

University of Information Technology

Faculty of Computer Network and Communications



UIT

FINAL REPORT

Subject : Cryptography

Class : NT219.O11.ANTN

Lecturer : Nguyen Ngoc Tu

Cryptanalysis on ECC-based Algorithms

Thai Vinh Dat – 22520235

Dinh Le Thanh Cong - 22520167

Table of Contents

1. Contents

1.Contents	2
2.Overview	3
2.1 Elliptic curve:.....	3
2.2 Elliptic Curves over Finite Fields.....	8
2.3 The Elliptic Curve Discrete Logarithm Problem (ECDLP).....	10
2.3.1 The Double-and-Add Algorithm.....	10
2.3.2 How hard is the ECDLP?	11
3.Key generation in ECC.....	12
4.Some attack models in ECC.....	13
4.1 Baby-step giant-step	13
4.2 Polard's rho attack.....	14
4.3 Pohlig – Hellman attack.....	16
4.4 Smart attack where $\#E(F_p) = p$.....	16
4.5 Invalid curve attack.....	17
5.Implementation and testing	18
5.1 Baby-step giant-step (BSGS)	18
5.2 Polard-rho attack.....	20
5.3 Pohlig-Hellman attack.....	24
5.4 Smart attack.....	25
5.5 Invalid curve attack.....	27
6.Deployment:.....	33
7.References	34

2. Overview

- Elliptic Curve Cryptography (ECC) is a type of public key cryptography that uses the mathematics behind elliptic curves to provide strong security with relatively small key sizes. This makes ECC more efficient than other methods, which is particularly beneficial in systems with limited computational resources. ECC is used in a variety of fields and applications:
 - o Internet of Things (IoT): ECC's efficiency makes it ideal for IoT devices, which often have limited computational power. It provides strong security without overburdening the device's resources.
 - o Blockchain Technology: ECC is fundamental to blockchain technologies like Bitcoin. It's used to generate the public-private key pairs that secure transactions on the blockchain.
 - o Wireless Security: Protocols like Zigbee, used in smart home devices, employ ECC to secure communications.
 - o Secure Web Browsing: ECC is used in security protocols like Transport Layer Security (TLS) and Secure Sockets Layer (SSL) to secure web traffic.
 - o Cloud Computing: Cloud service providers use ECC to ensure the privacy and security of their users' data.
 - o Mobile Applications: Many mobile apps use ECC to secure communications, protecting user data and privacy.
- In summary, ECC is a powerful and efficient cryptographic method that's widely used in many areas of technology. Its ability to provide strong security with small key sizes makes it particularly valuable in our increasingly connected world. 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*).

$$E1: Y^2 = X^3 - 3X + 3 \quad \text{and} \quad E2: Y^2 = X^3 - 6X + 5$$

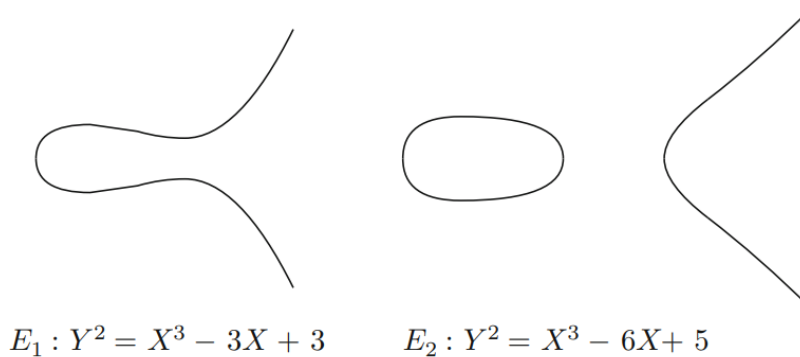


Figure 1

- There are some other forms of Elliptic curve, such as:
 - Montgomery form: $By^2 = x^3 + Ax^2 + x$
 - Twisted Edwards form: $ax^2 + y^2 = 1 + dx^2y^2$
 - Hessian form: $x^3 + y^3 + 1 = 3Dxy$, $D^3 - 1 \neq 0$
 - Koblitz form: $y^2 + xy = x^3 + ax^2 + b$
 - ...

