

UNIVERSITY OF INFORMATION TECHNOLOGY

FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS



SUBJECT: CRYPTOGRAPHY

FINAL REPORT

CRYPTANALYSIS

ON ECC-BASED ALGORITHMS

LECTURER:

Dr. NGUYEN NGOC TU

REPORTED BY:

VO NGUYEN THAI HOC - 22520489

NGUYEN VIET HOANG - 22520471

NGO HONG PHUC - 22521124

HO CHI MINH CITY, 2024

TABLE OF CONTENTS

I. OVERVIEW	1
1.1 Defining ECC	4
1.2 Elliptic Curve	5
1.3 Elliptic Curve over Finite Fields	8
1.4 The Elliptic Curve Discrete Logarithm Problem (ECDLP)	9
1.4.1 The Double-and-Add Algorithm	9
1.4.2 How hard is the ECDLP?	9
II. KEY GENERATION IN ECC	10
III. SOME ATTACK MODELS IN ECC	11
3.1 Smart attack	11
3.2 Pollard's Rho attack	12
3.3 Invalid Curve Attack	13
3.4 MOV Attack	14
3.5 Frey Ruck Attack	16
IV. IMPLEMENTATION AND TESTING	17
4.1 Smart attack	17
4.2 Pollard's Rho attack	19
4.3 Invalid Curve Attack	22
4.4 MOV Attack	26
4.5 Frey Ruck Attack	29
V. DEPLOYMENT	32
VI. REFERENCE	33

Table of Image, Graph

Image 1.2.a	Illustration of elliptic curves
Image 1.2.b	Illustration of Point Adding
Table 1.4.1	The double-and-add algorithm

PART I. OVERVIEW

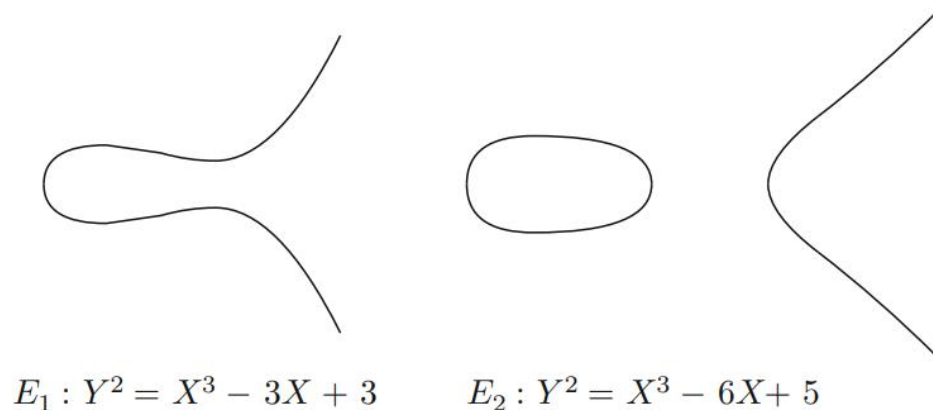
1.1 Defining ECC

- Elliptic curve cryptography (ECC) is a public key cryptographic algorithm used to perform critical security functions, including encryption, authentication, and digital signatures. ECC is based on the elliptic curve theory, which generates keys through the properties of the elliptic curve equation, compared to the traditional method of factoring very large prime numbers. ECC is used in lots of context, such as:
 - *IoT (Internet of Things)*: In IoT, ECC provides strong security with smaller key sizes, which is crucial for resource-constrained devices. It enables secure communication and authentication between IoT devices, ensuring data integrity and confidentiality.
 - *Digital Signature*: ECC is used to generate digital signatures that authenticate the origin and integrity of a message or document. ECDSA (Elliptic Curve Digital Signature Algorithm) is a widely used ECC-based signature algorithm, offering high security with efficient performance.
 - *Wireless Security*: ECC enhances wireless security by enabling secure key exchange protocols, such as ECC-based Diffie-Hellman. This ensures that data transmitted over wireless networks remains confidential and protected against eavesdropping and man-in-the-middle attacks.
 - *Cloud Computing*: In cloud computing, ECC secures data storage and transmission. It provides strong encryption for data at rest and in transit, ensuring that sensitive information remains protected against unauthorized access.
 - *Smart Cards*: Smart cards use ECC for secure authentication and encryption. ECC's efficiency in key generation and encryption/decryption operations makes it suitable for smart cards, which have limited computational resources.
 - *Blockchain*: ECC is integral to blockchain technology, particularly in cryptocurrencies like Bitcoin and Ethereum. It secures transactions through digital signatures (e.g., ECDSA) and ensures that only the rightful owner can authorize the transfer of assets.
 - *Secure Web Browsing*: ECC is used in secure web browsing through protocols like TLS (Transport Layer Security). ECC-based certificates ensure that data exchanged between web browsers and servers is encrypted and secure from interception.

- To summarize, ECC is a powerful tool that can protect data, authenticate connections, and verify integrity in various applications. As more and more of our lives move online, cryptography is essential to keep our data safe and secure. And Elliptic curve cryptography is just one type of cryptographic algorithm that can be used for this purpose.

1.2 Elliptic Curve

- The generalized Weierstrass equation is the expression of the generic form of an elliptic curve. $E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, elliptic curve (E) defined over a finite field K.
- However in this paper, I will constantly work with the reduced form of the elliptic curve, which is $E : y^2 = x^3 + Ax + B$, where A and B are constants. This simplified equation is called the Weierstrass equation of an elliptic curve. The variables x, y and the constants A, B lie in the finite field of form F_p and F_q , where p is prime number and $q = p^k$, where $k \geq 1$.
- All the points on an elliptic curve lies in, $E(\mathbb{L}) = \{\infty\} \cup \{(x, y) \in \mathbb{L} \times \mathbb{L} \mid y^2 = x^3 + Ax + B\}$
- Generally there is a condition which an elliptic curve has to fulfill. In elliptic curves we do not allow singular points or multiple roots. So, one of the way to check a given curve is an elliptic curve or not, is to check whether the curve satisfies the following condition or not, $\Delta = 4A^3 + 27B^2 \neq 0$. There are two example Elliptic Curves below:



[Image 1.2.a](#)

- And 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: $y^2 + xy = x^3 + ax^2 + b \dots$

⊗ Addition Law:

- 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.

