \dots, n_r \in \{0,1\}$$

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

$$Q_0 = P, Q_1 = 2Q_0, \dots, Q_r = 2Q_{r-1}.$$

Notice that  $Q_i$  is simply twice the previous  $Q_{i-1}$ , so:

$$Q_i = 2^i P$$

**Input: Point  $P \in E(\mathbb{F}_p)$  and integer  $n \geq 1$**

1. Set  $Q = P$  and  $R = O$ .
2. Loop while  $n > 0$ .
  3. If  $n \equiv 1 \pmod{2}$ , set  $R = R + Q$ .
  4. Set  $Q = 2Q$  and  $n = \lfloor n/2 \rfloor$
  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,

$$nP = n_0 P_0 + n_1 Q_1 + \dots + n_r Q_r$$

We'll refer to the addition of two points in  $E(\mathbb{F}_p)$  as a point operation. Thus the total time to compute  $nP$  is at most  $2r$  point operations in  $E(\mathbb{F}_p)$ . Notice that  $n \geq 2^r$ , so it takes no more than  $2 \log_2 n$  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.

### 2.3.2 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  $O(\sqrt{p})$  steps. In other words, despite the highly structured nature of the group  $E(F_p)$ , 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  $E(F_p)$  takes approximately  $\sqrt{p}$  steps**

### 3. 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  $F_p$
    - **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.

- Example code for key generation using Curve25519 using Sagemath:

- Sample output:

16 | Page



## 4. Some attack models in ECC

### 4.1 Baby-step giant-step:

- The baby-step giant-step is a “meet in the middle” algorithm. Contrary to the brute-force attack, we will calculate a precomputed table of values for  $bP$  and values for  $Q - amP$  until we find some correspondence.
- Baby-step giant-step algorithm base on the facts that we can always write any integer  $x$  as  $x = am + b$ , where  $a, m$  and  $b$  are three arbitrary integers. With this in mind, we can rewrite the equation for the discrete logarithm problem as follows:

$$\begin{aligned}Q &= xP \\Q &= (am + b)P \\Q &= amP + bP \\Q - amP &= bP\end{aligned}$$

- The algorithm works as follows:
  1. Calculate  $m = \lceil \sqrt{n} \rceil$
  2. For every  $a$  in  $0, 1, \dots, m$ : calculate  $bP$  and store the results in a hash table.
  3. For every  $b$  in  $0, 1, \dots, m$ :
    - a. Calculate  $amP$
    - b. Calculate  $Q - amP$
    - c. Check the hash table and look if there exist a point  $bP$  such that  $Q - amP = bP$
    - d. If such point exists, then we have found  $x = am + b$
- Modern computing hardware has practical limits on how much data can be stored .For  $n = 2^{80}$  the storage requirement is  $\sqrt{n} = 2^{40}$  group element and At 32 bytes per element this amounts to  $2^{40} \times 32 \text{ bytes} = 2^{45} \text{ bytes}$  or 32 terabytes .

### 4.2 Pollard'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 can solve the ECDLP

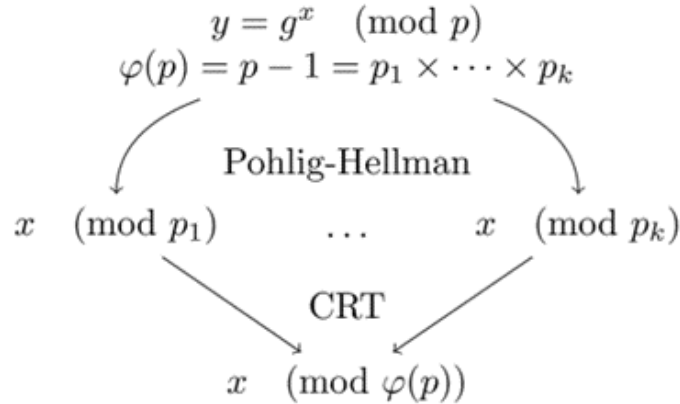


where the intersection happened, we can use the equations above to figure out the discrete logarithm.

- The practicality of an attack is based on the number of operations required. For a group order  $n$  of  $2^{160}$ , the algorithm requires approximately  $2^{80}$  operations. This is considered computationally infeasible with current technology.

### 4.3 Pohlig – Hellman attack

- The Pohlig-Hellman algorithm was presented by Stephan C. Pohlig and Martin E. Hellman in 1978. In the original paper it is presented as an improved algorithm used to compute discrete logarithms over the cyclic field  $G = GF(p)$ , and how their findings impact elliptic curve cryptography.
- The strategy is that given the ECDLP  $Q = lP$ , the Pohlig-Hellman algorithm is a recursive algorithm that reduces the problem by computing discrete logarithms in the prime order subgroups of  $\langle P \rangle$ . Each of these smaller subproblems can then be solved using methods, such the Pollards rho algorithm.



- The Pohlig-Hellman algorithm works as follows:
  - + Write  $n$  as  $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$
  - + Compute  $l_i = l \bmod p_i^{e_i}$  for all  $1 \leq i \leq r$
- The Chinese Remainder Theorem is then used to obtain a unique solution to the following system of congruences:

$$\begin{aligned} l &\equiv l_1 \pmod{p_1^{e_1}} \\ l &\equiv l_2 \pmod{p_2^{e_2}} \\ &\dots \\ l &\equiv l_r \pmod{p_r^{e_r}} \end{aligned}$$

- Here  $p_1, p_2, \dots, p_r$  is a mutually coprime set of positive integers, i.e.  $\gcd(p_i, p_j) = 1$  for all  $i \neq j$ .  $l_1, l_2, l_3, \dots, l_r$  are all the positive integer such that  $0 \leq l_i < p_i$ . The unique positive integer  $l$  can be computed efficiently by using the Extended Euclidean Algorithm to compute the linear congruences above

Each  $l_i$  can then be expressed in base  $p$  the following way:

$$li = z_0 + z_1p + z_2p^2 + \dots + z_{e-1}p^{e-1}$$

+ Where  $z_i \in [0, p - 1]$ . Let

$$P_0 = \frac{n}{p_i}$$

$$\text{And } Q_0 = \frac{n}{p_i} Q$$

+ Rewriting the equations, and using the fact that the order of  $P_0$  is  $p$ , we get

$$Q_0 = lP_0 = z_0P_0$$

+ Then  $z_0$  is found by computing the ECDLP solution in  $P_0$ . Every  $z_0, \dots, z_{e-1}$  is then computed by solving  $Q_i = z_i P_0$  in  $\langle P_0 \rangle [1, 3]$ .

- $p_i \leq 2^{160}$ , because  $2^{160}$  is the limit where solving the Chinese Remainder Theorem is taking so much time and solving the problem is infeasible.

#### 4.4 Invalid curve attack

- Invalid Curve Attack is mainly relying on the fact that given the Weierstrass equation:  $y^2 = x^3 + ax + b$  of an elliptic curve over a prime field  $E(F_p)$  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).