*** Point addition

- An amazing feature 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. We put quotation marks around “add” because we are referring to an operation that combines two points in a manner analogous to addition in some respects (it is commutative and associative, and there is an identity), but very unlike addition in other ways. The most natural way to describe the “addition law” on elliptic curves is to use geometry.
- Let P and Q be two point on an elliptic curve E, as illustrated in Figure 6.2. We draw the line L through P and Q. This line intersects E at thre 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:

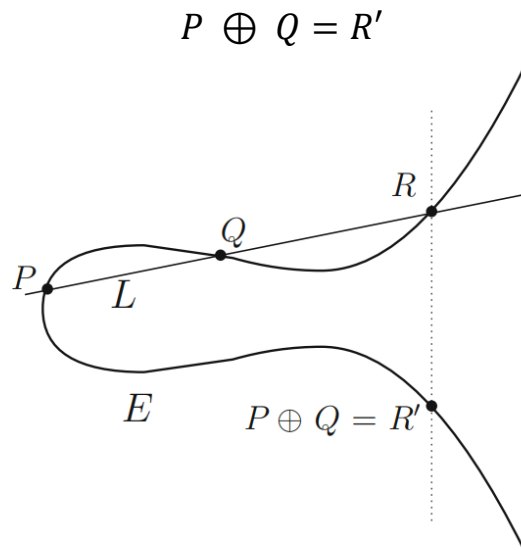


Figure 2: The addition law on an elliptic curve

- 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.

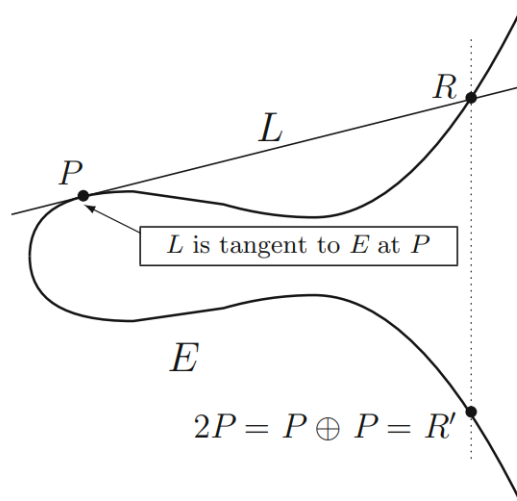


Figure 3: Adding a point P to itself

- 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:

$$P + P' = O$$

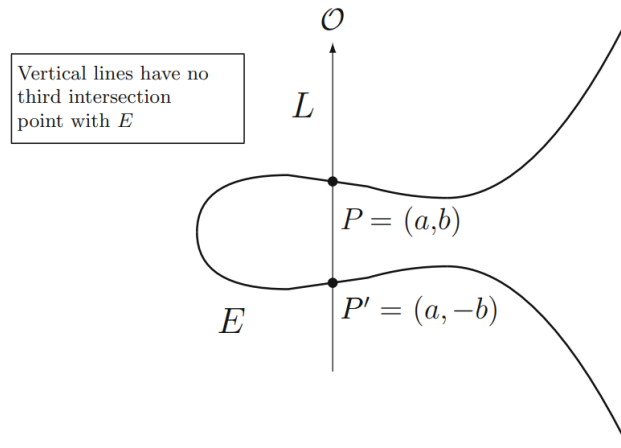
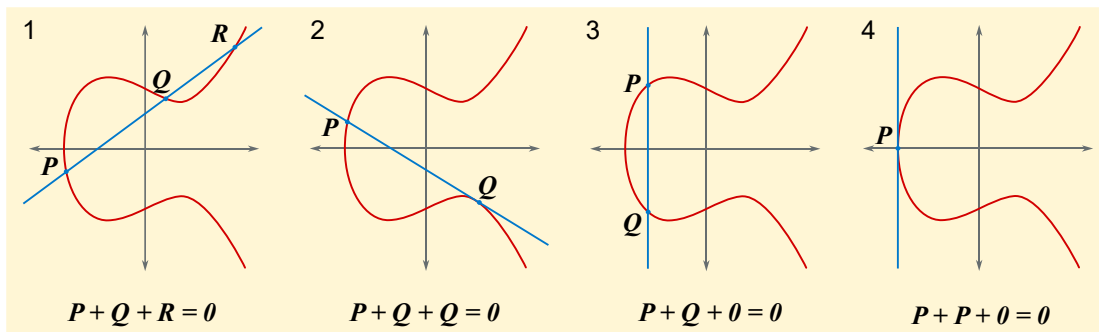