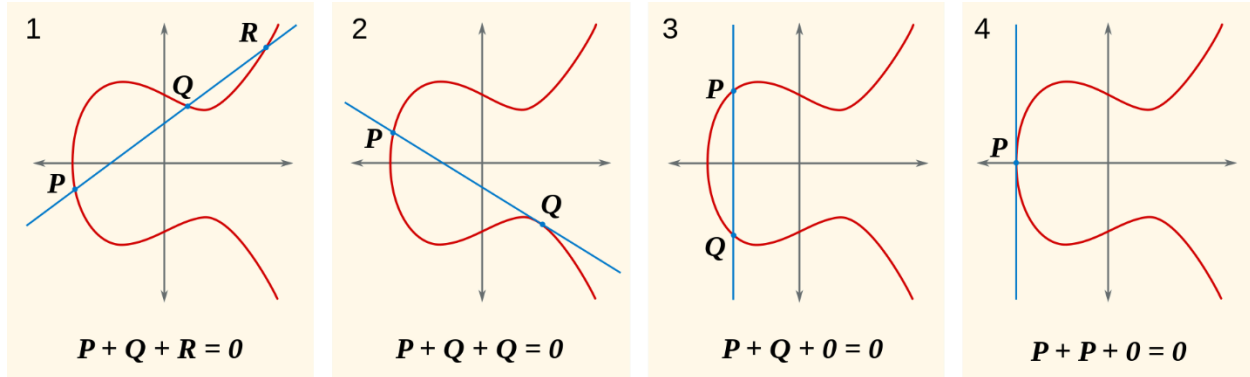


Image 1.2.b

- Or detailly: 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.*

+ Let P_1, P_2 be two points in which $P_1, P_2 \in E$, where $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ of the elliptic curve E . A new, third point $P_3 = (x_3, y_3)$ is produced using points P_1 & P_2 .

+ Let l be the line passing through P_1, P_2 where $P_1, P_2 \in E$, then $P_3 = P_1 + P_2$. And P_3 is analytically classified into the following cases:

- Case 1: $P_1 \neq P_2$ and $x_1 \neq x_2$

$$x_3 = m^2 - x_2 - x_1, y_3 = m(x_1 - x_3) - y_1, \text{ where } m = (y_2 - y_1)/(x_2 - x_1)$$

- Case 2: $P_1 \neq P_2$ and $x_1 = x_2$

$$P_1 + P_2 = \infty$$

- Case 3: $P_1 = P_2$ and $y_1 \neq 0$

$$x_3 = m^2 - 2x_2, y_3 = m(x_1 - x_3) - y_2, \text{ where } m = (3x_2 + A)/(2y_2)$$

○ Case 4: $P_1 = P_2$ and $y_1 = 0$

$$P_1 + P_2 = \infty$$

○ Case 5: $P_1 = \infty$

$$\infty + P_2 = P_2$$

⇒ Point Addition on Elliptic Curve Suffice the below Properties Under the Condition $P, P_1, P_2 \in E$:

- $O + P = P + O = P$ where O is an identity element.
- $P_1 + P_2 = P_2 + P_1$ (commutative)
- $\infty + P_1 = P_1$ (existence of identity)
- if $P_1 + P_2 = \infty$, then $P_1 \equiv -P_2$ (existence of inverse)
- $(P_1 + P_2) + P = P_1 + (P_2 + P)$ (associative)

Example:

Consider an elliptic curve E over a field K .

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

Let $P=(x_1, y_1)$ be a point on $E(K)$.

Suppose we have a second point $Q=(x_2, y_2)$ different from P . We wish to find $P+Q$ whose coordinates we shall denote by (x_3, y_3) . If $P=-Q$, then $P+Q=O$. Otherwise the gradient of the line determined by P and Q is

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

and the equation of the line between P and Q is $Y=\lambda X-\lambda x_1+y_1$. Substituting this into the curve gives the equation:

$$(\lambda X - \lambda x_1 + y_1)^2 + (a_1X + a_3)(\lambda X - \lambda x_1 + y_1) = X^3 + a_2X^2 + a_4X + a_6$$

1.3 Elliptic Curve over Finite Fields

- In mathematics, a finite field or Galois field is a field that contains a finite number of elements. 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 .
- Definition:
 - + Given a prime power $q = p^r > 3$, consider the finite field F_q and a pair (A, B) where $A, B \in F_q$ such that $4A^3 + 27B^2 \neq 0$. We consider the set of solutions (x, y) to the cubic equation $y^2 = x^3 + Ax + B$ over the finite field F_q .
 - + This set of ordered pairs, when augmented by a “point at infinity” O , can be given a natural abelian group structure, where O serves as the identity element. Such groups are called elliptic curves over F_q , and in fact elliptic curve groups can be defined similarly over any field K .
- For 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 .

\Rightarrow *Theorem : 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.

1.4 The Elliptic Curve Discrete Logarithm Problem (ECDLP)

- 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 .

1.4.1 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: $Q_i = 2 * Q_{i-1}$

<p>Input: a point P, an n-bit integer $k = \sum_{i=0}^{n-1} k_i 2^i$</p> <p>Output: $k \times P$</p> <p>$Q[0] = \mathcal{O}$</p> <p>for i from $n - 1$ down to 0</p> <p> $Q[0] = 2Q[0]$</p> <p> $Q[1] = Q[0] + P$</p> <p> $Q[0] = Q[k_i]$</p> <p>return $Q[0]$</p>
--

Table 1.4.1. The double-and-add algorithm for elliptic curves

1.4.2 How hard is the ECDLP?

- The DLP is considered to be a hard problem and it is the basis for the security of many cryptographic systems. For example, in the Diffie-Hellman key exchange, the two parties agree on a prime number p and a generator g , and then each party generates a secret number (the exponent) x and computes $g^x \text{ mod } p$. They then exchange the results ($g^x \text{ mod } p$) and can use them to compute a shared secret key. Without knowledge of the secret exponent x , it is computationally infeasible to determine the shared secret key.
- In Elliptic Curve Cryptography (ECC), the DLP is also used to generate private and public key pairs. The private key is an integer (scalar) and the public key is the point multiplication of a generator point with the scalar.

The private key is kept secret, while the public key is shared. Without knowledge of the private key, it is computationally infeasible to determine the private key from the public key.

- Therefore, the discrete logarithm problem is a fundamental problem in cryptography, because if it could be solved efficiently, it would break the security of many cryptographic systems that rely on it.

PART II. KEY GENERATION IN ECC

Here is step-by-step of key generation in ECC:

- **Selecting Parameters:**

- Choose an appropriate elliptic curve E defined over a finite field F_p . This involves selecting parameters such as the prime p , the coefficients A and B of the curve equation $y^2 = x^3 + Ax + B$, and the base point G on the curve.
- Determine the order n of the base point G , which is the number of points on the curve generated by repeatedly adding G to itself.
- 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”. We suggest using standard curves like curve25519 (<https://cr.yp.to/ecdh.html>) which was tested and ensured to be safe.

- **Generating the Private Key:**

- Select a random integer d from the interval $[1, n-1]$. This integer will serve as the private key. The size of the private key depends on the desired level of security. Ensure that d is kept secret and securely stored, as it will be used to derive the corresponding public key.