Doubling	Addition
If $y = 0$ then $2P = P_\infty$ , else $\lambda = \frac{3x^2 + a}{2y}$ $x_2 = \lambda^2 - 2x$ $y_2 = -\lambda^3 + 3\lambda x - y$	$\lambda = \frac{y_1 - y_2}{x_1 - x_2}$ $x_3 = \lambda^2 - x_1 - x_2$ $y_3 = -\lambda^3 + 2\lambda x_1 + \lambda x_2 - y_1$

Table 1. Doubling and Addition law over  $E(F_p)$

- Thus, if a point is not checked to be on the curve, it could lead to compute over a curve  $\tilde{E}(F_p)$  of equation  $y^2 = x^3 + ax + c$ . The attacker can select  $\tilde{E}(F_p)$  with some weak security properties (for example a very smooth order) to make the server side calculate point additions, multiplications on that curve for further exploitation.
- Recently, this attack was extended to several models of curves (including Edwards and Hessian)
- For  $n$  bits curve order  $E$ , we will send different points from different curves (which have smooth order) and solve ECDLP using Pohlig-Hellman attack describe as above until we have enough of known  $n$  bits of secret to use Chinese Remainder Theorem and recover the full secret. For example, if  $n = 256$ :

- $y^2 = x^3 + ax + c_1 \pmod{p}$  (leak  $n_1$  bits)
- $y^2 = x^3 + ax + c_2 \pmod{p}$  (leak  $n_2$  bits)
- $y^2 = x^3 + ax + c_3 \pmod{p}$  (leak  $n_3$  bits)
- ...
- $y^2 = x^3 + ax + c_n \pmod{p}$  (leak  $n_n$  bits)

If  $n_1 + n_2 + n_3 + \dots + n_n \geq 256$  we can recover secret key using CRT.

- The simple way to circumvent this attack is to check if a given point is on a curve.

#### 4.5 Mov attack

- The MOV attack, named after Menezes, Okamoto, and Vanstone (1993), uses the Weil pairing to convert discrete log problem in  $E(F_p)$  to one in  $F_{p^k}^*$
- The MOV attack transfers a discrete log from  $E(F_p)$  to  $F_{p^k}^*$  for some positive integer  $k$ . The idea is to leverage sub-exponential time algorithms for solving discrete logs in the multiplicative group of a finite field. A necessary condition for this transfer is that  $|E(F_p)|$  divides  $p^k - 1$ . The smallest possible  $k$  is called the embedding degree of  $E$ . This is the same as the multiplicative order of  $p$  in  $(\mathbb{Z}/N\mathbb{Z})^*$  where  $N = |E(F_p)|$ .
- Suppose we are given points  $P, nP$  of an elliptic curve and asked to recover  $x$  (This is the discrete logarithm problem.)
- Let  $e()$  be the Weil pairing. Let  $m$  be the order of  $P$ . Let  $Q$  be a point of order that is linearly independent to  $P$  (in other words, there is no  $n$  such that  $Q = nP$ ).
- Then  $e(P, Q)$  and  $e(xP, Q) = e(P, Q)^x$  can be computed and are both elements of a finite field. (They are  $m$ -th roots of unity). Since  $P, Q$  are linearly independent, cannot  $e(P, Q)$  be unity by the nondegeneracy of the Weil pairing. Thus, we have reduced the discrete logarithm problem on the group of points on an elliptic curve to the discrete logarithm on finite fields, where sub exponential attacks are known.
- If  $k > \log^2 p$ , then the MOV attack will not be faster than trying to solve the discrete log on  $E$  directly. Therefore, we should make curves with embedding degree  $k > \log^2 p$ .

#### 4.6 Frey ruck attack

#### 4.7 Smart attack

- Smart attack is first announced when curves with order equal to  $p$ , and is recommended to prevent MOV attack.
- Smart attack describes a linear time method of computing of ECDLP in curves over a field  $F_p$  such that  $\#E(F_p) = p$ , or in other words such that trace of Frobenius is one,  $t = p + 1 - \#E(F_p) = 1$
- First recall that we are trying to find  $k$  such that  $Q = kP$  where  $Q, P \in E(F_p)$  and  $\#E(F_p) = p$  (order of curve  $E(F_p)$ ). Our first step is to lift these points to  $E(F_{p^k})$  to get two new points  $P', Q'$ . We do this by setting the  $x$  component of  $P$  equal to the  $x$

component of  $P$ . We then use Hensel's Lemma described above to compute  $y$  in  $Q_p$  ( $p$ -adic field) by using the curve equation with  $x$  fixed. We know that  $Q = kP$  in  $E(F_p)$  so thus the value  $Q' - kP'$  in  $E(Q_p)$  goes to the point at infinity by the Reduction Modulo  $P$  map and is thus in the kernel of that homomorphism.

$$Q - kP' \in E_1(Q_p) \text{ ( } E_1(Q_p) \text{ kernel of } E(Q_p) \text{)}$$

- Now we rely on the fact that the order of  $E(F_p)$  is  $p$ , which ensures that multiplying any element in  $E(Q_p)$  by  $p$  maps the elements into  $E_1(Q_p)$  since for any point  $R \in E(Q_p)$  the point  $pR$  will map via Reduction Modulo  $P$  to  $O$  in  $E(F_p)$ . So multiply through by  $p$  and we get:

$$pQ' - k(pP') \in E_2(Q_p) \text{ ( } E \text{ kernel of } E(Q_p) \text{)}$$

with  $pQ' \in E_1(Q_p)$  and  $pP' \in E_1(Q_p)$ . We can now apply the  $p$ -adic Elliptic Log to get:

$$\psi(pQ') - k\psi_p(pP') \in pZ_p$$

And thus

$$k = \frac{\psi_p(pQ')}{\psi_p(pP')}$$

And then reduce  $k$  modulo  $p$  to return to  $F_p$  solving ECDLP.

## 5. 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.

### 5.1 Baby-step giant-step (BSGS)

Chall.py:

File “chall.py”, first encrypt the pdf file by secret key, after that generate 10 non-singular curve with 32-bit  $p$ , with  $a, b$  random also having prime order, each curve create private key  $x$ , give public point  $G$  and  $G*x$  to the client. If pass 10 times, server will give the client the key.

```
from sage.all import *
from Crypto.Util.number import getPrime
from Crypto.Cipher import AES
import random
from secret import key

assert len(key) == 16
assert type(key) == bytes
```

```

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) %
block_size))

def encrypt(key, filein):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open('encrypted.enc', 'wb') as f:
        f.write(cipher.encrypt(pad(data,16)))
    print(f"Encrypted {filein} with key")

def gen_curve():
    while True:
        p = getPrime(32) # 32-bit prime
        a = randint(1, p-1)
        b = randint(1, p-1)
        try:
            E = EllipticCurve(GF(p), [a, b])
            if (4*a**3 + 27*b**2) % p == 0 or not is_prime(E.order()): # singular curve
                continue
            return p,a,b
        except ArithmeticError:
            pass

def welcome():
    print("Welcome to the Elliptic Curve Discrete Logarithm Problem!")
    print("You will be given a curve E over a finite field F_p, and a point G on E.")
    print("You will be asked to compute G*x for a random x, and you need to enter x.")
    print("If G*x = (x, y), you need to enter x.")
    print("If G*x is not a valid point on E, you will be given another chance.")
    print("If you enter the wrong x, the program will terminate.")
    print("We will handout the key if you can solve 10 rounds.")
    print("Good luck!\n")

if __name__ == "__main__":
    encrypt(key, "2015-605.pdf")
    welcome()
    p, a, b = gen_curve()
    E = EllipticCurve(GF(p), [a, b])
    print(f"{p = } \n{a = } \n{b = }")
    try:
        G = E.random_point()
        print(f"G = {G.xy()}")
        for i in range(10): # 10 rounds
            x = random.randint(1, E.order())

```

```

        print("G*x =", (G*x).xy())
        x_input = int(input("Enter x: "))
        if(G*x == G*x_input):
            if(i == 9):
                print("Congratulations! You have solved all 10 rounds!")
                print(key.decode())
                break
            print("Correct, next round!")
        else:
            print("Wrong, bye!")
            break
    except Exception as e:
        print(e)
        print("[-] Something's wrong!")