Figure 4: The vertical line L through $P = (a, b)$ and $P' = (a, -b)$

- 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, $P \oplus O = P$, 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 .

- If $P_1 = O$, then $P_1 + P_2 = P_2$
- Otherwise, if $P_2 = O$, then $P_1 + P_2 = P_1$
- Otherwise, write $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$
- If $x_1 = x_2$ and $y_1 = -y_2$, then $P_1 + P_2 = O$
- Otherwise, define λ 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

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

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,

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

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

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 = 64 - 9 - 1 = 54 = 2, \\ y_3 &= -(\lambda x_3 + v) = -8 * 2 = -16 = 10. \end{aligned}$$

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 a fundamental process that involves generating pairs of cryptographic keys—a private key and a corresponding public key. Here is step-by-step of key generation in ECC:
 - Selecting parameters: The first step in key generation is to select the parameters for the elliptic curve. These parameters include the equation defining the curve, the base point on the curve, and the order of the base point. 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 curves from <https://www.secg.org/sec2-v2.pdf>, which was tested and ensured to be safe.
 - Generating the Private Key: The private key in ECC is a randomly selected integer within a specific range (lower than the order of base point). 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.

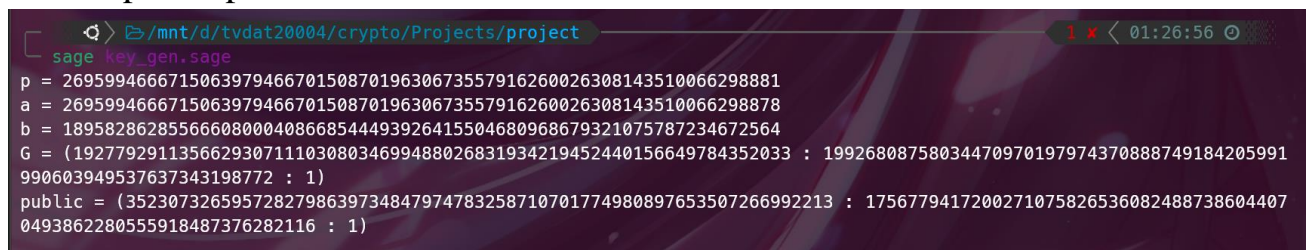
- Computing the Public Key: The public key is derived from the private key and the chosen elliptic curve parameters. It is computed by performing scalar multiplication of the base point on the curve with the private key. This process involves adding the base point to itself multiple times according to the binary representation of the private key.
- Example code for key generation using Sagemath:

```

1. # Step 1: Choose parameter from P-224 curve
2. p = 0xffffffffffffffffffffffffffffffff000000000000000000000001
3. K = GF(p)
4. a = K(0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffe)
5. b = K(0xb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4)
6. E = EllipticCurve(K, (a, b))
7. G = E(0xb70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21, 0xbd376388b5f723fb
4c22dfe6cd4375a05a07476444d5819985007e34)
8. # Step 2: generate private key
9. import random
10. priv = random.randint(1, G.order() - 1)
11. # Step 3: Compute public key
12. public = priv * G
13.
14. # Print public key and all parameters
15. print(f'{p = }')
16. print(f'{a = }')
17. print(f'{b = }')
18. print(f'{G = }')
19. print(f'{public = }')