- **Computing the Public Key:**

- Use scalar multiplication to compute the public key Q , which is a point on the elliptic curve E . The public key Q is computed as $Q = d \cdot G$, where G is the base point and \cdot denotes scalar multiplication. This operation involves adding the base point G to itself d times using elliptic curve point addition.

And here is an example key generation in ECC using Sagemath library:

```

genKey.py
1 from sage.all import *
2 from secrets import *
3
4 # NIST P-256
5 a = -3
6 b = 41058363725152142129326129788047268409114441015993725554835256314039467401291
7 p = 2**256 - 2**224 + 2**192 + 2**96 - 1
8 K = GF(p)
9 n = 11579208921035624876269744694940757352999695522413576034242259061068512044369
10 Gx = 48439561293906451759052585252797914202762949526041747995844080717082404635286
11 Gy = 36134250956749795798585127919587881056611106672985015071877198253568414405109
12 E = EllipticCurve(K, (a, b))
13 G = E(Gx, Gy)
14
15 # Generate key
16 privateKey = randbelow(n)
17 publicKey = G * privateKey
18
19 # Result
20 print('{p = }')
21 print('{a = }')
22 print('{b = }')
23 print('{Base point G = ", G.xy()')
24 print('{privateKey = }')
25 print('{publicKey = ", publicKey.xy()')

```

```

thnall08@thnall08:~/Documents/gen_key_ecc$ python3 genKey.py
p = 115792089210356248762697446949407573530886143415290314195533631308867097853951
a = -3
b = 41058363725152142129326129788047268409114441015993725554835256314039467401291
Base point G = (48439561293906451759052585252797914202762949526041747995844080717082404635286, 36134250956749795798585127919587881056611106672985015071877198253568414405109)
privateKey = 65448547248653465236196485877318513664783935243996817794484499831560534536596
publicKey = (113067565239868382697142219511674028487178113335722545130330772500248099670797, 1420838463248287628965345107899694368842613955277545403863909703033691119867)
thnall08@thnall08:~/Documents/gen_key_ecc$

```

PART III. SOME ATTACK MODELS IN ECC

3.1 Smart Attack

- For an elliptic curve E over a field Fp , a linear time approach of computing the elliptic curve discrete logarithm problem (ECDLP) is presented in Smart Attack. The primary condition for a curve vulnerable to smart's attack is its trace of Frobenius is equal to one, which indirectly implies the number of points on the elliptic curve E is equal to p (prime which the elliptic curve is defined).
- 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.

Algorithm: Smart Attack

- 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 ?
 - First step, we try to lift these points to $E(QP)$ (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 Qp . We know that $Q = kP$ in $E(Fp)$ so thus in the kernel of that homomorphism.

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

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

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

- with $pQ' \in E_1(Qp)$ and $pP' \in E(QP)$. 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 Fp solving ECDLP.

3.2 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 $G = E(FP)$, 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: Pollard's Rho attack

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 $R_i = aiP + biQ$, and therefore

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

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,

$$R_m = R_{2m} \text{ with some argument } m$$

Once we found a match, we have

$$R_m = a_m P + b_m Q$$

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

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 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 $E(Fp)$. If the $\gcd(bm - b_{2m}, n) > 1$, the inverse of $(bm - b_{2m})$ 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, Fp and the point P are chosen such that $\#E(Fp) = n$ is a prime. That means that there is a high probability of success in solving the ECDLP using the Pollard-Rho Algorithm. While choosing n to be prime increases the success of this attack, recall that the Pohlig-Hellman attack works by factoring n . Curves that are susceptible to the Pohlig-Hellman attack are deemed insecure and are unacceptable for use in commercial implementations. Therefore, for a cryptosystem to be protected against the Pohlig-Hellman attack, n should be prime.

3.3 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 $E'(F_p)$ 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 :

$$y^2 = x^3 + ax + b \mod p$$

$$y^2 = x^3 + ax + b_1 \mod p$$

$$y^2 = x^3 + ax + b_2 \mod p$$

$$y^2 = x^3 + ax + b_3 \mod p$$

- Now we have multiple curves from which we can choose our points and share as public keys, and this will not affect the results of Elliptic Curve Arithmetic. We can selectively choose points having small order of the subgroup, generated by scalar multiplication. The remaining steps we can use Pohlig-Helman to solve $DLP(xP=Q)$ then use CRT to find x .

3.4 MOV Attack

- MOV Attack is used with the target of reducing the ECDLP to an easier problem (Ex: an easier group).
- With the *Idea* that: Use the Weil/Tate pairing to convert a discrete log problem in $E(F_p)$ to one in $F_{p^k}^*$.
- The MOV attack uses a bilinear pairing, which (roughly speaking) is a function e that maps two points in an elliptic curve $E(F_q)$ to a element in the finite field $F_{p^k}^*$, where k is the embedding degree associated with the curve. The

bilinearity means that $e(rP, sQ) = e(P, Q)^{rs}$ for points P, Q . Therefore, if you want to compute the discrete logarithm of $rPrP$, you can instead compute $u = e(P, Q)$ and $v = e(rP, Q)$ for any Q . Due to bilinearity, we have that $v = e(P, Q)r = ur$. Now you can solve the discrete logarithm in $F_{p^k}^x$ (given ur and u , find r) in order to solve the discrete logarithm in the elliptic curve.

- Usually, the embedding degree k is very large (the same size as q), therefore transferring the discrete logarithm to $F_{p^k}^x$ won't help you. But for some curves the embedding degree is small enough (specially supersingular curves, where $k \leq 6$), and this enables the MOV attack. For example, a curve with a 256-bit q usually offers 128 bits of security (i.e. can be attacked using 2^{128} steps); but if it has an embedding degree 2, then we can map the discrete logarithm to the field F_{q^2} which offers only 60 bits of security.
- Since discrete log problems in finite fields can be attacked by index calculus (or other) methods, they can be solved faster than elliptic curve discrete log problems, as long as the field $F_{p^k}^x$ is not much larger than F_p .
- Recall that for an elliptic curve E defined over F_p
 - we let $E[m]$ denote the set of points of order dividing m with coordinates in the algebraic closure.
 - If $\gcd(p, m) = 1$ and $S, T \in E[m]$, then the Weil pairing $e_m(S, T)$ is an m th root of unity and can be computed fairly quickly.
 - The pairing is bilinear, and if $\{S, T\}$ is a basis for $E[m]$, then $e_m(S, T)$ is a primitive m th root of unity.
 - For any $S \Rightarrow e_m(S, S) = 1$.
- There exists n such that $Q = nP \Leftrightarrow mQ = O$ and the Weil pairing $e_m(P, Q) = 1$.
 - Let E be an elliptic curve over F_p .
 - Let $P, Q \in E(F_p)$.
 - Let m be the order of P .
 - Assume that $\gcd(m, p) = 1$.