```

Solve.py:

```

from sage.all import *
from Crypto.Cipher import AES
from pwn import process, remote
import multiprocessing

def compute_baby_steps(start, end, P, queue):
    lookup = {j * P: j for j in range(start, end)}
    queue.put(lookup)

def compute_giant_steps(start, end, m, P, Q, lookup_table, queue):
    for i in range(start, end):
        temp = Q - (i * m) * P
        if temp in lookup_table:
            queue.put((i * m + lookup_table[temp]) % P.order())
            return
    queue.put(None)

def bsgs_ecdlp(P, Q, E, num_processes=8):
    if Q == E((0, 1, 0)):
        return P.order()
    if Q == P:
        return 1

    m = ceil(sqrt(P.order()))
    chunk_size = (m + num_processes - 1) // num_processes

    # Precompute the Lookup table (baby steps) using multiple processes
    queue = multiprocessing.Queue()
    processes = [
        multiprocessing.Process(target=compute_baby_steps, args=(i * chunk_size,
min((i + 1) * chunk_size, m), P, queue))

```



```

        for i in range(num_processes)
    ]

    for p in processes:
        p.start()

    lookup_table = {}
    for _ in range(num_processes):
        lookup_table.update(queue.get())

    for p in processes:
        p.join()

    # Compute the giant steps in parallel
    processes = [
        multiprocessing.Process(target=compute_giant_steps, args=(i * chunk_size,
min((i + 1) * chunk_size, m), m, P, Q, lookup_table, queue))
        for i in range(num_processes)
    ]

    for p in processes:
        p.start()

    result = None
    for _ in range(num_processes):
        res = queue.get()
        if res is not None:
            result = res
            break

    for p in processes:
        p.terminate()

    return result

def unpad(data, block_size):
    return data[:-data[-1]]

def decrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(unpad(cipher.decrypt(data), 16))
    print(f"Decrypted file {filein} to file {fileout}")

```

```

if __name__ == "__main__":
    io = remote("localhost", 8001)
    # io = process(["python3", "chall.py"]) # Local testing

    io.recvuntil(b"p = ")
    p = int(io.recvline().strip())
    io.recvuntil(b"a = ")
    a = int(io.recvline().strip())
    io.recvuntil(b"b = ")
    b = int(io.recvline().strip())

    E = EllipticCurve(GF(p), [a, b])
    print(f"{p = } \n{a = } \n{b = }")
    print(E) #  $y^2 = x^3 + ax + b$ 

    io.recvuntil(b"G = ")
    G = E(eval(io.recvline().strip()))
    print("G =", G.xy()) #  $G = (x, y)$ 

    for i in range(10): # 10 rounds
        print("Round", i+1)
        io.recvuntil(b"G*x = ")
        xG = E(eval(io.recvline().strip()))
        print("G*x =", xG.xy())
        io.recvuntil(b"Enter x: ") # Enter x:
        x = bsgs_ecdlp(G, xG, E, 12)
        print("Found x:", x)
        io.sendline(str(x).encode())
        print(io.recvline()) # Correct, next round! or Congratulations! You have solved
all 10 rounds!

    key = io.recvline().strip() # bytes
    print("Key:", key)
    decrypt(key, "encrypted.enc", "decrypted.pdf")
    io.close()

```

Final output:

```
/mnt/e/St/Cr/P/C/EC_cryptographic_attacks/baby_step_giant_step_attack P main ?1
python3 solve.py
[+] Opening connection to localhost on port 8001: Done
p = 4175795947
a = 373489667
b = 3057991173
Elliptic Curve defined by  $y^2 = x^3 + 373489667x + 3057991173$  over Finite Field of size 4175795947
G = (2266017146, 2378987508)
Round 1
G*x = (1426931532, 3764909968)
Found x: 206301846
b'Correct, next round!\n'
Round 2
G*x = (2204393595, 2583152596)
Found x: 677898618
b'Correct, next round!\n'
Round 3
G*x = (458103365, 2229640564)
Found x: 1295474252
b'Correct, next round!\n'
Round 4
G*x = (772332331, 2568177090)
Found x: 1626391559
b'Correct, next round!\n'
Round 5
G*x = (299517171, 779559855)
Found x: 3932687129
b'Correct, next round!\n'
Round 6
G*x = (1069102977, 2115901941)
Found x: 788090484
b'Correct, next round!\n'
Round 7
G*x = (3052937110, 2159942042)
Found x: 2253080235
b'Correct, next round!\n'
Round 8
G*x = (2019093434, 1876292398)
Found x: 3826040243
b'Correct, next round!\n'
Round 9
G*x = (2661611096, 1336808955)
Found x: 3961575816
b'Correct, next round!\n'
Round 10
G*x = (753304122, 3898870008)
Found x: 3720364670
b'Congratulations! You have solved all 10 rounds!\n'
Key: b'testkey{Nh4tg4'}"
Decrypted file encrypted.enc to file decrypted.pdf
[*] Closed connection to localhost port 8001
```

## 5.2 Polard-rho attack

File “chall.py”, equivalent to the baby-step giant-step example, give the client the base point  $G$  and  $P = secret * G$  of a 32-bit curve  $E$ . The key is also used for encrypting the pdf file. If the client could recover the secret, they will decrypt the file.

Chall.py:

```
from sage.all import *
from Crypto.Cipher import AES
from Crypto.Util.number import isPrime, getPrime
```

```

from Crypto.Util.Padding import pad
from hashlib import sha1
import random

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) %
block_size))

def encrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(cipher.encrypt(pad(data,16)))
    print(f"Encrypted file {filein} to file {fileout}")

def genPara():
    while True:
        p = getPrime(32)
        a,b = random.randrange(0,p-1), random.randrange(0,p-1)
        E = EllipticCurve(GF(p), [a,b])
        if (4*a**3 + 27 * b**2) % p != 0 and isPrime(int(E.order())): # make sure it's
not a singular curve
            return p,a,b

if __name__ == "__main__":
    print("Welcome to Elliptic Curve Cryptography!!!")
    print("We are going to encrypt the content of Project.pdf using Elliptic Curve
Cryptography.")
    print("Please wait a moment, we are generating parameters for Elliptic Curve...")

    p,a,b = genPara()
    F = GF(p)
    E = EllipticCurve(F, [a,b])
    P = E.gens()[0]
    secret = random.randint(0,p-1)
    Q = P * secret

    print(f'{a = }')
    print(f'{b = }')
    print(f'{p = }')
    print('P =', P.xy())
    print('Q =', Q.xy())

    encrypt(int.to_bytes(secret, 16), 'Project.pdf', "encrypted.enc")