```

- Sample output:



```

sage key_gen.sage
p = 26959946667150639794667015087019630673557916260026308143510066298881
a = 26959946667150639794667015087019630673557916260026308143510066298878
b = 18958286285566608000408668544493926415504680968679321075787234672564
G = (19277929113566293071110308034699488026831934219452440156649784352033 : 19926808758034470970197974370888749184205991
990603949537637343198772 : 1)
public = (3523073265957282798639734847974783258710701774980897653507266992213 : 1756779417200271075826536082488738604407
0493862280555918487376282116 : 1)

```

4. Some attack models in ECC

4.1 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 F_p 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 $m = \lceil \sqrt{p} \rceil$
 - Using Euclid theorem, we write: $k = im + j$ for $i, j \in \{0, 1, 2, \dots, m-1\}$. 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 $j = 0, 1, 2, \dots, m-1$ and store the results in table.
 - We start computing

$$Q - [im]P \text{ for each } j = 0, 1, \dots, m-1$$
 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.

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 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 $O(\sqrt{n})$ 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 $G = E(F_p)$, such that $|G| = n$, and P and Q such that $Q = x * P$ in G , our aim is to calculate x . With Pollard's rho, we will solve a slightly 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 :

$$x = \frac{a - A}{B - b} \pmod{n}$$

Algorithm: Polard's Rho algorithm

1. Using hash function, we partition G into 3 sets, S_1, S_2, S_3 of roughly the same size, but $O \notin S_2$
2. Define an iterating function f of a random walk:

$$R_{i+1} = f(R_i) = \begin{cases} Q + R_i, & R_i \in S_1 \\ 2R_i, & R_i \in S_2 \\ P + R_i, & R_i \in S_3 \end{cases} \quad (1)$$

3. Let $R_i = a_iP + b_iQ$, and therefore

$$a_{i+1} = \begin{cases} a_i, & R_i \in S_1 \\ 2a_i, & R_i \in S_2 \\ a_i + 1, & R_i \in S_3 \end{cases} \quad (2)$$

and

$$b_{i+1} = \begin{cases} b_i + 1, & R_i \in S_1 \\ 2b_i, & R_i \in S_2 \\ b_i, & R_i \in S_3 \end{cases} \quad (3)$$

4. Start with $R_0 = P, a_0 = 1, b_0 = 0$ and generate pairs (R_i, R_{2i}) until a match is found, in example,

$$R_m = R_{2m} \text{ for some } m$$

Once we found a match, we have

$$R_m = a_mP + b_mQ$$

$$R_{2m} = a_{2m}P + b_{2m}Q$$

Hence we compute x to be:

$$x = \frac{a_{2m} - a_m}{b_m - b_{2m}} \pmod{n} \quad (4)$$

- Assuming that the random walk (4.1) defined in the algorithm produces random terms, the algorithm solves the elliptic curve discrete logarithm problem in $O(\sqrt{n})$ operations.
- To be able to calculate x , the denominator in (4) has to be invertible in \mathbb{Z}_n , where n is the group order of $E(F_p)$. If the $\gcd(b_m - b_{2m}, n) > 1$, the inverse

of $(b_m - b_{2m})$ 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, F_p and the point P are chosen such that $\#E(F_p) = n$ 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.

4.3 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 used 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:

$$N = \prod_{i=1}^k q_i^{e_i}$$

- Then starting with q_1 , we consider $q_1^{e_1} | N$ and we compute

$$Q_0 = \left\lfloor \frac{N}{q_1^{e_1}} \right\rfloor Q \text{ and } P_0 = \left\lfloor \frac{N}{q_1^{e_1}} \right\rfloor P$$

By using BSGS to solve DLP, we can find an integer k_0 such that

$$Q_0 = k_0 P_0 \text{ mod}(q_1^{e_1})$$

- Now we can find $k \pmod{q_i^{e_i}}$ for each i then use the [Chinese Remainder theorem](#) (CRT) to combine them and then obtain $k \pmod{N}$.