We want to find n such that $Q = nP$

Lemma: $\exists n$ such that $Q = nP \Leftrightarrow mQ = O$ and $e_m(P, Q) = 1$

Algorithm: MOV attack

Choose embedding degree k such that $E[m] \subset E(F_{p^k}^x)$

1. Compute the number of points $N = |E(\mathbb{F}_{p^k}^x)|$
2. Choose a random point $T \in E(\mathbb{F}_{p^k}^x)$, $T \notin E(\mathbb{F}_{p^k})$
3. Compute the order t of T
4. let $d = \gcd(t, m)$, let $T' = (\frac{t}{d})T \Rightarrow T'$ has order d which divides $m \Rightarrow T' \in E[m]$
 - We can use N/m instead of d but takes more computation
5. Compute
 - $\alpha = e_m(P, T')$
 - $\beta = e_m(Q, T')$
6. Solve $\alpha = \beta n$ in $\mathbb{F}_{p^k}^x \Rightarrow n \bmod d$
7. If there are more n 's, crt the results

3.5 Frey Ruck Attack

- For the purpose of the attack we will use what is referred to as a modified Tate-Lichtenbaum pairing. We note that the group $K^*/(K^*)^n$ is isomorphic to the group of roots of unity μ_n and thus an instance of the ECDLP on $E(K)$ can be mapped to an instance of the DLP in μ_n . Now we can define τ_n to be the following bilinear map:

$$\tau_n(\cdot, \cdot) : E[n] \times E(K)/nE(K) \rightarrow \mu_n$$

$$\tau_n(P, Q) = (P, Q)^{(q-1)/n}.$$

- Although the setting is exactly the same, the second setup is more desirable since it will yield a definite answer instead of a coset in K^* modulo n -th powers. Again, since we are mapping into the group of n -th roots of unity, we are mapping into a suitable extension field K such that $\mu_n \subseteq K$. Now we describe the Frey-Ruck algorithm as given in:

Algorithm : The Frey-Ruck attack

Input: An element $P \in E(\mathbf{k})$ of order n and $Q \in E(K)$.

Output: An integer λ such that $Q = \lambda P$.

- 1) Determine the smallest integer l such that $n \mid q^l - 1$ and set $K = \mathbb{F}_{q^l}$.
- 2) Pick $S, T \in E(K)$ randomly.
- 3) Compute the element $f \in K(E)^*$ such that $\text{div}(f) = n((P) - (O))$ and

compute $\alpha = f(S)/f(T)$.

4) Compute the element $\gamma = \alpha^{(q^{l-1})/n}$. If $\gamma = 1$, then go to 2).

5) Compute the element $g \in K(E)^*$ such that $\text{div}(g) = n((Q) - (O))$ and compute $\beta = g(S)/g(T)$, and $\delta = \beta^{(q^{l-1})/n}$.

6) Solve the DLP $\delta = \gamma^x$ in K^* , i.e. the logarithm of δ to the base γ in K^* .

PART IV. IMPLEMENTATION AND TESTING

In this part, we use python3.x, sagemath, ecdsa package, pycryptodome package, socat package, pwn tools package and some other of cryptographic libraries to present our model attack with building Docker:

4.1 Smart Attack

Firstly, we have build curve with the parameters 512-bits below:

```
p = 0xa15c4fb663a578d8b2496d3151a946119ee42695e18e13e90600192b1d0abdbb6f787f90c8d102ff88e284dd4526f5f6b6c980bf88f1d0490714b67e8a2a2b77
a = 0x5e009506fcc7eff573bc960d88638fe25e76a9b6c7caeea072a27dcd1fa46abb15b7b6210cf90caba982893ee2779669bac06e267013486b22ff3e24abae2d42
b = 0x2ce7d1ca4493b0977f088f6d30d9241f8048fdea112cc385b793bce953998caae680864a7d3aa437ea3ffd1441ca3fb352b0b710bb3f053e980e503be9a7fece
E = EllipticCurve(GF(p), [a, b])
def get_plain_text(file_path):
    try:
        with open(file_path, "rb") as f:
            plaintext = f.read()
            return plaintext
    except FileNotFoundError:
        print(f"File '{file_path}' not found.")
        return None
    except Exception as e:
        print(f"Error reading file '{file_path}': {str(e)}")
        return None
def encrypt(key, plain):
    try:
        aes_key = sha3_512(str(key).encode()).digest()[:16]
        iv = random.randbytes(16)
        cipher = AES.new(aes_key, AES.MODE_CBC, iv)
        encrypted_data = cipher.encrypt(pad(plain, 16))
        return iv + encrypted_data
    except Exception as e:
        print(f"Encryption error: {str(e)}")
        return None
def generate_private_key(P):
    try:
        return random.randint(1, P.order() - 1)
    except Exception as e:
        print(f"Error generating private key: {str(e)}")
        return None
def generate_public_key(P, n):
    try:
        return P * n
    except Exception as e:
        print(f"Error generating public key: {str(e)}")
        return None
assert is_prime(E.order())
P = E.gen(0)
file_path = "test.txt"
plaintext = get_plain_text(file_path)
n = generate_private_key(P)
Q = generate_public_key(P, n)
print(f'{a = }')
print(f'{b = }')
print(f'{p = }')
print('P =', P.xy())
print('Q =', Q.xy())
encrypted = encrypt(n, plaintext)
with open("/output/cipher.enc", "wb") as cipher_file:
    if encrypted is not None:
        cipher_file.write(encrypted)
    else:
        cipher_file.write(b'None')
```

At this phase, we use smart_attack function to compute a discrete logarithm. By these steps, we takes three parameters: points P and Q on an elliptic curve E, and a prime number p, after that we defines a new elliptic curve E_{Qp} over the field Q_p (the field of p-adic numbers) using the same invariants as the curve E, but modified slightly by adding a random multiple of p. And so on, we have :

- Lifting Points to Q_p
- Multiplication by p
- Calculation of ϕ -values
- Calculation of k and we have the result is $k = \frac{\phi Q}{\phi P} = \frac{-\frac{xQ}{yQ}}{-\frac{xP}{yP}}$

```
def smart_attack(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)

r = remote('localhost', 8888)
a = int(r.recvline().split('=')[1].strip())
b = int(r.recvline().split('=')[1].strip())
p = int(r.recvline().split('=')[1].strip())
P = eval(r.recvline().split('=')[1].strip())
Q = eval(r.recvline().split('=')[1].strip())
E = EllipticCurve(GF(p), [a, b])
P = E(*P)
Q = E(*Q)

with open("output/cipher.enc", "rb") as cipher_file:
    enc = cipher_file.read()

def decrypt(key, enc):
    try:
        key = sha3_512(str(key).encode()).digest()[:16]
        iv = enc[:16]
        enc = enc[16:]
        cipher = AES.new(key, AES.MODE_CBC, iv)
        return unpad(cipher.decrypt(enc), 16)
    except Exception as e:
        print(f"Decryption error: {str(e)}")
        return None

secret = smart_attack(P, Q, p)
assert P * secret == Q
print(f'Secret: {secret}')
decryptText = decrypt(secret, enc)
with open("recovered.txt", "wb") as recover_file:
    if decryptText is not None:
        recover_file.write(decryptText)
        print("Write to recovered.txt successfully!!!")
    else:
        print("Write to recovered.txt failed!!!")
```