```

Solve.py:

```
from sage.all import *
from Crypto.Cipher import AES
from pwn import remote, process

def compute_next(P,Q,Ri,ai,bi):
    y = Ri.xy()[1]
    if 0 <= y <= n//3:
        return Q+Ri, ai, (bi+1) % n
    elif n//3 <= y <= 2*n//3:
        return 2*Ri, 2*ai % n, 2*bi % n
    else:
        return P+Ri, ai+1 % n, bi

def pollard_rho(P,Q,a,b):
    common_list = {}
    R = a*P + b*Q
    while R not in common_list:
        common_list[R]=(a,b)
        R, a, b = compute_next(P,Q,R,a,b)
    c,d = common_list[R]
    assert b != d, "b cant be d!"
    return int((a-c) * pow(d-b,-1,n) % n) # convert Integer (sage) to int (python)

def unpad(data, block_size):
    return data[:-data[-1]]

def decrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(unpad(cipher.decrypt(data),16))
    print(f"Decrypted file {filein} to file {fileout}")

if __name__ == "__main__":
    io = remote("localhost", 8002)
    # io = process(["python3", "chall.py"]) # local testing
    print(io.recvuntil(b"Please wait a moment, we are generating parameters for Elliptic Curve...\n").decode())
    a = int(io.recvlineS().split('=')[1].strip())
    b = int(io.recvlineS().split('=')[1].strip())
    p = int(io.recvlineS().split('=')[1].strip())
    P = eval(io.recvlineS().split('=')[1].strip())
    Q = eval(io.recvlineS().split('=')[1].strip())
    io.recvline()
```

```

print(f'{a = }')
print(f'{b = }')
print(f'{p = }')
print('P =', P)
print('Q =', Q)

F = GF(p)
E = EllipticCurve(F, [a,b])
P = E(*P)
Q = E(*Q)
n = E.order()
d = pollard_rho(P,Q,a,b)
assert P*d == Q, "Wrong secret :<" # check if d (secret) is correct
print("found secret!")
print("secret =",d)

decrypt(int.to_bytes(d, 16), "encrypted.enc", "decrypted.pdf")

```

Final output:

```

/mnt/e/St/Cr/Projects/CryptoProject/EC_cryptographic_attacks/pollard_rho_attack
python3 solve.py
[+] Opening connection to localhost on port 8002: Done
Welcome to Elliptic Curve Cryptography!!!
We are going to encrypt the content of Project.pdf using Elliptic Curve Cryptography.
Please wait a moment, we are generating parameters for Elliptic Curve...

a = 1411358680
b = 674657445
p = 3199880869
P = (2804330088, 774820765)
Q = (921087832, 590541333)
found secret!
secret = 301197652
Decrypted file encrypted.enc to file decrypted.pdf
[*] Closed connection to localhost port 8002

```

### 5.3 Pohlig-Hellman attack

The server using a constant curve with defined  $p, a, b$  (256-bit), the curve  $E$  defined by  $p, a, b$  is secretly a smooth order curve, we will apply polig-hellman attack for this kind of vulnerability to recover the secret.

Chall.py:

```

from sage.all_cmdline import *
from Crypto.Cipher import AES

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) %
block_size))

def encrypt(key, filein, fileout):
    with open(filein, 'rb') as f:

```





```

print(f"factored: {factors}")
dlogs = []
primes = []
total_bit_prime = 0
for prime, exponent in factors:
    P_0 = (order // (prime ** exponent)) * P
    Q_0 = (order // (prime ** exponent)) * Q
    log = discrete_log(Q_0, P_0, operation='+', algorithm='rho')
    dlogs.append(log)
    print(f"DL found = {log} mod({prime**exponent})")
    primes.append(prime**exponent)

    # Calculate total bit length of primes
    total_bit_prime += (prime**exponent).bit_length()
    if bound:
        print(f"total_bit_prime: {total_bit_prime} bound: {bound}")
        if total_bit_prime > bound:
            break

print("Calculating in CRT...")
secret = int(crt(dlogs, primes))
return secret

if __name__ == "__main__":
    io = remote("localhost", 8003)
    # io = process(["python3", "chall.py"]) # Local testing

    # Elliptic curve parameters
    p = int(io.recvline().strip().split(b" = ")[1])
    a = int(io.recvline().strip().split(b" = ")[1])
    b = int(io.recvline().strip().split(b" = ")[1])
    G = eval(io.recvline().strip().split(b" = ")[1])
    P = eval(io.recvline().strip().split(b" = ")[1])
    E = EllipticCurve(GF(p), [a, b])
    G = E(G)
    P = E(P)

    # find the secret key with known bound of 128 bits
    secret = pohlig_hellman(G, P, bound=128)

    decrypt(secret.to_bytes(16, 'big'), "encrypted.enc", "decrypted.pdf")

```

Final output:



```
python3 solve.py
[+] Opening connection to localhost on port 8003: Done
factors: 2 * 2 * 5 * 7 * 13 * 617 * 4759 * 8260807 * 144722299 * 928070153503 * 2590206143359 * 13389517011401 * 56307812771039
DL found = 3 mod(4)
total_bit_prime: 3 bound: 128
DL found = 2 mod(5)
total_bit_prime: 6 bound: 128
DL found = 4 mod(7)
total_bit_prime: 9 bound: 128
DL found = 3 mod(13)
total_bit_prime: 13 bound: 128
DL found = 156 mod(617)
total_bit_prime: 23 bound: 128
DL found = 4716 mod(4759)
total_bit_prime: 36 bound: 128
DL found = 6352348 mod(8260807)
total_bit_prime: 59 bound: 128
DL found = 140524930 mod(144722299)
total_bit_prime: 87 bound: 128
DL found = 838156726596 mod(928070153503)
total_bit_prime: 127 bound: 128
DL found = 665925764517 mod(2590206143359)
total_bit_prime: 169 bound: 128
Calculating in CRT...
Decrypted file encrypted.enc to file decrypted.pdf
[*] Closed connection to localhost port 8003
```

## 5.4 Invalid curve attack

This is a typical implement of the invalid curve attack, whatever the point the server receive if the server side does not check, will raise the chance of being vulnerable to invalid curve attack. The server side using Curve25519, and computing the shared secret between the client and server every time the client sends their point and print it. The client can send multiple point not on the curve and attack.

Chall.py:

```
from Crypto.Util.number import inverse, bytes_to_long
from Crypto.Cipher import AES
from random import randint

def add(P, Q, E):
    if (P == (0, 0)):
        return Q
    elif (Q == (0, 0)):
        return P
    else:
        Ea, Ep = E['a'], E['p']
        x1, y1 = P
        x2, y2 = Q
        if (x1 == x2 and y1 == -y2 % Ep):
            return (0, 0)
        else:
            if P != Q:
                l = (y2 - y1) * inverse(x2 - x1, Ep)
            else:
                l = (3 * (x1**2) + Ea) * inverse(2 * y1, Ep)
            x3 = (l**2 - x1 - x2) % Ep
```

```

        y3 = (1 * (x1 - x3) - y1) % Ep
        return x3, y3

def multiply(P, n, E):
    Q = P
    R = (0, 0)
    while n > 0:
        if n % 2 == 1:
            R = add(R, Q, E)
        Q = add(Q, Q, E)
        n //= 2
    return R

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) %
block_size))

def encrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(cipher.encrypt(pad(data, 16)))
    print(f"Encrypted file {filein} to file {fileout}")

if __name__ == '__main__':

    # Curve25519 parameters
    a = 486662
    b = 1
    p = 2**255 - 19
    order = 2**252 + 2774231777372353535851937790883648493
    E = {'a': a, 'b': b, 'p': p}

    print("Establishing the TLS handshake...\n")
    secret = randint(1, order - 1)
    encrypt(secret.to_bytes(32, 'big'), "2017-554.pdf", "encrypted.enc")

    while True:
        C = input("Awaiting public key of the client (enter x y):\n")
        try:
            x, y = [int(i) for i in C.strip().split()]
            S = multiply((x, y), bytes_to_long(secret.to_bytes(32, 'big')), E)
            print(f"Shared secret: {S}\n")
        except Exception as e:

```

```
print(f"Error occurred: {e}\n")
```

Solve.py:

```
from sage.all_cmdline import *
from Crypto.Cipher import AES
from pwn import remote, process

def solveDL(E):
    while True:
        try:
            p = E['p']
            a = E['a']
            b = randint(1, p)
            E = EllipticCurve(GF(p), [a, b])
            order = E.order()
            factors = prime_factors(order)

            valid = []
            for factor in factors:
                if factor <= 2**36:
                    valid.append(factor)

            prime = valid[-1]

            G = E.gen(0) * int(order // prime)

            # Here we send G to the server
            tmp_point = G.xy()
            tmp_x, tmp_y = str(tmp_point[0]), str(tmp_point[1])
            tmp_point = tmp_x + " " + tmp_y
            break
        except Exception as e:
            print(e)
            print("Error in generating the curve")
            print("Trying again")
            continue

    message = b"Awaiting public key of the client (enter x y):\n"
    io.sendlineafter(message, tmp_point)

    # We get back P which is G * k
    data = io.recvline()
    print(data.decode().strip())

    if b"Error" in data:
        print("An error on the server occurred")
```

```

        return None, None

P = eval(data[len("Shared secret: "):])

try:
    P = E(P)
    dlog = discrete_log(P, G, operation="+", algorithm="bsgs")
    print(f"DL found: {dlog}")
    print(f"Prime: {prime}")
except Exception as e:
    print(e)
    print("Error in getting the discrete log")
    return None, None

return dlog, prime

def getDLs(E):
    dlogs = []
    primes = []
    total_primes_bit_size = 0
    while total_primes_bit_size < p.bit_length():
        dlog, prime = solveDL(E)
        if dlog and prime:
            dlogs.append(dlog)
            primes.append(prime)
        else:
            print("Error in getting the discrete log")
            continue
        total_primes_bit_size += prime.bit_length()
        print(f"total primes bit size: {total_primes_bit_size}, need {p.bit_length()}")
    return dlogs, primes

def unpad(data, block_size):
    return data[:-data[-1]]

def decrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(unpad(cipher.decrypt(data), 16))
    print(f"Decrypted file {filein} to file {fileout}")

def pwn(E):
    dlogs, primes = getDLs(E)
    print(f"dlogs: {dlogs}")

```

```

print(f"primes: {primes}")
super_secret = CRT_list(dlogs, primes)
return int(super_secret) # convert Integer to int

if __name__ == "__main__":
    # Curve25519 parameters (from the server) usually used for key exchange
    a = 486662
    b = 1
    p = 2**255 - 19
    E = {'a': a, 'b': b, 'p': p}

    io = remote("localhost", 8004)
    # io = process(["python3", "chall.py"]) # local testing

    secret = pwn(E)
    print(f"secret:", secret)
    decrypt(secret.to_bytes(32, 'big'), "encrypted.enc", "decrypted.pdf")

```

Final output:

```

python3 solve.py
[*] Opening connection to localhost on port 8004: Done
/mnt/e/Stuff or sth like that/Crypto/Projects/CryptoProject/EC_cryptographic_attacks/invalid_curve_attack/solve.py:36: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.python.org/3/library/bytes.html
io.sendline(f"{message}, {tmp_point}")
Shared secret: (46092789385786124414252704319684700936753074351361872169756377406876558606272, 40206353277380494561754098101091665475638188093760081605328978113283112974162)
DL found: 2672
Prime: 2857
total primes bit size: 12, need 255
Shared secret: (28209752998392270879829191378012351753494874006600857822685111013591600250312, 1379319129020428516608995855381113194399824004443654015543276507179837642737)
DL found: 41427
Prime: 57143
total primes bit size: 28, need 255
Shared secret: (566432948907676308563299468830485821944574558530794972559054368665628371218, 39750703675935990407370001502253116155608165020126410231440623699687641273499)
DL found: 385370273
Prime: 10904815463
total primes bit size: 62, need 255
Shared secret: (5727080647271954644098798993593414308069873723472534826897881434813150217696, 0)
DL found: 1
Prime: 2
total primes bit size: 64, need 255
Shared secret: (5230729356641320059648975031448002442117093937612700418645978027328103206249, 56871654907574322663959335846810358635974985629512252647901179161162459801260)
DL found: 3861
Prime: 13463
total primes bit size: 78, need 255
Shared secret: (28714419702796687287450337090622896253776803012667904483206700360368557514759, 38187484745087255259470111439937703502682612865717007535299460529690906065987)
DL found: 12
Prime: 17
total primes bit size: 83, need 255
Shared secret: (949256798472344234944974242570335950633970439627067646070838035430654735351, 7325952340802700571622389578826399398049455296455265838158298132853459924)
DL found: 1328
Prime: 4999
total primes bit size: 96, need 255
Shared secret: (43936837532019528604305127742417837815236000860395072091517302851104256172889, 622762409225489774137680795287117239835416492128469792892075430956807675904)
DL found: 2444
Prime: 6633
total primes bit size: 100, need 255
Shared secret: (6928576461766292013279833798769884232239333921535205424917410120178882816372, 16783807165824233159647500710802301247121635830232502442509785245825924535878)
DL found: 65233937
Prime: 76929329
total primes bit size: 136, need 255
Shared secret: (1720827715601280339724299024889805771530688302499123472484296675950696815676, 500173558479793480368878379927151958588739983698798061151911694316316988503)
DL found: 83918
Prime: 93711
total primes bit size: 153, need 255
Shared secret: (11513979478213032240017821935592892171897628687800144708041328615320358282450, 30798033281957839604327671216232278015256687635194722368720759425442436956116)
DL found: 18
Prime: 131
total primes bit size: 161, need 255
Shared secret: (5908048376166119340431906513737184254412876627158270178684069087533513964937, 27660615485678840590334009671930850633667126047119610061295897151401281884349)
DL found: 461213
Prime: 3857023
total primes bit size: 183, need 255
Shared secret: (3838113756336718916810491805815671658711339553890360691563770078124566206775, 41931482224815824132088015050036697478404789028824345480091923819035779067189)
DL found: 888054724
Prime: 1913609783
total primes bit size: 214, need 255
Shared secret: (131796007384296108012832848590476982450948619209458132360267134690716915415, 27831672736380785486627082254538622291765850848360025874139663474505953681636)
DL found: 21262
Prime: 42473
total primes bit size: 230, need 255
Shared secret: (240639457969276268854607390599860760011825551743390098045916893689208893176, 5533571403268162538254342239099295226926613774229696213709187531336322540238)
DL found: 1583
Prime: 10243
total primes bit size: 244, need 255
Shared secret: (3333282748024484504929086842022891880106642418957638201259107490367843819043, 21806853219586539746368566491513508390906160357975116307060162125655161462)
DL found: 1187617
Prime: 3756927
total primes bit size: 266, need 255
dlogs: [2672, 41427, 385370273, 1, 3861, 12, 1328, 2444, 65233937, 83918, 18, 461213, 888054724, 21262, 1583, 1187617]
primes: [2857, 57143, 10904815463, 2, 13463, 17, 4999, 6633, 76929329, 91711, 131, 3857023, 1913609783, 42473, 10243, 3756927]
secret: 366943498177998901857162673605261819426281287896666094393069924981799188929
Decrypted file encrypted.enc to file decrypted.pdf
[*] Closed connection to localhost port 8004

```

## 5.5 Mov attack



This implement of attack creates smooth order primes  $p, q$  and define  $n = p * q$ , so when map two point  $G, Q$  to  $F_{p^k}$ , it will easier to compute discrete logarithm on  $F_{p^k}$ .