4.4 Smart attack where $\#E(F_p) = p$

- 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 $\#E(F_p) = p$ (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 F_p such that $\#E(F_p) = p$, or in other words $t_p = p + 1 - \#E(F_p) = 1$ (the quantity t_p 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 subgroup with order of p , then ECDLP problem can be solved in $O(1)$ time.
- Now, given arbitrary curve E over a finite field size p (F_p) with $\#E(F_p) = p$, and point $Q = d * P$, find d ?
 - o First step, we try to lift these points to $E(Q_p)$ (elliptic curve on p -adic field) using [Hessel's lift](#) 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 Q_p . We know that $Q = kP$ in $E(F_p)$ so thus in the kernel of that homomorphism.

$$Q' - kP' \in E_1(Q_p)$$

- o 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 \mathcal{O} in $E(F_p)$. So multiply through by p and we get

$$pQ' - k(pP') \in E_2(Q_p)$$

with $pQ' \in E_1(Q_p)$ and $pP' \in E(Q_p)$. We can now apply the p -adic elliptic log to get

$$\psi_p(pQ') - k\psi_p(pP') \in p\mathbb{Z}_p$$

and thus

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

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

4.5 Invalid curve attack

- Invalid Curve Attack relies 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 $\lambda = \frac{3x^2+a}{2y}$ $x_2 = \lambda^2 - 2x$ $y_2 = -\lambda^3 + 3\lambda x - y$	$\lambda = \frac{y_1-y_2}{x_1-x_2}$ $x_3 = \lambda^2 - x_1 - x_2$ $y_3 = -\lambda^3 + 2\lambda x_1 + \lambda x_2 - y_1$

- Thus, if a point is not checked to be on the curve, the attacker can send a point which lie on the curve $E'(\mathbb{F}_p)$ of equation $y^2 = x^3 + ax + b_1$, 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 :
 - $y^2 = x^3 + ax + b \pmod p$
 - $y^2 = x^3 + ax + b_1 \pmod p$
 - $y^2 = x^3 + ax + b_2 \pmod p$
 - $y^2 = x^3 + ax + b_3 \pmod p$
- 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 .

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)

- We get the following curve
- $a = 196220977$
- $b = 687298370$
- $p = 845337859$
- $P = (523283288, 42887324)$
- $Q = (394977470, 739173023)$


```
sage.groups.generic.discrete_log(a, base, ord=None, bounds=None, operation='**',
    identity=None, inverse=None, op=None, algorithm='bsgs')
```

Totally generic discrete log function.

INPUT:

- `a` – group element
- `base` – group element (the base)
- `ord` – integer (multiple of order of base, or `None`)
- `bounds` – a priori bounds on the log
- `operation` – string: `'**'`, `'+'`, other
- `identity` – the group's identity
- `inverse()` – function of 1 argument `x`, returning inverse of `x`
- `op()` – function of 2 arguments `x`, `y`, returning `x*y` in the group
- `algorithm` – string denoting what algorithm to use for prime-order logarithms: `'bsgs'`, `'rho'`, `'lambda'`

- Let's use it to solve the above problem:

```
d = discrete_log(Q,P, operation="+")
```

- Output:

```
dat at hno3 in [~/Documents/crypto/project/BSGS]
23:35:56 > python3 tool.py
The solution for ECDLP is 734724895
```

- Both get the same result.

5.2 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 *
2. from Crypto.Util.number import *
3. import random
4. import secrets
5. from Crypto.Cipher import AES
6. from Crypto.Util.Padding import pad
7. from hashlib import sha3_512 # most secure hash I've heard :v
8.
9. def check(prime):
10.     if not isPrime(prime):
11.         print("Not a prime!!!")
12.         return False
13.     if prime <= (1>>40):
```