With the provided algorithm and some Sagemath function, we can easily decrypt and recover the pdf file so we execute and find out the secret values. This is our output:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
○ thna1108@thna1108:~/Documents/attack_smart$ nc localhost 8888
a = 4923298572065486992549817192831990694521484100405815221208729152966589637309679506395496485479241352064070239922464839767350734501408775691888115823029570
b = 235189422232427722574035031607638866089626162814242940057864245019071402791248171844982397521930502454173681491594668678670733042074255983768208195714766
p = 845113990555190283116035499024344899473923344327179631310525520435196861496551764162970081762137226285330762159796842785356834064520159547540428116601719
P = (3034712809375537908102988750113382444008758539448972750581525810900634243392172703684905257490982543775233630011707375189041302436945106395617312498769005, 498664509858261641569
007408223781762442433339074969364527548107042876175480894132576399611027847402879885574130125050842710052291870268101817275410204850)
Q = (7510583788277517265915963166882107781140335049269319621110549637774321757218202802818020148380732542631849608862007729899269357714952177655676754528698639, 188861974477811001203
6328715210365862486236495628989010055371133422037786146516178982199903273578843678223373301809937158155305628008165734808216204704652)

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
○ thna1108@thna1108:~/Documents/attack_smart$ python solve.py
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
● thna1108@thna1108:~/Documents/attack_smart$ python3 solve.py
[+] Opening connection to localhost on port 8888: Done
Secret: 7849454767088139314739722948120023949965288487367170014470515033785920719617528964909170482197051559132853469807764277243543645768204556200641196039626761
Write to recovered.txt successfully!!!
[*] Closed connection to localhost port 8888
○ thna1108@thna1108:~/Documents/attack_smart$

```

4.2 Pollard' Rho Attack

The script below is an implementation of elliptic curve cryptography (ECC) to encrypt a file (input.pdf) using a prime number chosen by the user. The script includes functions to check the validity of the prime, generate elliptic curve parameters, and encrypt the file.

Specialy, the genPara function generates random coefficients a and b for the elliptic curve $E: y^2 = x^3 + ax + b$ over the finite field $GF(p)$:

- It ensures that the discriminant $4a^3 + 27b^2$ is not zero (which means the curve is non-singular).
- It checks that the order of the elliptic curve is prime.

```

def check(prime):
    if not isPrime(prime):
        print("Not a prime!!!")
        return False
    if prime <= (2**35):
        print("Prime too small!!!")
        return False
    return True

def genPara(p):
    while True:
        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())):
            return a,b

def encrypt(key, mess):
    key = sha3_512(str(key).encode()).digest()[16:]
    iv = random.randbytes(16)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ct = cipher.encrypt(pad(mess, AES.block_size))
    return iv + ct

while True:
    p = int(input("Enter prime: "))
    if check(p):
        break

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

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

with open("input.pdf", 'rb') as file:
    pt = file.read()

ciphertext = encrypt(secret, pt)
with open("./output/cipher.enc", "wb") as file:
    file.write(ciphertext)
    print("Write ciphertext to cipher.enc successfully!")

```

```

def pollardrho(P, Q, E):
    n = P.order()

    for j in range(100):
        a_i = random.randint(2, P.order() - 2)
        b_i = random.randint(2, P.order() - 2)
        a_2i = random.randint(2, P.order() - 2)
        b_2i = random.randint(2, P.order() - 2)

        X_i = a_i * P + b_i * Q
        X_2i = a_2i * P + b_2i * Q

        i = 1
        while i <= n:
            a_i = func_g(a_i, P, X_i, E)
            b_i = func_h(b_i, P, X_i, E)
            X_i = func_f(X_i, P, Q, E)

            a_2i = func_g(func_g(a_2i, P, X_2i, E), P, func_f(X_2i, P, Q, E), E)
            b_2i = func_h(func_h(b_2i, P, X_2i, E), P, func_f(X_2i, P, Q, E), E)
            X_2i = func_f(func_f(X_2i, P, Q, E), P, Q, E)

            if X_i == X_2i:
                if b_i == b_2i:
                    break
                if GCD(b_2i - b_i, n) != 1:
                    break
                print(f"Collision found at iteration {j}")
                return ((a_i - a_2i) * inverse_mod(b_2i - b_i, n)) % n
            else:
                i += 1
                continue
        print(f"No collision found in iteration {j}")
    return None

```

This is our output:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
○ thna1108@thna1108:~/Documents/attack_pollard_rho$ nc localhost 6003
Enter prime:

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● thna1108@thna1108:~/Documents/attack_pollard_rho$ python3 rho.py
[+] Opening connection to localhost on port 6003: Done
a = 33696931238
b = 36679072517
p = 38963647271
P = (9440622162, 37575300856)
Q = (30784951408, 23641553016)
Collision found at iteration 0
Found secret key: 20343344313
Successfully decrypted and saved to recovered.pdf
[*] Closed connection to localhost port 6003
○ thna1108@thna1108:~/Documents/attack_pollard_rho$
```

4.3 Invalid Curve Attack

The code below is used to illustrate the curve to encrypt data on the server. We use NIST P-256, a standard Curve with the chosen parameters.

```

# NIST P-256
a = -3
b = 41058363725152142129326129780047268409114441015993725554835256314039467401291
p = 2**256 - 2**224 + 2**192 + 2**96 - 1
E = Curve(p, a, b)
n = 115792089210356248762697446949407573529996955224135760342422259061068512044369
Gx = 48439561293906451759052585252797914202762949526041747995844080717082404635286
Gy = 36134250956749795798585127919587881956611106672985015071877198253568414405109
G = Point(E, Gx, Gy)
d = randbelow(n)
P = G * d
def point_to_bytes(P):
    return P.x.to_bytes(32, "big") + P.y.to_bytes(32, "big")
def encrypt(P, m):
    key = point_to_bytes(P)
    return bytes([x ^ y for x, y in zip(m.ljust(64, b"\0"), key)])
quotes = [
    "Lap trinh mAng Can ban truong Dai HOC CoNg NGE thONG tin +++_++",
    "NT 219 Mat ma hoc, Mat ma hoc",
    "Nhom do an!",
    "ky niem 70 nam chien thang Dien Bien Phu",
    "HA!HA!HA!HA!HA!",
    "HA!HA!HA!HA!",
    "it's me group sixxx!",
    "imPleMentAtION_%1",
]
print("Show public key: %s" % P)
while True:
    try:
        print("What do you want?")
        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(encrypt(S, choice(quotes).encode()).hex())
        elif option == 2:
            r = randbelow(n)
            C1 = r * G
            C2 = encrypt(r * P, flag)
            print(point_to_bytes(C1).hex())
            print(C2.hex())
        elif option == 3:
            print("otsupeko!")
            break
        print()
    except Exception as ex:
        print("kusa peko")
        print(ex)
        break