Chall.py:

```
from sage.all import *
from Crypto.Util.number import getPrime, isPrime, bytes_to_long
from Crypto.Cipher import AES

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) %
block_size))

def encrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(cipher.encrypt(pad(data,16)))
    print(f"Encrypted file {filein} to file {fileout}")

def getPp1(x, k):
    while True:
        a = 4
        for _ in range(k):
            a *= getPrime(x // k)
        p = a - 1
        if isPrime(p):
            return p

if __name__ == "__main__":
    secret = os.urandom(32) # also the key to encrypt pdf file
    p, q = getPp1(512, 16), getPp1(512, 16)

    assert isPrime(p) and isPrime(q)
    n = p * q
    a, b = matrix(ZZ, [[p, 1], [q, 1]]).solve_right(
        vector([p**2 - p**3, q**2 - q**3])
    )
    E = EllipticCurve(Zmod(n), [a, b])
    G = E(p, p) + E(q, q)
    Q = bytes_to_long(secret) * G

    print(f"{p = }")
    print(f"{q = }")
    print(f"{a = }")
    print(f"{b = }")
```

```

print(f"C =", Q.xy())
print(f'{n = }')
encrypt(secret, "2018-307.pdf", "encrypted.enc")

```

Solve.py:

```

from sage.all import *
from pwn import remote, process
from Crypto.Cipher import AES

def unpad(data, block_size):
    return data[:-data[-1]]

def decrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(unpad(cipher.decrypt(data), 16))
    print(f"Decrypted file {filein} to file {fileout}")

def mov_attack(E, P, G):
    k = 2
    p = E.base_ring().characteristic()
    K = GF(p**k, "a")
    EK = E.base_extend(K)
    PK = EK(P)
    GK = EK(G)
    QK = EK.random_point() # Assuming QK is linear independent to PK
    egqn = PK.tate_pairing(QK, E.order(), k) # e(P,Q)=e(G,Q)^n
    egq = GK.tate_pairing(QK, E.order(), k) # e(G,Q)
    odr = ZZ(pari.fforder(egq, p + 1))
    lg = ZZ(pari.fflog(egqn, egq, odr))
    return lg, odr

if __name__ == "__main__":
    io = remote("localhost", 8005)
    # io = process(["python3", "chall.py"]) # Local testing

    p = int(io.recvlineS().split('=')[1].strip())
    q = int(io.recvlineS().split('=')[1].strip())
    a = int(io.recvlineS().split('=')[1].strip())
    b = int(io.recvlineS().split('=')[1].strip())
    C = eval(io.recvlineS().split('=')[1].strip())
    n = int(io.recvlineS().split('=')[1].strip())

```

```

print(f"{p = }")
print(f"{q = }")
print(f"{a = }")
print(f"{b = }")
print(f"C =", C)
print(f'{n = }')

E = EllipticCurve(Zmod(n), [a, b])
G = E(p, p) + E(q, q)
Ep = E.change_ring(Zmod(p))
Eq = E.change_ring(Zmod(q))

mp, op = mov_attack(Ep, Ep(C), Ep(G))
mq, oq = mov_attack(Eq, Eq(C), Eq(G))
secret = int(crt([mp, mq], [op, oq])) # convert Integer to int

decrypt(int.to_bytes(secret, 32), "encrypted.enc", "decrypted.pdf")

```

Output format:

```

python3 solve.py
[*] Opening connection to localhost on port 8005: Done
p = 19415201800488863904264050518450147687312241301941763028211634241636631389803510562172178488798050643467776358068778951169889342968231231252551367053869
1
q = 17751660617026098531368032947747509917022743082014253628296995073654633613073798167658731879062592118127539371511833613456675163707468038315365400495064
3
a = -1036723588789133792909783163211792934879569527292476309866565367110975866344361029982555048713285462961076212284600078894289077311191065843619385283975
18423209955853000178643050821471929545582751153255167350506359721229862119270551774040334922179533457178247547303811027108387573143539717298014857995445909
b = 12809636185545286810533829638129303208583034435778718230332280707734134622349992356812628425596109419896019062333192931557657198420687589457731020657094
813074538491176475833179720861771292461206504832798853674235327489687087386266909014838838783347655331370327473000081985360879135144104009793555021949816426
610911506918861305539422088856581531724663273111801395044458156026396345261378221595813427838455742351323316330513223102017359983101850231786703885229
C = (2379667912493818533113523906189010225049716489798727278786026382668083739824338131891370904110953516150221656521164650619646669997786152260367750447048
0798032251844843947520052308922615266389077415855348739687343056208671961092006121251418263171323635569655462746738284831610344000748787339029335571564413,
337150875582352737750535362417699838649485664253345426050545257897579280951203654454116129244757002719804174740630023181079123888037678115960387409581224257
6086656241964237845886282180799325482712466427283396659324705798445242364806675995553857306847816122872921640932221409511230683915830625406611715785)
n = 3446520731733523649696037523367768134435671488748869022284634057365483724818565514013364362447123814731661497083803789792171160751977414511548055860732
55903347106783998323862580786532420435744158141470335787125001060073045650141956512530038809735209356600954234621839247735746259211549851601370676828313
Decrypted file encrypted.enc to file decrypted.pdf
[*] Closed connection to localhost port 8005

```

## 5.6 Frey ruck attack

The challenge is generating a non-singular with biggest primes factor is limit to 50-bits prime, also the curve have to have the embedding degree  $k < 6$

Chall.py:

```

from sage.all_cmdline import *
from Crypto.Util.number import *
from Crypto.Cipher import AES

def get_embedding_degree(q, n, max_k):
    for k in range(1, max_k + 1):
        if q ** k % n == 1:
            return k
    return None

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) % block_size))

```



```

def encrypt(key, filein):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open('encrypted.enc', 'wb') as f:
        f.write(cipher.encrypt(pad(data,16)))
    print(f"Encrypted {filein} with key")

def gen_curve():
    while True:
        p = getPrime(128)
        a = p-1
        b = 0
        E = EllipticCurve(GF(p), [a, b])
        # print(prime_factors(E.order()))
        if prime_factors(E.order())[-1] <= 2**50 and 4*a**3 + 27*b**2 != 0:
            for _ in range(10):
                G = E.random_point()
                k = get_embedding_degree(E.base_ring().order(), G.order(), 6)
                if(k != None):
                    return p, a, b, G

if __name__ == "__main__":
    p, a, b, G = gen_curve()
    E = EllipticCurve(GF(p), [a, b])
    key = randint(1, G.order() - 1)
    d = key
    P = G * d

    print(f'{p = }')
    print(f'{a = }')
    print(f'{b = }')
    print(f"G =", G.xy())
    print(f"P =", P.xy())
    encrypt(int.to_bytes(key, 16), "3-540-48910-X_14.pdf")