```

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 Pollard's rho to attack because the Pollard's rho take $O(\sqrt{n})$ step ($\approx 2^{20}$, which is nearing towards the "long" end of computing times, but is still feasible for our purposes). So let's use the Pollard-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
2. from hashlib import sha3_512
3. ciphertext = open("cipher.enc", 'rb').read()
4.
5. key = sha3_512(str(x).encode()).digest()[:16]
6. iv = ciphertext[:16]
7. ciphertext = ciphertext[16:]
8. cipher = AES.new(key, AES.MODE_CBC, iv)
9. with open("recovered.pdf", 'wb') as write:
10.     write.write(unpad(cipher.decrypt(ciphertext),16))
11. print("Generate recovered.pdf successfully!!!")

```

```

[+] Starting local process '/usr/bin/python3': pid 12018
Found secret!!!, x = 393800342728
Generate recovered.pdf successfully!!!
[*] Process '/usr/bin/python3' stopped with exit code 0 (pid 12018)

```

- Sagemath also has a build-in function `discrete_log_rho` that can solve ECDLP using Pollard's rho algorithm.

```

sage.groups.generic.discrete_log_rho(a, base, ord=None, operation='*', identity=None,
                                     inverse=None, op=None, hash_function=<built-in function hash>)

```

Pollard Rho algorithm for computing discrete logarithm in cyclic group of prime order. If the group order is very small it falls back to the baby step giant step algorithm.

INPUT:

- `a` – a group element
- `base` – a group element
- `ord` – the order of `base` or `None`, in this case we try to compute it
- `operation` – a string (default: `'*'`) denoting whether we are in an additive group or a multiplicative one
- `identity` – the group's identity
- `inverse()` – function of 1 argument `x`, returning inverse of `x`
- `op()` – function of 2 arguments `x`, `y`, returning `x*y` in the group
- `hash_function` – having an efficient hash function is critical for this algorithm (see examples)

OUTPUT: an integer n such that $a = base^n$ (or $a = n * base$)

ALGORITHM: Pollard rho for discrete logarithm, adapted from the article of Edlyn Teske, 'A space efficient algorithm for group structure computation'.

- Let's use it to attack
- ```

d = discrete_log_rho(Q,P, operation="+")

```

```

23 n = E.order()
dat at hno3 in [~/Documents/crypto/project/polard_rho_attack/attack]
0:02:17 > python3 tool.py
[+] Opening connection to localhost on port 6002: Done
38538238461 try:
Generate recovered.pdf successfully!!!, P, operation="+")
[*] Closed connection to localhost port 6002

```

## 5.3 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"factored: {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:

```
dat at hno3 in [~/Documents/crypto/project/Pohlig-hellman]
11:16:42 > python3 solve.py
factors: 2^2 * 5 * 7 * 13 * 617 * 4759 * 8260807 * 144722299 * 928070153503 * 2590206143359 * 13389517011401 * 56307812771039
x = 2 mod(4)
x = 0 mod(5)
x = 4 mod(7)
x = 7 mod(13)
x = 606 mod(617)
x = 3671 mod(4759)
x = 7405985 mod(8260807)
x = 39023084 mod(144722299)
x = 152594204597 mod(928070153503)
x = 1993486725824 mod(2590206143359)
x = 10543700931599 mod(13389517011401)
x = 46327739947315 mod(56307812771039)
Calculating in CRT...
=> x = 4249406938295033803194153142956196601485465109243425432452524823075512698870 mod (1157920892103562487626974469494075735298
3406581957583176174383474827192619340)
```

- 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:

```
dat at hno3 in [~/Documents/crypto/project/Pohlig-hellman]
11:37:25 > python3 tool.py
4249406938295033803194153142956196601485465109243425432452524823075512698870
```

And the same result is given.

## 5.4 Smart attack

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

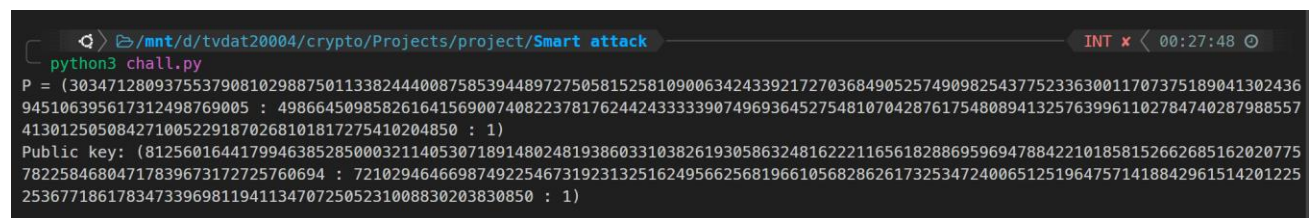
```
1. from sage.all import *
2. import random
3. from hashlib import sha1
4. from Crypto.Cipher import AES
5. from Crypto.Util.Padding import pad
6. # Curve params
7. p = 0xa15c4fb663a578d8b2496d3151a946119ee42695e18e13e90600192b1d0abdbb6f787f90c8d102
 ff88e284dd4526f5f6b6c980bf88f1d0490714b67e8a2a2b77
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())
14. # open file to get plaintext
15. with open("input.pdf", "rb") as f:
16. pt = f.read()
17.
18. def encrypt(key):
19. key = sha1(str(key).encode()).digest()[:16]
20. iv = random.randbytes(16)
21. cipher = AES.new(key, AES.MODE_CBC, iv)
22. return iv + cipher.encrypt(pad(pt, 16))
23.
24. P = E.gen(0)
25. n = random.randint(1, P.order() - 1)
26. print(f'{P = }')
27. print(f'Public key: {P*n}')
28. # write ciphertext to cipher.enc
29. encrypted = encrypt(n)
30. with open("cipher.enc", "wb") as cipher:
31. cipher.write(encrypted)

```

The server give us the output:



```

python3 chall.py
P = (3034712809375537908102988750113382444008758539448972750581525810900634243392172703684905257490982543775233630011707375189041302436945106395617312498769005 : 498664509858261641569007408223781762442433339074969364527548107042876175480894132576399611027847402879885574130125050842710052291870268101817275410204850 : 1)
Public key: (8125601644179946385285000321140530718914802481938603310382619305863248162221165618288695969478842210185815266268516202077578225846804717839673172725760694 : 7210294646698749225467319231325162495662568196610568286261732534724006512519647571418842961514201225253677186178347339698119411347072505231008830203830850 : 1)

```

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 = 0xa15c4fb663a578d8b2496d3151a946119ee42695e18e13e90600192b1d0abdbb6f787f90c8d102ff88e284dd4526f5f6b6c980bf88f1d0490714b67e8a2a2b77
6. a = 0x5e009506fcc7eff573bc960d88638fe25e76a9b6c7caeea072a27dcd1fa46abb15b7b6210cf90caba982893ee2779669bac06e267013486b22ff3e24abae2d42
7. b = 0x2ce7d1ca4493b0977f088f6d30d9241f8048fdea112cc385b793bce953998caae680864a7d3aa437ea3ffd1441ca3fb352b0b710bb3f053e980e503be9a7fece
8. E = EllipticCurve(GF(p), [a,b])
9.
10. P = E(3034712809375537908102988750113382444008758539448972750581525810900634243392172703684905257490982543775233630011707375189041302436945106395617312498769005, 498664509858261641569007408223781762442433339074969364527548107042876175480894132576399611027847402879885574130125050842710052291870268101817275410204850)
11. Q = E(8125601644179946385285000321140530718914802481938603310382619305863248162221165618288695969478842210185815266268516202077578225846804717839673172725760694, 7210294646698749225467319231325162495662568196610568286261732534724006512519647571418842961514201225253677186178347339698119411347072505231008830203830850)
12. enc = open("cipher.enc", "rb").read()