```

These code below are showing the setting up for 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 * l - 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)
```



```

class Point:
    def __add__(self, other):
        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 * l - 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)

```

From the code 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 setting of curve does not 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. We will find 4 curves where the order of the curve is a prime factor greater than or equal to 64 bits and smooth. Because the curve of the server is secp256r1, so that if we want to send 4 curves to the server, they must be at least 64 bits in order to obtain a 256-bit result after performing the final CRT computation, matching the original order size. And this is our output:

```
Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
thna1108@thna1108:~/Documents/attack_invalid_curve$ nc localhost 6001
Show public key: (4163638555053197585820142690773791825935430937838807073915593853235770064972, 101637957977318705812607462777926313537772472689460348576015816782126409347822)
What do you want?
1. Start a Diffie-Hellman key exchange
2. Get an encrypted flag
3. Exit
>

solving size 257467541
d: 92592070 mod 257467541

Trying y^2 = x^3 - 3x + 73633465256250339103776962879052917369957363386373067311696537523933666695863
solving size 2
d: 1 mod 2
solving size 13
d: 8 mod 13
solving size 1103
d: 324 mod 1103
solving size 601333
d: 564446 mod 601333
solving size 1735271
d: 923976 mod 1735271
solving size 3850373
d: 1756962 mod 3850373

Trying y^2 = x^3 - 3x + 80463878268200288270757505298185669035709877255855862263440097314911653584
solving size 4
d: 3 mod 4
solving size 3
d: 0 mod 3
solving size 5
d: 1 mod 5
solving size 529
d: 193 mod 529
solving size 283
d: 67 mod 283
solving size 336143
d: 65173 mod 336143
solving size 15597487
d: 6041556 mod 15597487

Trying y^2 = x^3 - 3x + 6759871680847809782271110194234668451992108185471980919599226530970565598430
solving size 8
d: 3 mod 8
solving size 3
d: 0 mod 3
solving size 19
d: 14 mod 19
solving size 3061
d: 837 mod 3061
solving size 997739
d: 336630 mod 997739
solving size 19310171
d: 2820545 mod 19310171
solving size 54620359
d: 53231875 mod 54620359
solving size 125716051
d: 1176265 mod 125716051
Secret key = 94703791345897506305581512463257202989835636016389366901031983085321513162051
Secret plaintext: b'MMH{implementation_invalid_curve_attack_successful_nIcE!}'
[*] Closed connection to localhost port 6001
thna1108@thna1108:~/Documents/attack_invalid_curve/solve_invalid$
```

4.4 MOV Attack

This code snippet is about setting up a custom-curve for a server:

```

with open("input.pdf", "rb") as re:
    inputt = re.read()
a = -35
b = 98
p = 434252269029337012720086440207
Gx = 16378704336066569231287640165
Gy = 377857010369614774097663166640
ec_order = 434252269029337012720086440208
E = ecc.CurveFp(p, a, b)
G = ecc.Point(E, Gx, Gy, ec_order)
def exchange():
    alice_secret, bob_secret = random.randint(1,pow(2,64)), random.randint(1,pow(2,64))
    aG = G * alice_secret
    bG = G * bob_secret
    print(f'a = {alice_secret}')
    print(f'b = {bob_secret}')
    print(f'p = {p}')
    print('Gx =', Gx)
    print('Gy =', Gy)
    print('aGx =', aG.x())
    print('aGy =', aG.y())
    print('bGx =', bG.x())
    print('bGy =', bG.y())
    shared_secret = aG * bob_secret
    return shared_secret
def encrypt(shared_secret: int):
    hash = sha256(str(shared_secret).encode()).digest()
    iv, key = hash[:16], hash[16:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(pad(inputt, 16))
    return encrypted
while True:
    try:
        print("What do you want?")
        print("1. Get an encrypted flag")
        print("2. Exit")
        option = int(input("> "))
        if option == 1:
            s = exchange()
            encSend = encrypt(int(s.x()))
            with open("/output/cipher.enc", "wb") as cipher:
                cipher.write(encSend)
        elif option == 2:
            print("Exit!")
            break
        print()
    except Exception as ex:
        print("Error")
        print(ex)
        break

```

This MOV attack takes advantage of the embedding degree k to map the problem from an elliptic curve to a finite field, where the discrete logarithm problem is more manageable. And follow these steps: **Finding the Embedding Degree k :**

- The embedding degree k is the smallest positive integer such that $p^k = 1 \pmod{N}$, where N is the order of the elliptic curve.
- This loop iteratively increments k until the condition is satisfied.

Finding T_1 :

- Calculates T_1 , which is a multiple of T such that T_1 has an order that is the least common multiple of M and N .

$$T_1 = (M // \gcd(M, N)) * T$$

Weil Pairing:

- Computes the Weil pairing of G and Q with T1 on the extended curve E2.

$$_G = E2(G).weil_pairing(T1, N)$$

$$_Q = E2(Q).weil_pairing(T1, N)$$

Solve the logarithm discrete:

- Solves the discrete logarithm problem in the finite field $GF(p^k)$ to find nQ, such that $Q=n \cdot G$

$$nQ = _Q.log(_G)$$

```
E = EllipticCurve(GF(p), [a, b])
G = E(Gx, Gy)
A = E(aGx, aGy)
B = E(bGx, bGy)

info(f"Base Point: {(Gx,Gy)}")
info(f"Alice : {(aGx, aGy)}")
info(f"Bob   : {(bGx, bGy)}")

# Actually solving
def movAttack(G, Q, p, a, b):
    # finding the embedding degree
    k = 1
    while (p**k - 1) % E.order():
        k += 1

    E2 = EllipticCurve(GF(p**k), [a,b])
    T = E2.random_point()
    M = T.order()
    N = G.order()
    T1 = (M//gcd(M, N)) * T
    _G = E2(G).weil_pairing(T1, N)
    _Q = E2(Q).weil_pairing(T1, N)
    nQ = _Q.log(_G)
    return nQ