```

Solve.py:

```

from sage.all import *
from pwn import logging, remote, process
from Crypto.Util.Padding import unpad
from Crypto.Cipher import AES

# from jvdsn
def get_embedding_degree(q, n, max_k):
    """

```

```

    Returns the embedding degree  $k$  of an elliptic curve.
    Note: strictly speaking this function computes the Tate-embedding degree of a curve.
    In almost all cases, the Tate-embedding degree is the same as the Weil-embedding
    degree (also just called the "embedding degree").
    More information: Maas M., "Pairing-Based Cryptography" (Section 5.2)
    :param q: the order of the curve base ring
    :param n: the order of the base point
    :param max_k: the maximum value of embedding degree to try
    :return: the embedding degree  $k$ , or None if it was not found
    """
    for k in range(1, max_k + 1):
        if q ** k % n == 1:
            return k

    return None

def frey_ruck(P, R, max_k=6, max_tries=10):
    """
    Solves the discrete logarithm problem using the Frey-Ruck attack.
    More information: Harasawa R. et al., "Comparing the MOV and FR Reductions in
    Elliptic Curve Cryptography" (Section 3)
    :param P: the base point
    :param R: the point multiplication result
    :param max_k: the maximum value of embedding degree to try (default: 6)
    :param max_tries: the maximum amount of times to try to find  $l$  (default: 10)
    :return:  $l$  such that  $l * P == R$ , or None if  $l$  was not found
    """
    E = P.curve()
    q = E.base_ring().order()
    n = P.order()
    assert gcd(n, q) == 1, "GCD of base point order and curve base ring order should be 1."

    logging.info("Calculating embedding degree...")
    k = get_embedding_degree(q, n, max_k)
    if k is None:
        return None

    logging.info(f"Found embedding degree {k}")
    Ek = E.base_extend(GF(q ** k))
    Pk = Ek(P)
    Rk = Ek(R)
    for _ in range(max_tries):
        S = Ek.random_point()
        T = Ek.random_point()
        if (gamma := Pk.tate_pairing(S, n, k) / Pk.tate_pairing(T, n, k)) == 1:

```

```

        continue

    delta = Rk.tate_pairing(S, n, k) / Rk.tate_pairing(T, n, k)
    logging.info(f"Computing {delta}.log({gamma})...")
    l = delta.log(gamma)
    return int(l)

return None

def unpad(data, block_size):
    return data[:-data[-1]]

def decrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(unpad(cipher.decrypt(data), 16))
    print(f"Decrypted file {filein} to file {fileout}")

if __name__ == "__main__":
    # io = remote("localhost", 8006)
    io = process(["python3", "chall.py"]) # local testing
    try:
        print("Receiving parameters...")
        p = int(io.recvlineS().split('=')[1].strip())
        a = int(io.recvlineS().split('=')[1].strip())
        b = int(io.recvlineS().split('=')[1].strip())
        G = eval(io.recvlineS().split('=')[1].strip())
        P = eval(io.recvlineS().split('=')[1].strip())
        print(io.recvline())

        print(f'{p = }')
        print(f'{a = }')
        print(f'{b = }')
        print(f"G =", G)
        print(f"P =", P)

        E = EllipticCurve(GF(p), [a, b])
        G = E(G)
        P = E(P)
        print("G order:", G.order())

        d = frey_ruck(G, P)
        assert G*d == P, "wrong d!" # check if d is correct, have chance to fail
        print("d:", d)

```

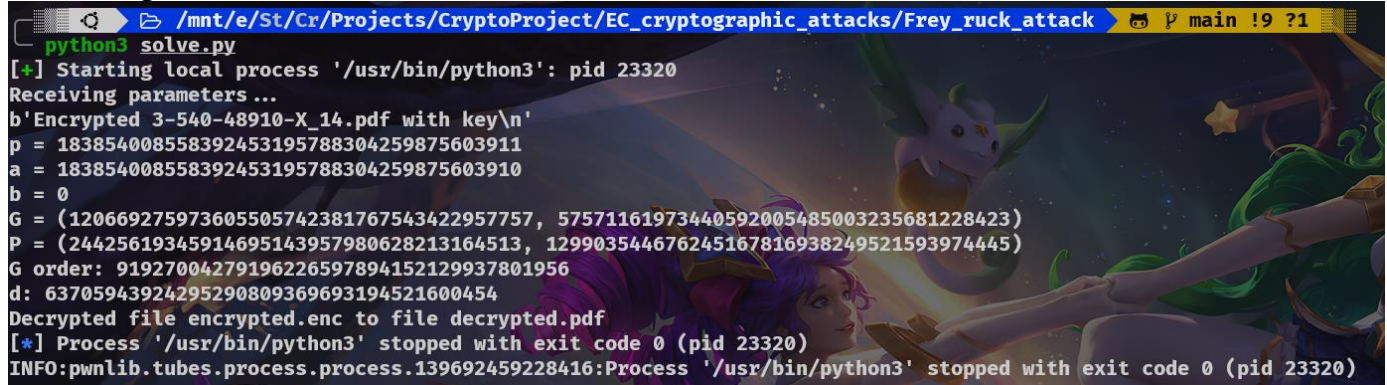
```

        decrypt(int.to_bytes(d, 16), "encrypted.enc", "decrypted.pdf")

except Exception as e:
    print(e)
    io.close()

```

Final output:



```

python3 solve.py
[+] Starting local process '/usr/bin/python3': pid 23320
Receiving parameters ...
b'Encrypted 3-540-48910-X_14.pdf with key\n'
p = 183854008558392453195788304259875603911
a = 183854008558392453195788304259875603910
b = 0
G = (120669275973605505742381767543422957757, 57571161973440592005485003235681228423)
P = (24425619345914695143957980628213164513, 129903544676245167816938249521593974445)
G order: 91927004279196226597894152129937801956
d: 63705943924295290809369693194521600454
Decrypted file encrypted.enc to file decrypted.pdf
[*] Process '/usr/bin/python3' stopped with exit code 0 (pid 23320)
INFO:pwnlib.tubes.process.process.139692459228416:Process '/usr/bin/python3' stopped with exit code 0 (pid 23320)

```

## 5.7 Smart attack

The server side create an anomalous curve each time, making it easier to apply the smart attack.