```

```

13.
14. # Lifts a point to the p-adic field.
15. def _lift(E, P, gf):
16. x, y = map(ZZ, P.xy())
17. for point_ in E.lift_x(x, all=True):
18. _, y_ = map(gf, point_.xy())
19. if y == y_:
20. return point_
21. def attack(G, P):
22. """
23. Solves the discrete logarithm problem using Smart's attack.
24. More information: Smart N. P., "The discrete logarithm problem on elliptic curve
25. s of trace one"
26. :param G: the base point
27. :param P: the point multiplication result
28. :return: l such that l * G == P
29. """
30. E = G.curve()
31. gf = E.base_ring()
32. p = gf.order()
33. assert E.trace_of_frobenius() == 1, f"Curve should have trace of Frobenius = 1."
34. E = EllipticCurve(Qp(p), [int(a) + p * ZZ.random_element(1, p) for a in E.a_invariants()])
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.
41. secret = attack(P,Q)
42. assert P*secret == Q
43. print(secret)
44. key = sha1(str(secret).encode()).digest()[:16]
45. iv, ct = enc[:16], enc[16:]
46. cipher = AES.new(key, AES.MODE_CBC, iv)
47. with open("recovered.pdf", "wb") as file:
48. file.write(unpad(cipher.decrypt(ct),16))
49. print("Write to recovered.pdf successfully!!!")

```

```

python3 solve.py
400340006326103370146826000596776818027752721260316720577253266091185646839562852163154040259134000288949148151932850122480912064832566
9279871433135295349
Write to recovered.pdf successfully!!!

```

## 5.5 Invalid curve attack

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

```

- class Curve:
- def __init__(self, p, a, b):
- self.p = p
- self.a = a
- self.b = b
-
- def __eq__(self, other):
- if isinstance(other, Curve):
- return self.p == other.p and self.a == other.a and self.b == other.b
- return None
-

```

```

- 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 = (1 * 1 - self.x - other.x) % p
- y3 = (1 * (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 = (1 * l - 2 * self.x) % p
- y3 = (1 * (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 = 11579208921035624876269744694940757352999695522413576034242259061068512044369
- 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:
 

```

- p = 2**256 - 2**224 + 2**192 + 2**96 - 1
- F = GF(p)
- a = -3
- n = 11579208921035624876269744694940757352999695522413576034242259061068512044369

```

```

- 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  $dQ_i = P_i$ .

```

- 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](#).

## 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. If the order is small, the curve can be attacked by Pollard's rho or BSGS. If the order is smooth, it can be attacked by Pohlig – Hellman. If the order of the curve is equal to order of the field, it can be attacked by Smart attack, ... So we can use standard curves, which was ensured to be safe for almost attack from weak curves. <https://www.secg.org/sec2-v2.pdf>
- 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.

## 7. References

- Novotney, P. (2010). Weak Curves In Elliptic Curve Cryptography. modular. math. washington. edu/edu/2010/414/projects/novotney. pdf.  
(<https://wstein.org/edu/2010/414/projects/novotney.pdf>)
- Smart, N. P. (1999). The discrete logarithm problem on elliptic curves of trace one. Journal of cryptology, 12, 193-196.  
(<https://www.hpl.hp.com/techreports/97/HPL-97-128.pdf>)
- 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>)
- Qu, M. (1999). Sec 2: Recommended elliptic curve domain parameters. Certicom Res., Mississauga, ON, Canada, Tech. Rep. SEC2-Ver-0.6.  
(<https://www.secg.org/sec2-v2.pdf>)
- Seet, M. Z., Franklin, J., & Brown, P. (2007). Elliptic Curve Cryptography: Improving the Pollard-Rho Algorithm. The University of New South Wales.  
(<https://www.maths.unsw.edu.au/sites/default/files/mandyseetthesis.pdf> )
- Antipa, A., Brown, D., Menezes, A., Struik, R., & Vanstone, S. (2002, December). Validation of elliptic curve public keys. In *International workshop on public key cryptography* (pp. 211-223). Berlin, Heidelberg: Springer Berlin Heidelberg.  
(<https://www.iacr.org/archive/pkc2003/25670211/25670211.pdf>)
- Jager, T., Schwenk, J., & Somorovsky, J. (2015). Practical invalid curve attacks on TLS-ECDH. In Computer Security--ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I 20 (pp. 407-425). Springer International Publishing.  
(<https://www.nds.ruhr-uni-bochum.de/media/nds/veroeffentlichungen/2015/09/14/main-full.pdf>)