# see `source.py`
def decrypt(secret: int, ciphertext: bytes):
    hash = sha256(str(secret).encode()).digest()
    iv, key = hash[:16], hash[16:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = cipher.decrypt(ciphertext)
    return decrypted

with open("output/cipher.enc", "rb") as cipher_file:
    enc = cipher_file.read()
alice_secret = movAttack(G, A, p, a, b)
shared_secret = B * alice_secret
info(f"shared: {shared_secret[0]} {shared_secret[1]}")
flag = unpad(decrypt(shared_secret[0], enc), 16)
with open("recovered.txt", "wb") as recover_file:
    if flag is not None:
        recover_file.write(flag)
        print("Write to recovered.txt successfully!!!")
    else:
        print("Write to recovered.txt failed!!!")
```

And this is our output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● thna1108@thna1108:~/Documents/attack_mov_pairing$ python3 attack.py
[+] Opening connection to localhost on port 6002: Done
[*] Base Point: (16378704336066569231287640165, 377857010369614774097663166640)
[*] Alice : (147225595015416452140687832210, 88328581507564332604648787754)
[*] Bob   : (90395855244375249077436762013, 286174683398001198001159554100)
[*] shared: 197113916782848229727798069970 241573921941641588654083393758
Write to recovered.txt successfully!!!
[*] Closed connection to localhost port 6002
○ thna1108@thna1108:~/Documents/attack_mov_pairing$
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
○ thna1108@thna1108:~/Documents/attack_mov_pairing$ nc localhost 6002
What do you want?
1. Get an encrypted flag
2. Exit
>
```

4.5 Frey Ruck Attack

We start with these parameters

```

with open("input.pdf", "rb") as re:
    inputt = re.read()
a = -35
b = 98
p = 434252269029337012720086440207
Gx = 16378704336066569231287640165
Gy = 377857010369614774097663166640
ec_order = 434252269029337012720086440208
E = ecc.CurveFp(p, a, b)
G = ecc.Point(E, Gx, Gy, ec_order)

def exchange():
    alice_secret, bob_secret = random.randint(1,pow(2,64)), random.randint(1,pow(2,64))
    aG = G * alice_secret
    bG = G * bob_secret
    print(f'{a = }')
    print(f'{b = }')
    print(f'{p = }')
    print('Gx =', Gx)
    print('Gy =', Gy)
    print('aGx =', aG.x())
    print('aGy =', aG.y())
    print('bGx =', bG.x())
    print('bGy =', bG.y())
    shared_secret = aG * bob_secret
    return shared_secret

def encrypt(shared_secret: int):
    hash = sha256(str(shared_secret).encode()).digest()
    iv, key = hash[:16], hash[16:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(pad(inputt, 16))
    return encrypted

while True:
    try:
        print("What do you want?")
        print("1. Get an encrypted flag")
        print("2. Exit")
        option = int(input("> "))
        if option == 1:
            s = exchange()
            encSend = encrypt(int(s.x()))
            with open("/output/cipher.enc", "wb") as cipher:
                cipher.write(encSend)
        elif option == 2:
            print("Exit!")
            break
        print()
    except Exception as ex:
        print("Error")
        print(ex)
        break

```

And so on, this method use a technique that leverages the embedding degree k to map the problem to a finite field where Elliptic Curve Discrete Logarithm Problem (ECDLP) can be solved more easily.

The steps include calculating the embedding degree k , extending the elliptic curve to $\text{GF}(q^k)$, computing Tate pairings for random points, and solving the discrete logarithm in the finite field. This function calculates the embedding degree k , which is the smallest positive integer such that $q^k \equiv 1 \pmod{n}$. Then we extend the elliptic curve to $\text{GF}(q^k)$ and mapping points P and Q to the extended curve. Random Points and Pairing: generate random points S and T on the extended curve; compute the Tate pairings and their ratio γ ; ensure $\gamma \neq 1$ and compute δ as the

ratio of Tate pairings involving Q . Solve the discrete logarithm l in the finite field $G^F(qk)$ using δ and γ . Return l – secret value.

```
E = EllipticCurve(GF(p), [a, b])
G = E(Gx, Gy)
A = E(aGx, aGy)
B = E(bGx, bGy)

info(f"Base Point: {(Gx,Gy)}")
info(f"Alice : {(aGx, aGy)}")
info(f"Bob   : {(bGx, bGy)}")

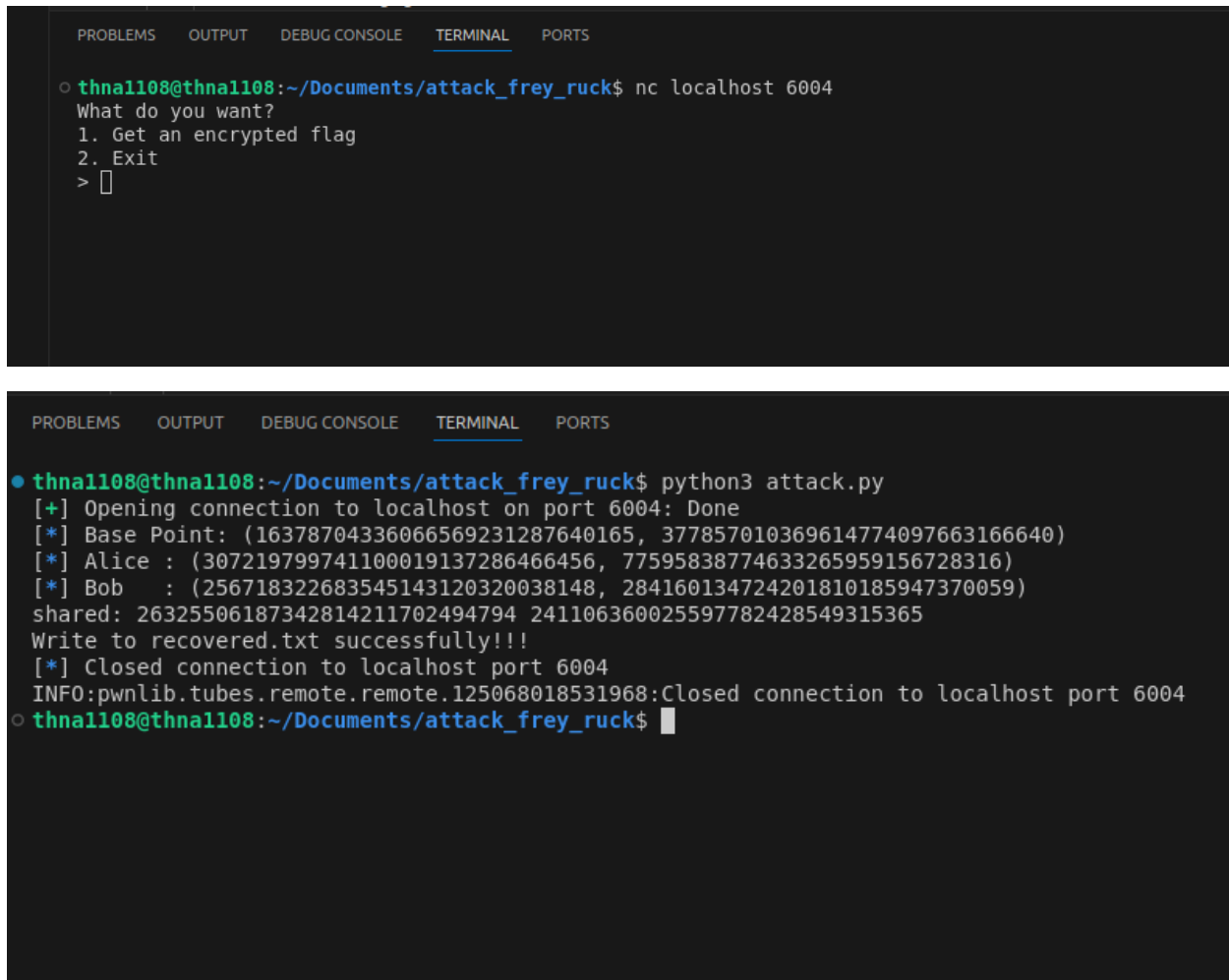
def get_embedding_degree(q, n, max_k):
    for k in range(1, max_k + 1):
        if q ** k % n == 1:
            return k
    return None

def attack(P, Q, E, max_k=100, max_tries=100):
    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)
    Qk = Ek(Q)
    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 = Qk.tate_pairing(S, n, k) / Qk.tate_pairing(T, n, k)
        logging.info(f"Computing {delta}.log({gamma})...")
        l = delta.log(gamma)
        return int(l)
    return None

def decrypt(secret: int, ciphertext: bytes):
    hash = sha256(str(secret).encode()).digest()
    iv, key = hash[:16], hash[16:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = cipher.decrypt(ciphertext)
    return decrypted

with open("output/cipher.enc", "rb") as cipher_file:
    enc = cipher_file.read()
    alice_secret = attack(G, A, E)
    shared_secret = B * alice_secret
    print(f"shared: {shared_secret[0]} {shared_secret[1]}")
    flag = unpad(decrypt(shared_secret[0], enc), 16)
    with open("recovered.txt", "wb") as recover_file:
        if flag is not None:
            recover_file.write(flag)
            print("Write to recovered.txt successfully!!!")
        else:
            print("Write to recovered.txt failed!!!")
```