Chall.py:

```

from sage.all_cmdline import *
from random import randint
from Crypto.Cipher import AES
import json

# params from http://www.monnerat.info/publications/anomalous.pdf
D = 11
j = -2**15

def anom_curve():
    m = 257743850762632419871495
    p = (11*m*(m + 1)) + 3
    a = (-3*j * inverse_mod((j - 1728), p)) % p
    b = (2*j * inverse_mod((j - 1728), p)) % p
    E = EllipticCurve(GF(p), [a,b])
    return p, a, b, E

def smart_attack():
    # with open("smarts_attack_curves.json", 'r') as f:
    #     curves = json.loads(f.read())
    # index = randint(0, len(curves)-1)
    # p = int(curves[index]['field']['p'], 16)
    # a = int(curves[index]['a'], 16)
    # b = int(curves[index]['b'], 16)

```

```

# E = EllipticCurve(GF(p), [a, b])
p, a, b, E = anom_curve()
print("The curve parameters are:")
print("p = "+str(p))
print("a = "+str(a))
print("b = "+str(b))
P1 = E.gens()[0]
print('\nP1: '+str(P1.xy()))
secret = randint(1, E.order() - 1)
P2 = secret * P1
print('P2: '+str(P2.xy()))
print('P2 = secret * P1')
return secret

def pad(data, block_size):
    return data + bytes([block_size - len(data) % block_size] * (block_size - len(data) %
block_size))

def encrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(cipher.encrypt(pad(data,16)))
    print(f"Encrypted file {filein} to file {fileout}")

def main():
    print("The answer will be a randomly generated solution and hence not an obvious
message.")
    print("Are you smart enough to crack this?\n")
    smarts = smarts_attack()
    encrypt(smarts.to_bytes(32)[:16], "s001459900052.pdf", "encrypted.enc")
    while True:
        print("\nWhat is the value of 'secret'? : ")
        n = int(input(""))
        if n == smarts:
            print("Success!")
            break
        else:
            print("Please try again!")

if __name__ == "__main__":
    main()

```

Solve.py:

```

from sage.all_cmdline import *
from pwn import remote, process

```

```

from Crypto.Cipher import AES

def SmartAttack(P,Q,p):
    E = P.curve()
    Eqp = EllipticCurve(Qp(p, 2), [ ZZ(t) + randint(0,p)*p for t in E.a_invariants() ])

    P_Qps = Eqp.lift_x(ZZ(P.xy()[0]), all=True)
    for P_Qp in P_Qps:
        if GF(p)(P_Qp.xy()[1]) == P.xy()[1]:
            break

    Q_Qps = Eqp.lift_x(ZZ(Q.xy()[0]), all=True)
    for Q_Qp in Q_Qps:
        if GF(p)(Q_Qp.xy()[1]) == Q.xy()[1]:
            break

    p_times_P = p*P_Qp
    p_times_Q = p*Q_Qp

    x_P,y_P = p_times_P.xy()
    x_Q,y_Q = p_times_Q.xy()

    phi_P = -(x_P/y_P)
    phi_Q = -(x_Q/y_Q)
    k = phi_Q/phi_P
    return ZZ(k)

def unpad(data, block_size):
    return data[:-data[-1]]

def decrypt(key, filein, fileout):
    with open(filein, 'rb') as f:
        data = f.read()
    cipher = AES.new(key, AES.MODE_ECB)
    with open(fileout, 'wb') as f:
        f.write(unpad(cipher.decrypt(data),16))
    print(f"Decrypted file {filein} to file {fileout}")

if __name__ == '__main__':
    io = remote('localhost', 8007)
    # io = process(['python3', 'chall.py']) # Local testing
    io.recvuntil(b'p = ')
    p = int(io.recvline().strip())
    io.recvuntil(b'a = ')
    a = int(io.recvline().strip())
    io.recvuntil(b'b = ')

```

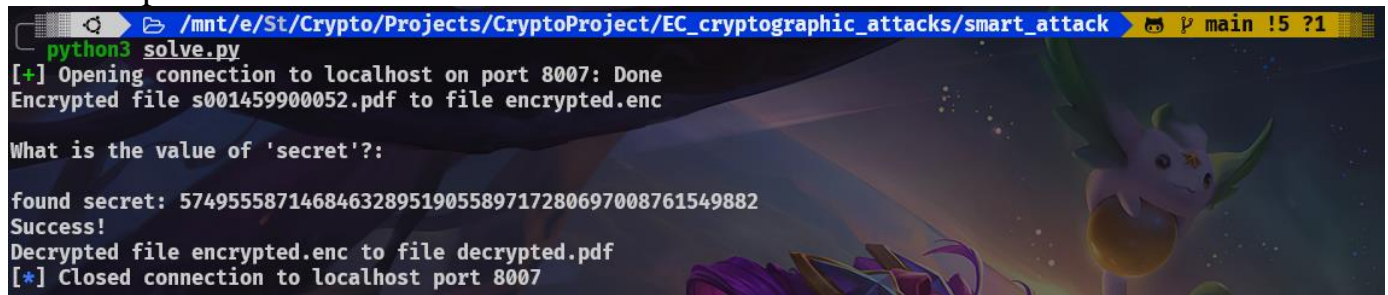
```

b = int(io.recvline().strip())
io.recvuntil(b'P1: ')
P = eval(io.recvline().strip())
io.recvuntil(b'P2: ')
Q = eval(io.recvline().strip())
io.recvuntil(b'P2 = secret * P1\n')
P = EllipticCurve(GF(p), [a, b])(P)
Q = EllipticCurve(GF(p), [a, b])(Q)

print(io.recvuntil(b"What is the value of 'secret'?: \n").decode())
secret = SmartAttack(P,Q,p)
print("found secret:", secret)
io.sendline(str(secret).encode())
result = io.recvline().strip().decode()
print(result)
if result == 'Success!':
    decrypt(secret.to_bytes(32)[:16], 'encrypted.enc', 'decrypted.pdf')
else:
    print('Failed!')
io.close()

```

Final output:



```

python3 solve.py
[+] Opening connection to localhost on port 8007: Done
Encrypted file s001459900052.pdf to file encrypted.enc

What is the value of 'secret'?:

found secret: 574955587146846328951905589717280697008761549882
Success!
Decrypted file encrypted.enc to file decrypted.pdf
[*] Closed connection to localhost port 8007

```

## 6. 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.
- We strongly recommend using some standard curve like Curve25519, Curve383187, M383, ... when implementing ECC in some fields like key exchange or digital signature.



## 7. References

- Silverman, J. H., Piper, 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>)
- Harasawa, R., Shikata, J., Suzuki, J., & Imai, H. (1999). Comparing the MOV and FR reductions in elliptic curve cryptography. In *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings* 18 (pp. 190-205). Springer Berlin Heidelberg.  
([https://link.springer.com/chapter/10.1007/3-540-48910-X\\_14](https://link.springer.com/chapter/10.1007/3-540-48910-X_14))
- Galbraith, S. D., Wang, P., & Zhang, F. (2015). Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm. *Cryptology ePrint Archive*.  
(<https://eprint.iacr.org/2015/605.pdf>)
- Smart, N. P. (1999). The discrete logarithm problem on elliptic curves of trace one. *Journal of cryptology*, 12, 193-196.  
(<https://link.springer.com/article/10.1007/s001459900052>)
- Sommerseth, M. L., & Hoeiland, H. (2015). *Pohlig-hellman applied in elliptic curve cryptography*. Technical Report, University of California Santa Barbara. 2015.  
(<http://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Sommerseth+Hoeiland.pdf>)
- Leprévost, F., Monnerat, J., Varrette, S., & Vaudenay, S. (2005). Generating anomalous elliptic curves. *Information processing letters*, 93(5), 225-230.  
(<https://www.sciencedirect.com/science/article/abs/pii/S0020019004003527>)