This is our output:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
○ thna1108@thna1108:~/Documents/attack_frey_ruck$ nc localhost 6004
What do you want?
1. Get an encrypted flag
2. Exit
>

● thna1108@thna1108:~/Documents/attack_frey_ruck$ python3 attack.py
[+] Opening connection to localhost on port 6004: Done
[*] Base Point: (16378704336066569231287640165, 377857010369614774097663166640)
[*] Alice : (307219799741100019137286466456, 77595838774633265959156728316)
[*] Bob : (256718322683545143120320038148, 284160134724201810185947370059)
shared: 26325506187342814211702494794 241106360025597782428549315365
Write to recovered.txt successfully!!!
[*] Closed connection to localhost port 6004
INFO:pwnlib.tubes.remote.remote.125068018531968:Closed connection to localhost port 6004
○ thna1108@thna1108:~/Documents/attack_frey_ruck$
```

This approach is useful for specific elliptic curves where the embedding degree k is small, allowing the ECDLP to be translated to a finite field problem, which can often be solved more efficiently.

Notes: We realize that increasing the number of curve additions to meet the requirements of a standard curve will increase the execution time, as evidenced [here](#). Please bear with our limitation.

We have included all code of this project in [this github link](#).

PART V. DEPLOYMENT

In this project, we have figure out some threats to ECC-based cryptosystem that attackers can ruin and exploit resources because of the fact that choosing a secure elliptic curve is crucial for many attacks on elliptic curve cryptography (ECC) exploit weaknesses in the curve. These suggestions below may help to avoid attack:

- For instance, if the curve's order is small, it becomes vulnerable to attacks like Pollard's rho.
 - If the curve's order matches the order of the field, it can be targeted by the Smart attack.
 - Additionally, the Frey-Rück attack targets curves with certain properties. Specifically, this attack can be applied to elliptic curves where there exists a non-trivial endomorphism (a map from the curve to itself that preserves the group structure). Curves that have small embedding degrees or that allow efficient computation of Weil or Tate pairings can be vulnerable to the Frey-Rück attack.
 - Similarly, the Menezes-Okamoto-Vanstone (MOV) pairing attack can be executed on curves with small embedding degrees, where the discrete logarithm problem on the elliptic curve can be reduced to a discrete logarithm problem in a finite field, which is easier to solve. In practice MOV attack can be simply avoided by not using curves with small embedding degree; standardized curves are safe. Since pairings also have many constructive applications, it is possible to carefully choose curves where the cost of attacking the elliptic curve itself or the mapped finite field is the same.
 - To prevent invalid curve attacks, servers must rigorously validate all data received from users. Therefore, it is advisable to use standard curves that have been rigorously evaluated and are known to resist these types of attacks <https://safecurves.cr.yp.to/>. This includes ensuring that any points provided by users indeed lie on the specified elliptic curve.
- ⇒ Furthermore, it is crucial to regularly upgrade, update, and periodically test the security measures of cryptographic systems to maintain their integrity and security.

PART VI. REFERENCE

- 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>
- Dubois, R. (2017). Trapping ECC with invalid curve bug attacks. *Cryptology ePrint Archive.*
<https://eprint.iacr.org/2017/554>
- 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>
- Seet, M. Z. (2007). *ELLIPTIC CURVE CRYPTOGRAPHY* (Doctoral dissertation, School of Mathematics and Statistics, The University of New South Wales).
<https://www.maths.unsw.edu.au/sites/default/files/mandyseetthesis.pdf>

- Novotney, P. (2010). Weak Curves In Elliptic Curve Cryptography. *modular.math.washington.edu/edu/2010/414/projects/novotney.pdf*.
[\(<https://wstein.org/edu/2010/414/projects/novotney.pdf>\)](https://wstein.org/edu/2010/414/projects/novotney.pdf)
- Roy, S., & Khatwani, C. (2017). Cryptanalysis and improvement of ECC based authentication and key exchanging protocols. *Cryptography*, 1(1), 9.
[\(<https://www.mdpi.com/2410-387X/1/1/9>\)](https://www.mdpi.com/2410-387X/1/1/9)
- Reddy, P. M., & Vidhyapeetham, A. V. Attacks on Elliptic Curve Cryptography Implementations in Sage Math.
[\(<https://ijisrt.com/assets/upload/files/IJISRT23MAY2172.pdf>\)](https://ijisrt.com/assets/upload/files/IJISRT23MAY2172.pdf)

-----*The end*-----

This image shows a full page of dot grid paper. It features approximately 28 horizontal rows of small, evenly spaced black dots on a white background. The dots are arranged in straight lines across the width of the page, providing a guide for writing or drawing without solid lines.

LỜI CẢM ƠN